CPSC532W Homework 4

Justin Reiher
Student ID: 37291151
CWL: reiher

Link to public repository for homework 4:

`https://github.com/justinreiher/probProg_Fall2021/tree/main/CS532-HW4`

1 Program 0: BBVI Implementation Details

Implementing algorithm 11 and 12 from the book are outlined as follows: For algorithm 11, the sample and observe functions,where a new variable v is introduced which is the address to the node in question. If the node does not exist then it is added to the program environment variable σ. The implementation of the `sample` and `observe` case are as follows:

```
  elif e == 'sample*':
    Dobj = args[0]
    if not(v in sig['Q']):
      sig['Q'][v] = Dobj.make_copy_with_grads()
    dg = sig['Q'][v].make_copy_with_grads()
    ret = dg.sample()
    sig['G'][v] = grad_log_prob(dg,ret)
    logWv = Dobj.log_prob(ret) - sig['Q'][v].log_prob(ret)
    sig['logW'] = sig['logW'] + logWv.detach().nan_to_num()
  elif e == 'observe*':
    Dobj = args[0]
    ret = args[1]
    sig['logW'] = sig['logW']+Dobj.log_prob(ret).detach().
    nan_to_num()
```

Like in the textbook, the `sample` case computes the `gradLogProb` of the sample from Q:

```
def grad_log_prob(Dobj,x):
  logProb = Dobj.log_prob(x)
  logProb.backward()
  params = Dobj.Parameters()
  g = []
  for i in params:
    g.append(i.grad)

  return g
```

Since we only want the gradient, a copy of the distribution object `Dobj` is made to then compute the `gradLogProb` and then subsequently the result is `detached`. If we do not do this, then the gradients are compounding and create an exponentially growing gradient compute tree. The BBVI algorithm is implemented as follows:

```python
    #set R_0, by sampling every node from the prior and set the initial map of G and Q to empty,
    #they get populated in deterministic_eval
    for v in vertices:
        linkFuncsEval = bindingVars(R,linkFuncs[v])
        R[v]  = deterministic_eval(linkFuncsEval,sig,v)

    for v in sig['Q']:
        sig['opt'][v] = [torch.optim.Adam(sig['Q'][v].Parameters(),lr),sig['Q'][v]]
        sig['Qp'][v] = [torch.stack(sig['Q'][v].Parameters()).detach()]

    for t in range(T):
        gradients.append([])
        ret.append([])
        for s in range(num_samples):
            sig['G'] = {}
            sig['logW'] = 0
            for v in vertices:
            #    mb = getMarkovBlanket(v)
            #    for x in mb:
                linkFuncsEval = bindingVars(R,linkFuncs[v])
                R[v] = deterministic_eval(linkFuncsEval,sig,v)
            gradients[t].append(sig['G'].copy())
            logWeights[t,s] = sig['logW'].detach()

            if type(E) == str:
                retExp = R[E]
            else:
                retExp = bindingVars(R,E)
            ret[t].append(deterministic_eval(retExp,sig,[]))
        ghat = elbo_grad(gradients[t],logWeights[t,:])
        print(t)
        print(logWeights[t,:].mean())
        optimize(sig['Q'],ghat,sig['opt'],sig['Qp'],t)
       # input('before after')

    return ret,logWeights,sig['Q'],sig['Qp']
```

A topological graph sort is performed on the nodes before hand and stored. The nodes are then initially populated and for every random variable with an associated $\mathcal{Q}(v)$ gets an optimization object initialized with the parameters of the prior associated with node v. The optimizer used in this case is `Adam`. This distribution object is used to make an update step, copy over the update to $\mathcal{Q}(v)$ and continue. The optimization requires computing the weighted gradients. The gradient keys are merged together because under differing control flow it's possible that some gradients don't exist in some traces. In these instances the gradients are computed to zero:

```python
def elbo_grad(G,logW):
    #merging list of dictionaries:
    # https://stackoverflow.com/questions/3494906/how-do-i-merge-a-list-of-dicts-into-a-single-dict
    # this line is horribly unreadable, but:
    # for every key 'k' in each dictionary 'd' of list 'G'
    # insert its value 'v'.
    # note that the values in this merged gradient object are irrelevant
    # we just want to know all the keys that exists in all L gradient samples
    # because different traces may have invoked different gradient addresses (or nodes)
    Gmerg = {k: v for d in G for k, v in d.items()}
  #  return Gmerg
    #initialize my F dictionary
    #with the correct number of keys and empty lists
    F = {k: [] for k in Gmerg}
    Gs = {k: [] for k in Gmerg}
    L = len(logW)
    ghat = {}
    for v in Gmerg:
        sizeG = torch.stack(Gmerg[v]).shape
        for s in range(L):
            if v in G[s].keys():
                F[v].append(torch.stack(G[s][v])*logW[s])
            else:
                F[v].append(torch.zeros(sizeG))
                G[s][v].append(torch.zeros(sizeG))
        #this picks out all gradients from 1:L of G in a list for variable v
        GL = torch.stack([torch.stack(g[v]) for g in G])
        Fv = torch.stack(F[v])

        bhat = computeBaseline(Fv,GL)

        ghat[v] = torch.sum((Fv - bhat*GL)/L,0)

    return ghat
```

and subtracting the *baseline*. The *baseline* is computed by computing the element-wise covariance of each mini-batch trace and the weighted gradients and then summed. Any results resulting in *nan* are turned to zeros using `torch.nan_to_num`. Using `numpy cov` routine returns a `dtype=float64` result, this is truncated back to `torch.float32` (This baseline computation was discussed at length with many of my classmates)

```python
def computeBaseline(Fv,GL):
    C = []
    #Check to see if the elements of Fv are multi-dimensional
    if len(Fv[0].shape) > 1:
        n,d = Fv[0].shape
        for i in range(n):
            C_i = []
            #collect the cov(Fi,Gi) terms
            for j in range(d):
                C_ij = np.cov(Fv.detach().numpy()[:,i,j],GL.numpy()
    [:,i,j], rowvar = True)
                C_i.append(C_ij[0,1])
        C.append(C_i)
        Cov = torch.tensor(C)
        #need to get ride of NaN, sum all the cov(Fi,Gi) terms
        bhat = torch.nan_to_num(Cov.sum()/GL.var(0).sum())
    else:
        for i in range(Fv[0].shape[0]):
            C_i = np.cov(Fv.detach().numpy()[:,i],GL.numpy()[:,i],
    rowvar = True)
            C.append(C_i[0,1])
        Cov = torch.tensor(C)
        #again need to get ride of NaN
        bhat = torch.nan_to_num(Cov.sum()/GL.var(0))

    # for some reason numpy turns the above into float64,
    everything else (and default torch floating point numbers) is
    in torch.float32
    # so truncate back to torch.float32
    return torch.tensor(bhat,dtype = torch.float32)
```

The optimization step retrieves the optimization object associated with the $\mathcal{Q}(v)$ that is being optimized, a step is taken the gradients are then set to zero (same reason as to why the log-weights are detached after being computed) and the distribution object in $\mathcal{Q}(v)$ is updated with the new parameters:

```
def optimize(Q,ghat,optimizers,Qp,t):
    for v in ghat:
        [opt,Dobj] = optimizers[v]
        grad = ghat[v]
        i = 0
        for p in Dobj.Parameters():
            p.grad = -torch.nan_to_num(grad[i].detach())
            i+=1
        opt.step()
        opt.zero_grad()

        i = 0
        for p in Q[v].Parameters():
            p.data = Dobj.Parameters()[i] #+ 1e-8/(t+1)*grad[i]
            i += 1
        Qp[v].append(torch.stack(Q[v].Parameters()).detach())
```

Finally, the results of the samples, the log-weights are stored and then returned.

1.1 Program 1

Number of samples per optimization is: 200, number of optimization steps is: 350, Adam learning rate: 0.5

```
Collect samples denoted by program 1:
Elapsed time for program  1 .daphne is:  0:01:28.018433  seconds
Mean of set of samples:  tensor(7.2636)
and variance of samples:  tensor(0.0539)
```

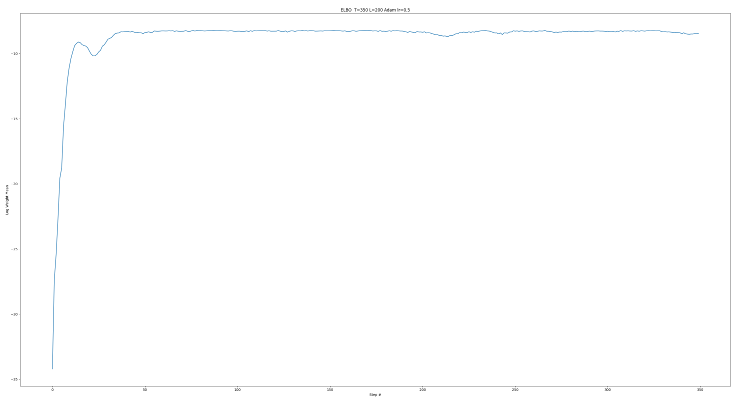

Figure 1: ELBO plot

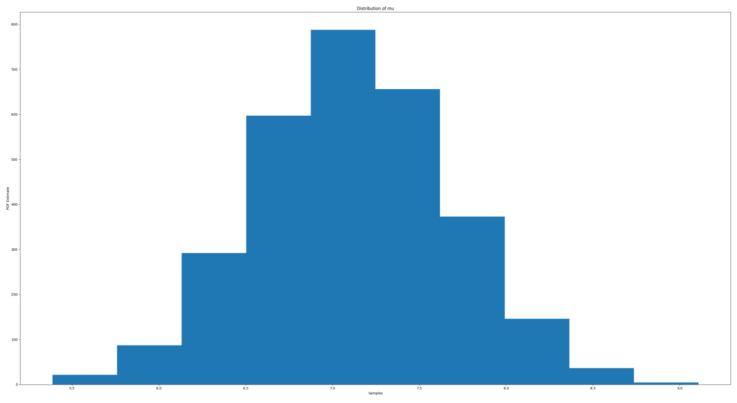

Figure 2: Estimate of Distribution μ from $\mathcal{Q}$

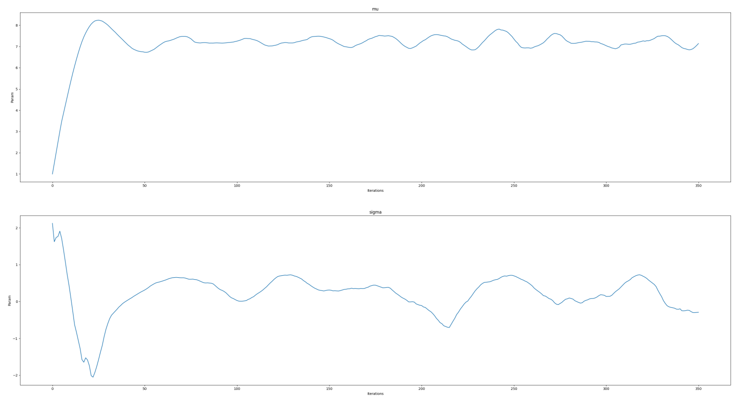

Figure 3: Parameters for $\mathcal{Q}$ for μ and Σ over iterations

1.2 Program 2

Number of samples per optimization is: 200, number of optimization steps is: 350, Adam learning rate: 0.25

```
Collect samples denoted by program 2:
Elapsed time for program  2 .daphne is:  0:03:02.441880  seconds
Mean of set of samples:  tensor([1.8603, 0.6375])
and variance of samples:  tensor([0.0843, 1.3375])
```

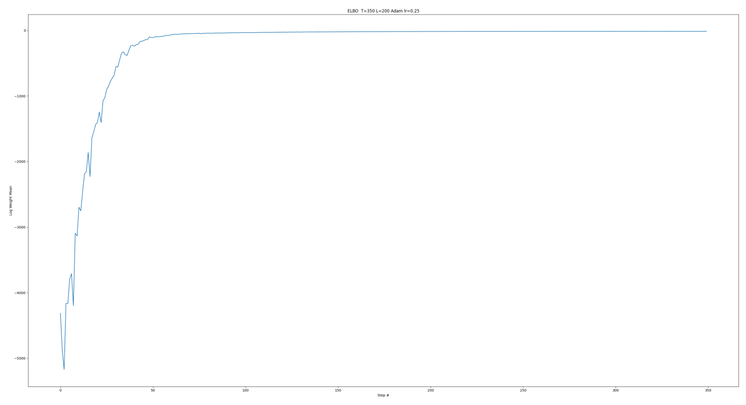

Figure 4: ELBO plot

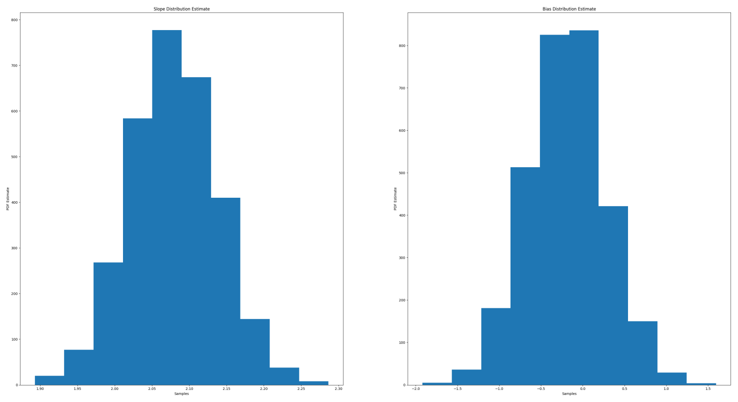

Figure 5: Estimate of Distributions for Slope and Bias from $\mathcal{Q}$ respectively

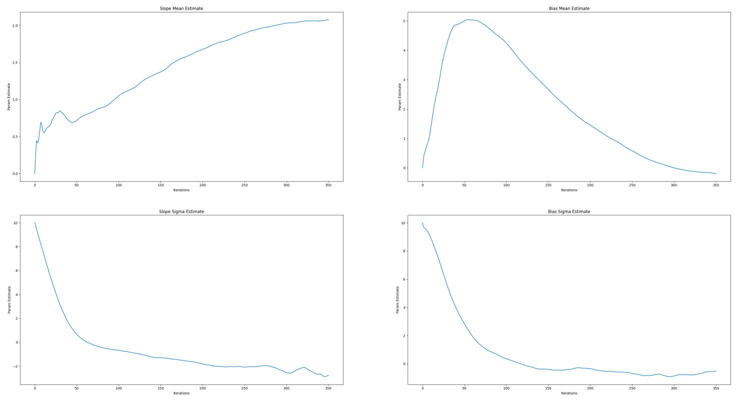

Figure 6: Parameters for $\mathcal{Q}$ for Slope and Bias over iterations

1.3 Program 3

Number of samples per optimization is: 25, number of optimization steps is: 350, Adam learning rate: 0.05

```
Collect samples denoted by program 3:
Elapsed time for program  3 .daphne is:  0:01:49.841174  seconds
Mean of samples:  tensor(0.5729)
and variance of samples:  tensor(0.2132)
```

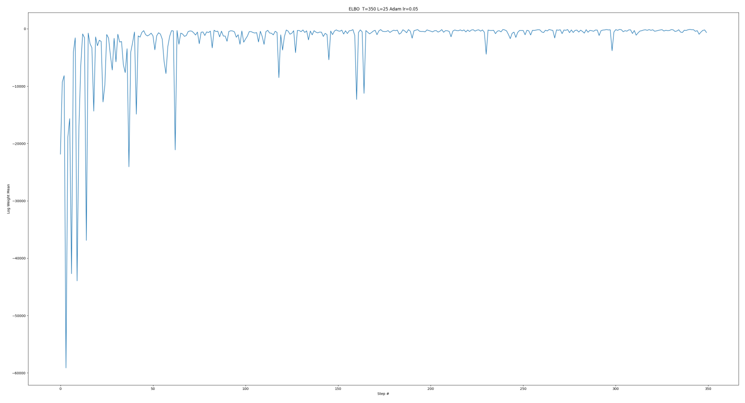

Figure 7: ELBO plot

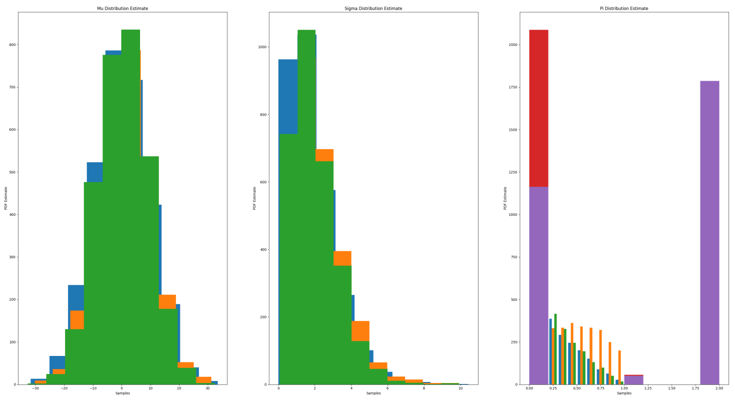

Figure 8: Estimates of Distributions for the 3 Modes (μ and Σ) and Π

During optimization there are sudden jumps where the ELBO drops significantly likely arising from the fact that the modes switch, i.e. sampling from one mode for awhile and then sampling from another. In these instances a sudden drop in the importance weights will arise from estimating the distribution from the "wrong" place. The convergence back to the previous ELBO is quick, however this bouncing around makes optimization hard.

Optimizing with a mix of discrete and continuous random variables is a difficult problem to optimize. Here the Dirichelet distribution would make pathwise gradient optimization impossible because of the discontinuity when switching from one mode to the other.

1.4 Program 4

Number of samples per optimization is: 150, number of optimization steps is: 200, Adam learning rate: 0.1

```
Collect samples denoted by program 4:
Elapsed time for program  4 .daphne is:  1:14:39.898121  seconds
Mean of samples  W0 :  tensor([[ 0.1319],
[ 0.2567],
[-0.0351],
[ 0.0944],
[-0.1974],
[-0.1980],
[ 0.2262],
[-0.0848],
[-0.0321],
[ 0.0394]])
and variance of samples  W0 :  tensor([[0.4367],
[0.3055],
[0.3614],
[0.4822],
[0.2883],
[0.2392],
[0.3769],
[0.2462],
[0.3062],
[0.3234]])
Mean of samples  b0 :  tensor([[ 0.2149],
[ 0.2840],
[ 0.0512],
[-0.1019],
[ 0.2738],
[-0.0108],
[ 0.0631],
[ 0.1515],
[-0.2039],
[-0.1049]])
and variance of samples  b0 :  tensor([[0.3262],
[0.2735],
[0.3409],
[0.3788],
[0.3440],
[0.3860],
[0.3567],
[0.4167],
[0.3477],
[0.3407]])
```

```
Mean of samples  W1 :  tensor([[-0.2270,  0.0882, -0.0121,  0.1743,  0.1357, -0.0302,  0.103
-0.2197,  0.2341],
[ 0.0335,  0.1321, -0.2206,  0.0070,  0.3172, -0.0196, -0.1112,  0.3747,
0.1385, -0.1991],
[ 0.1750, -0.2587,  0.0133, -0.2125,  0.1243,  0.0346, -0.2772,  0.2673,
-0.1649, -0.1107],
[ 0.1240,  0.0122,  0.1321, -0.1090, -0.0051, -0.0048, -0.0800,  0.0623,
-0.1135, -0.2212],
[ 0.1334,  0.0728, -0.0356,  0.0396,  0.1656, -0.1858,  0.0008,  0.1235,
-0.1176,  0.0950],
[-0.1629, -0.0923, -0.0512, -0.2433,  0.1016, -0.0403,  0.0888, -0.1327,
-0.1983, -0.2119],
[-0.2335, -0.1774, -0.0861, -0.1186, -0.1272, -0.0608, -0.0534, -0.0140,
0.0414,  0.1305],
[-0.0292, -0.0647,  0.1377,  0.2108, -0.0375,  0.0988,  0.0780,  0.1071,
-0.1232, -0.3151],
[-0.3171,  0.0690,  0.0247, -0.2124, -0.0200,  0.0238,  0.1399, -0.0331,
0.0021, -0.0122],
[-0.0929,  0.1975,  0.0876,  0.0975, -0.2569,  0.0861, -0.0492,  0.2758,
0.0288, -0.1190]])
and variance of samples  W1 :  tensor([[0.3592, 0.3715, 0.5487, 0.2535, 0.3442, 0.4236, 0.29
0.3786],
[0.4458, 0.3312, 0.4317, 0.3211, 0.3217, 0.3912, 0.3933, 0.4606, 0.4516,
0.3577],
[0.2782, 0.3205, 0.5102, 0.4104, 0.3093, 0.3729, 0.4369, 0.2085, 0.4021,
0.2435],
[0.4582, 0.3734, 0.3329, 0.3571, 0.3348, 0.2480, 0.3009, 0.2123, 0.3273,
0.4235],
[0.2321, 0.4526, 0.3034, 0.2971, 0.2951, 0.4715, 0.4704, 0.3166, 0.2394,
0.4468],
[0.2294, 0.1911, 0.3265, 0.3510, 0.3685, 0.3401, 0.3739, 0.5084, 0.4844,
0.3621],
[0.3258, 0.1964, 0.3032, 0.4131, 0.2503, 0.3646, 0.2690, 0.2913, 0.4242,
0.3185],
[0.2641, 0.4064, 0.3364, 0.4167, 0.3630, 0.3893, 0.3680, 0.4196, 0.2985,
0.4318],
[0.4190, 0.3290, 0.2814, 0.4302, 0.3249, 0.2755, 0.5409, 0.2783, 0.2799,
0.4033],
[0.2503, 0.4304, 0.2710, 0.3762, 0.4537, 0.3670, 0.4089, 0.3531, 0.5503,
0.3745]])
Mean of samples  b1 :  tensor([[ 0.0267],
[-0.1561],
[ 0.0901],
[-0.2309],
[ 0.0494],
[-0.0263],
[-0.0623],
[-0.0492],
[ 0.0212],
[-0.1859]])
and variance of samples  b1 :  tensor([[0.5040],
[0.4294],
[0.4290],
[0.3916],
[0.2523],
[0.4642],
[0.3364],
[0.3017],
[0.2614],
[0.3772]])
```

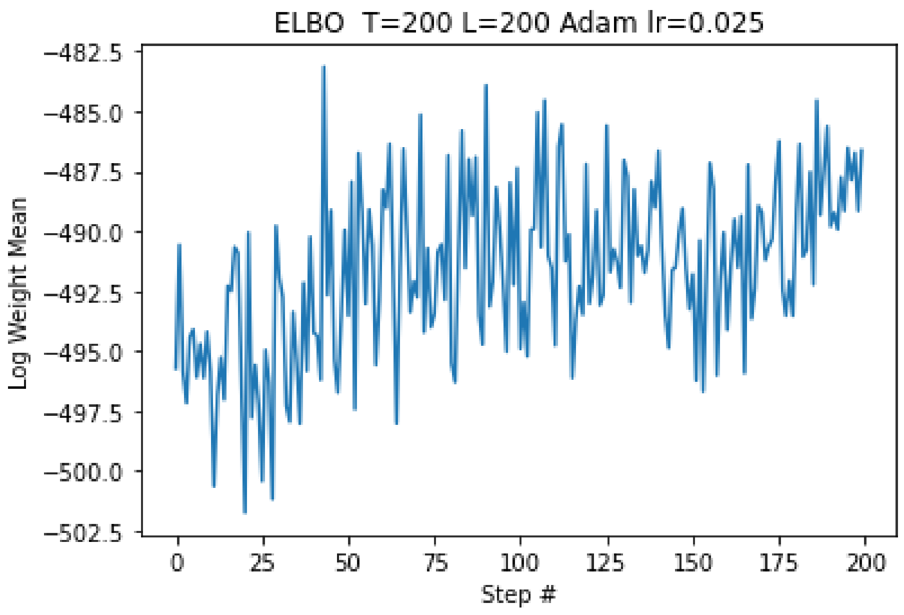

Figure 9: ELBO plot

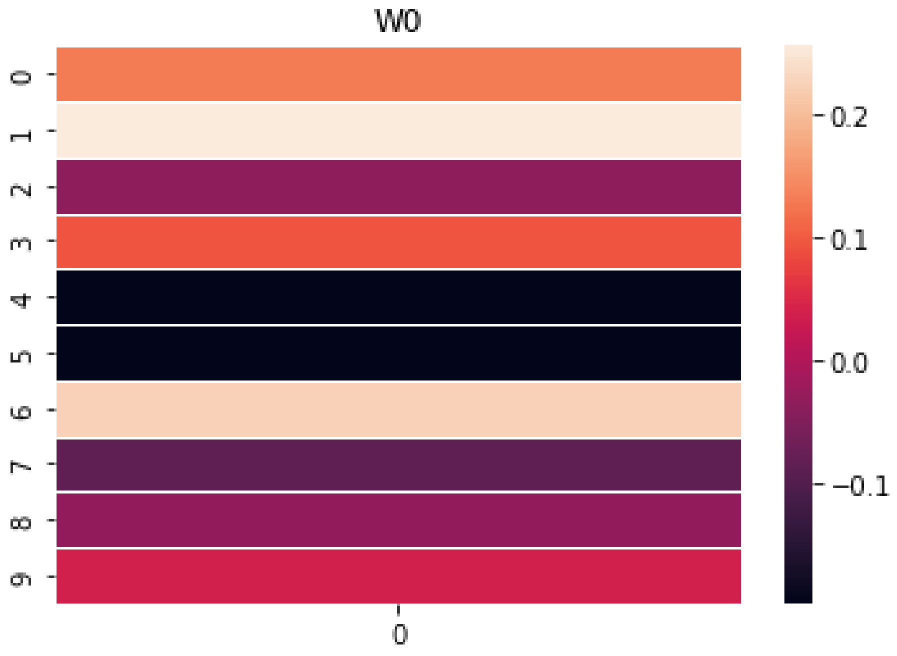

Figure 10: Heatmap of expectation of W_0

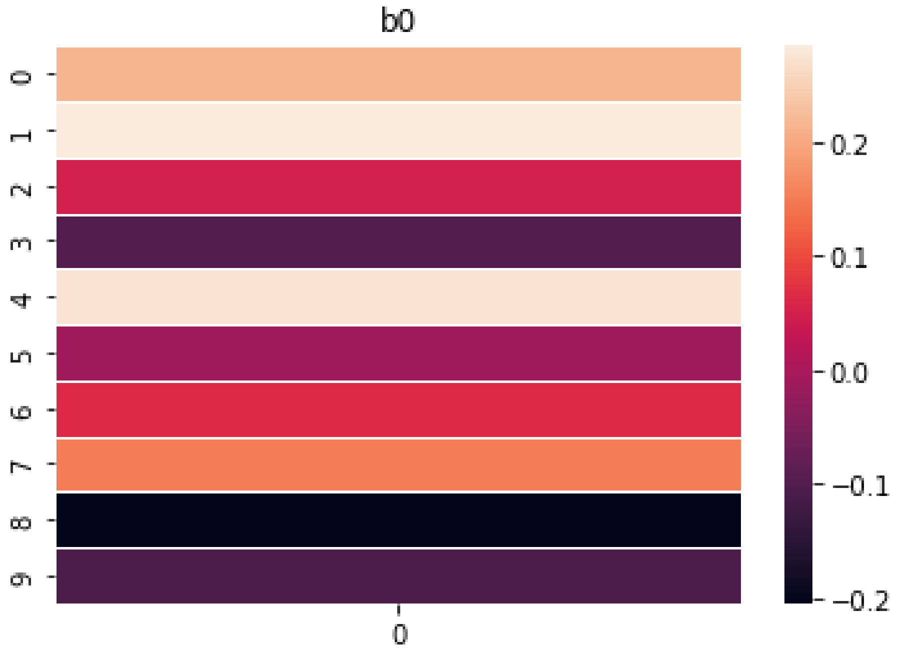

Figure 11: Heatmap of expectation of b_0

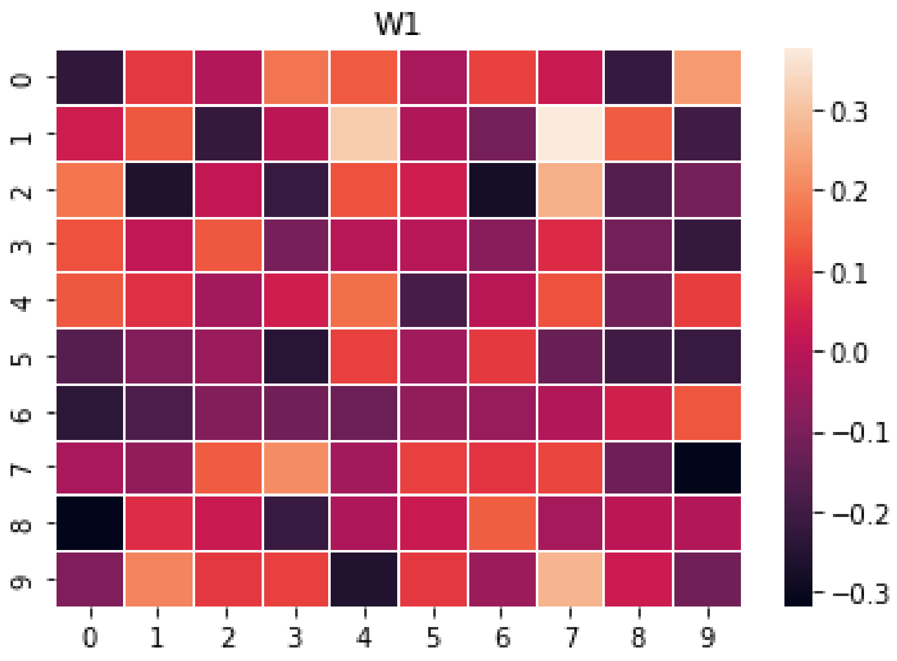

Figure 12: Heatmap of expectation of W_1

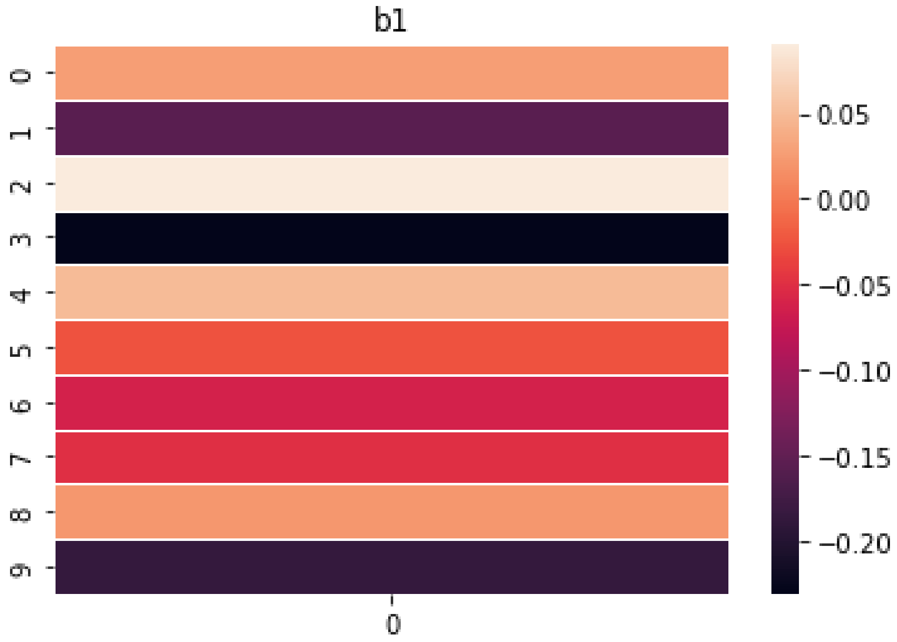

Figure 13: Heatmap of expectation of b_1

Gradient descent optimization assumes a fix structure to the parameters of the network and adjust weights of the network. In BBVI we are learning the distribution that the parameters come from, i.e. once we know the distribution for the parameters of the network we can instantiate networks that are all supposed to solve the same problem.

1.5 Program 5

For this program I am going to get creative about approximating the *Uniform* distribution using tanh functions. I have used Wolfram Alpha to check some concrete examples, but a derivation to my approach:

$$\begin{aligned} f_+(x) &= 0.5\tanh(x) + 0.5 \in [0,1] \\ \lim_{x\to\infty} f_+(x) &= 1 \\ \lim_{x\to-\infty} f_+(x) &= 0 \end{aligned} \tag{1}$$

If we define $f_-(x) = -0.5\tanh(x) + 0.5$, then $x \to \infty$ $f_-(x) = 0$ and $x \to -\infty$ $f_-(x) = 1$. We can scale how fast the transition occurs with a parameter $s > 0$ and translate where the transition occurs with a:

$$f(x) = 0.5\tanh(s(x-a)) + 0.5 \tag{2}$$

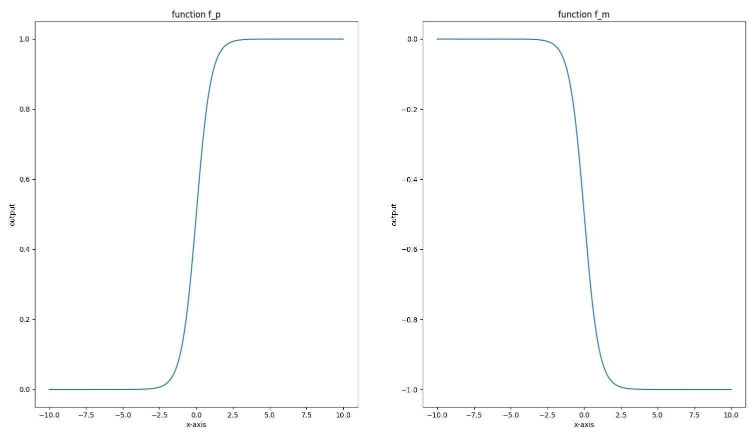

Figure 14: tanh functions to estimate `Uniform` distribution

To approximate `Uniform(0,1)` we compose f_+ and f_- as follows:

$$\begin{aligned} \text{Uniform}(x)[0,1] &\approx 0.5\,(\tanh(s(x)) - \tanh(s(x-1))) \\ \hat{\mathcal{U}}(x)[a,b] &= \tfrac{1}{b-a}\,(0.5\,(\tanh(s(x+a)) - \tanh(s(x-b)))) \end{aligned} \tag{3}$$

The question is the above a real probability density function? Using Wolfram Alpha and as an example $s = 10000$, $b = 5$ and $a = -5$ then $\int_{-\infty}^{\infty} \hat{\mathcal{U}}(x)[a,b]dx = 1$.

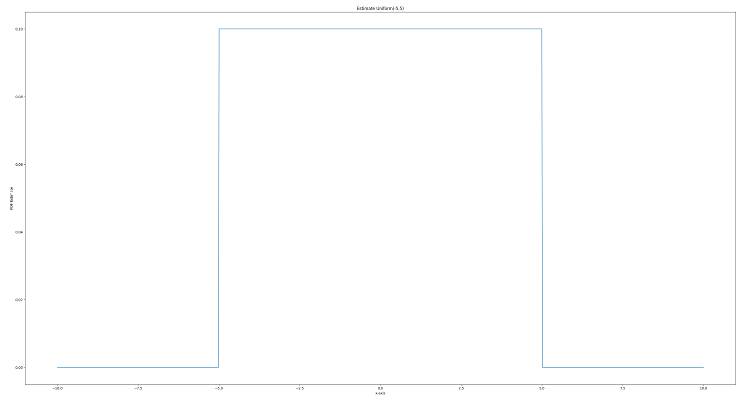

Figure 15: Estimate of `Uniform`(-5,5) with $s = 10000$

Why use tanh functions? They are differentiable everywhere, as $s \to \infty$ the approximation gets closer and closer to the real distribution. However making s

large means the gradient vanishes very quickly when scored outside the support of the distribution. Thus picking s to have smooth and non-zero derivatives outside the support is essential. To this end to make this program work a scale $s = 1$ works nicely. When sampling from this modified distirbution, we sample from `Uniform` in the support of the expected distribution, but score with the approximation to allow for scoring outside the support. The scoreable `Uniform` distribution outside it's support is implemented as follows:

```python
class Uniform:

    def __init__(self,low,high,copy=False):
        if low >= high:
            lowNew = high.clone().detach().requires_grad_()
            highNew = low.clone().detach().requires_grad_()
            low = lowNew
            high = highNew

        self.low = low
        self.high = high
    def Parameters(self):
        return [self.low,self.high]

    def make_copy_with_grads(self):
        low,high = [p.clone().detach().requires_grad_() for p in
    self.Parameters()]
        return Uniform(low,high)

    def sample(self):
        return dist.Uniform(self.low,self.high).sample().nan_to_num
    ()

    def log_prob(self,x):
        s = 1
        return torch.log(0.5/(self.high - self.low)*(-torch.tanh(s
    *(x-self.high))+torch.tanh(s*(x+self.low))))
```

```
Collect samples denoted by program 5:
Elapsed time for program  5 .daphne is:  0:01:02.974768  seconds
Mean of samples:  tensor(6.5028)
and variance of samples:  tensor(1.0372)
{'sample1': Normal(loc: 0.3225950300693512, scale: 6.593713283538818), 'sample2': <distribut
```

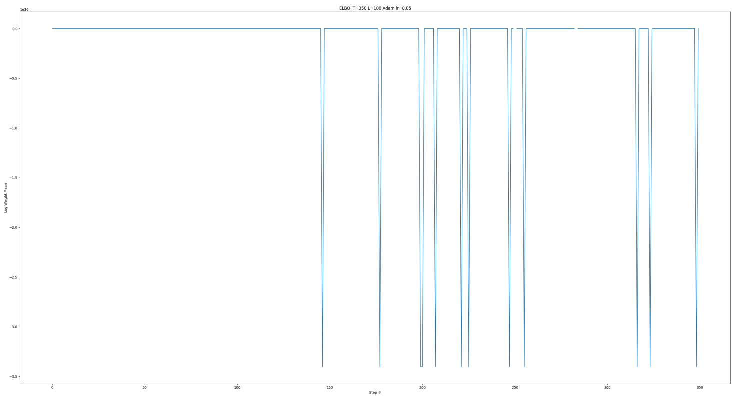

Figure 16: ELBO plot

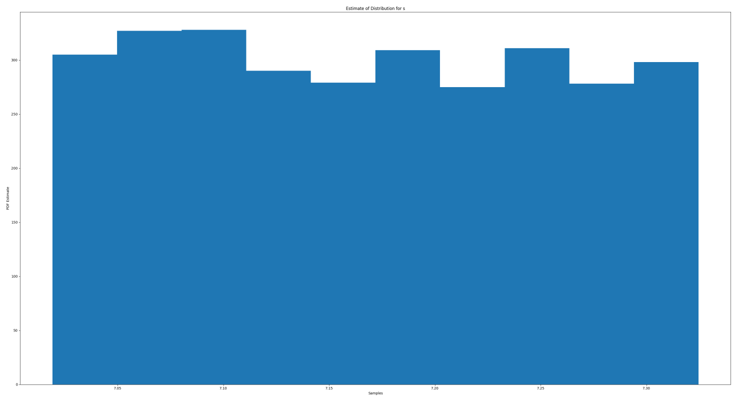

Figure 17: Distribution Estimate for s from $\mathcal{Q}$

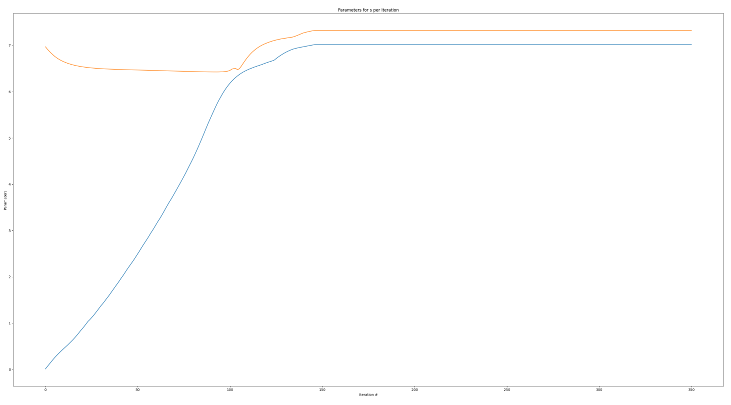

Figure 18: Parameters high and low for $\mathcal{Q}$ over iterations

What we see in the above is that the distribution returned is a `Uniform` tightly distributed around the value 7. This makes sense as all our observations in P5 are the value 7. It should be noted that making s larger although is more accurate to the real `Uniform` object scoring outside the support rapidly goes to 0 which means log-Prob $\to -\infty$. So a compromise is required in order to make progress when scoring outside the support vs accuracy.