CPSC 532W Homework 3

Naomi Graham

March 9, 2021

All the code can be found on: `https://github.com/n6graham/cpsc532_hw3`.

1 Importance sampling

1.1 Code

For Importance Sampling I used the evaluation-based implementation based off of Jason's HW2 code.

```python
def evaluate_program(ast):
    """Evaluate a program as desugared by daphne, generate a sample from the prior
    Args:
        ast: json FOPPL program
    Returns: sample from the prior of ast
    """
    PROCS = {} #program procedures
    for i in range(len(ast)-1):
        proc = ast[i]
        proc_name, proc_arg_names, proc_expr = proc[1], proc[2], proc[3]
        PROCS[proc_name] = (proc_arg_names,proc_expr)

    #print(PROCS)
    # expr is ast[-1]


    def eval(expr, sigma, scope):
        if is_const(expr, scope):
            if type(expr) in [int, float]:
                expr = torch.Tensor([expr]).squeeze()
            return expr, sigma
        elif is_var(expr, scope):
            return scope[expr], sigma
        elif is_let(expr, scope):
            var_name, sub_expr, final_expr = expr[1][0], expr[1][1], expr[2]
            var_value, sigma = eval(sub_expr, sigma, scope)
            return eval(final_expr, sigma, {**scope, var_name: var_value})
        elif is_if(expr,scope):
            cond_expr, true_expr, false_expr = expr[1], expr[2], expr[3]
            cond_value, sigma = eval(cond_expr, sigma, scope)
            if cond_value:
                return eval(true_expr, sigma, scope)
            else:
                return eval(false_expr, sigma, scope)
```

```python
        elif is_sample(expr,scope):
            dist_expr = expr[1]
            dist_obj, sigma = eval(dist_expr,sigma,scope)
            return dist_obj.sample(), sigma
        elif is_observe(expr,scope):
            # need to do something special here
            dist_expr, obs_expr = expr[1], expr[2]
            dist_obj, sigma = eval(dist_expr,sigma,scope)
            obs_value, sigma = eval(obs_expr,sigma,scope)
            sigma['logW'] = sigma['logW'] + dist_obj.log_prob(obs_value)
            return obs_value, sigma
        else:
            proc_name = expr[0]
            consts = []
            for i in range(1,len(expr)):
                const, sigma = eval(expr[i],sigma,scope)
                consts.append(const)
            if proc_name in PROCS:
                proc_arg_names, proc_expr = PROCS[proc_name]
                new_scope = {**scope}
                for i, name in enumerate(proc_arg_names):
                    new_scope[name] = consts[i]
                return eval(proc_expr, sigma, new_scope)
            else:
                return PRIMITIVES[proc_name](*consts), sigma


    return eval(ast[-1],{'logW':0}, {})

```

I also defined some extra functions: likelihood-weighting, compute-expectation, and compute-variance.

```python
def likelihood_weighting(L,ast):
    weighted_samples = []

    for i in range(0,L):
        r, sigma = evaluate_program(ast)
        #r, sigma = eval(e, {'logW':0}, [])
        logW = sigma['logW']
        weighted_samples.append((r,logW))

    return weighted_samples

```

```python
def compute_expectation(weighted_samples):
    '''
    using a stream of weighted samples, compute according
    to eq 4.6
    '''
    L = len(weighted_samples)
    log_weights = [weighted_samples[i][1] for i in range(0,L)]
    weights = np.exp(np.array(log_weights))
    print(weights)
    r = [ weighted_samples[i][0] for i in range(0,L)]
    denom = sum(weights)
    print("denominator is", denom)
    numerator = sum( [r[i] * weights[i] for i in range(0,L) ])
```

```
    #numerator = sum( [weighted_samples[i][0] * weighted_samples[i][1] for i in range(0,L) ])

    return numerator/denom

```

```
def compute_variance(weighted_samples, mu):

    L = len(weighted_samples)
    log_weights = [weighted_samples[i][1] for i in range(0,L)]
    weights = np.exp(np.array(log_weights))
    r = [ weighted_samples[i][0] for i in range(0,L)]
    denom = sum(weights)
    numerator = sum( [ (torch.square(r[i]) - torch.square(mu)) * weights[i] for i in range(0,L) ])
    return numerator/denom

```

1.2 results

The values returned are:

Program 1:
expectation is: tensor(7.2086) variance is tensor(0.7626)
Program 2
expectation is: tensor([2.1571, -0.5566]) variance is tensor([0.0561, 0.8358])
Program 3:
expectation is: tensor(0.7429) variance is tensor(0.1910)
Program 4:
expectation is: tensor(0.3221) variance is tensor(0.2183)

2 MH within Gibbs

2.1 code

For MH withing Gibbs I used a graph-based implementation based off of Jason's HW2 code.

```
def deterministic_eval(exp):
    "Evaluation function for the deterministic target language of the graph based representation."
    if type(exp) is list:
        op = exp[0]
        args = exp[1:]
        return env[op](*map(deterministic_eval, args))
    elif type(exp) in [int, float]:
        # We use torch for all numerical objects in our evaluator
        return torch.Tensor([float(exp)]).squeeze()
    elif type(exp) is torch.Tensor:
        return exp
    elif type(exp) is bool:
        return torch.tensor(exp)
    else:
        print("expression is:", exp)
        print(type(exp))
        raise Exception("Expression type unknown.", exp)
```

```
def topological_sort(nodes, edges):
    result = []
    visited = {}
    def helper(node):
        if node not in visited:
            visited[node] = True
            if node in edges:
                for child in edges[node]:
                    helper(child)
            result.append(node)
    for node in nodes:
        helper(node)
    return result[::-1]

def plugin_parent_values(expr, trace):
    if type(expr) == str and expr in trace:
        return trace[expr]
    elif type(expr) == list:
        return [plugin_parent_values(child_expr, trace) for child_expr in expr]
    else:
        return expr

def sample_from_joint(graph):
    "This function does ancestral sampling starting from the prior."
    # TODO insert your code here
    """
    1. Run topological sort on V using V and A, resulting in an array of v's
    2. Iterate through sample sites of the sorted array, and save sampled results on trace dictionary using P and Y
    - If keyword is sample*, first recursively replace sample site names with trace values in the expression from P. Then, run deterministic_eval.
    - If keyword is observe*, put the observation value in the trace dictionary
    3. Filter the trace dictionary for things sample sites you should return
    """
    procs, model, expr = graph[0], graph[1], graph[2]
    nodes, edges, links, obs = model['V'], model['A'], model['P'], model['Y']
    sorted_nodes = topological_sort(nodes, edges)

    sigma = {}
    trace = {}
    for node in sorted_nodes:
        keyword = links[node][0]
        if keyword == "sample*":
            link_expr = links[node][1]
            link_expr = plugin_parent_values(link_expr, trace)
            dist_obj  = deterministic_eval(link_expr)
            trace[node] = dist_obj.sample()
        elif keyword == "observe*":
            trace[node] = obs[node]

    expr = plugin_parent_values(expr, trace)
    return deterministic_eval(expr), sigma, trace
```

I added the function MH-Gibbs which also contains functions accept and Gibbs-step.

```python
def MH_Gibbs(graph, numsamples):
    model = graph[1]
    vertices = model['V']
    arcs = model['A']
    links = model['P'] # link functions aka P

    # sort vertices for ancestral sampling
    V_sorted = topological_sort(vertices,arcs)

    def accept(x, cX, cXnew, Q):
        # compute acceptance ratio to decide whether
        # we keep cX or accept a new sample/trace cXnew
        # cX and cXnew are the proposal mappings (dictionaries)
        # which assign values to latent variables

        # cXnew corresponds to the values for the new samples

        # take the proposal distribution for the current vertex
        # this is Q(x)
        Qx = Q[x][1]

        # we will sample from this with respect to cX and cXnew

        # the difference comes from how we evaluate parents
        # plugging into eval
        p = plugin_parent_values(Qx,cX)
        pnew = plugin_parent_values(Qx,cXnew)

        # p = Q(x)[X := \mathcal X]
        # p' = Q(x)[X := \mathcal X']
        # note that in this case we only need to worry about
        # the parents of x to sample from the proposal


        # evaluate
        d = deterministic_eval(p) # d = EVAL(p)
        dnew = deterministic_eval(pnew) #d' = EVAL(p')

        ### compute acceptance ratio ###

        # initialize log alpha
        logAlpha = dnew.log_prob(cXnew[x]) - d.log_prob(cX[x])

        ### V_x = {x} \cup {v:x \in PA(v)} ###
        startindex = V_sorted.index(x)
        Vx = V_sorted[startindex:]

        # compute alpha
        for v in Vx:
            Pv = links[v] # getting a bug here
            v_exp = plugin_parent_values(Pv,cX) #same as we did for p and pnew
            v_exp_new = plugin_parent_values(Pv,cXnew)
            dv_new = deterministic_eval(v_exp_new[1])
            dv = deterministic_eval(v_exp[1])


            ## change below
```

```
            logAlpha = logAlpha + dv_new.log_prob(cXnew[v])
            logAlpha = logAlpha - dv.log_prob(cX[v])
        return torch.exp(logAlpha)


    def Gibbs_step(cX,Q):
        # here we need a list of the latent (unobserved) variables
        Xobsv = list(filter(lambda v: links[v][0] == "sample*", V_sorted))

        for u in Xobsv:
            # here we are doing the step
            # d <- EVAL(Q(u) [X := \cX])
            # note it suffices to consider only the non-observed variables
            Qu = Q[u][1]
            u_exp = plugin_parent_values(Qu,cX)
            dist_u = deterministic_eval(u_exp).sample()
            cXnew = {**cX}
            cX[u] = dist_u

            #compute acceptance ratio
            alpha = accept(u,cX,cXnew,Q)
            val = Uniform(0,1).sample()

            if val < alpha:
                cX = cXnew
        return cX


    Q = links # initialize the proposal with P (i.e. using the prior)
    cX_list = [ sample_from_joint(graph)[2] ] # initialize the state/trace

    for i in range(1,numsamples):
        cX_0 = {**cX_list[i-1]} #make a copy of the trace
        cX = Gibbs_step(cX_0,Q)
        cX_list.append(cX)

    samples = list(map(lambda cX: deterministic_eval(plugin_parent_values(graph[2], cX)), cX_list))
    #samples = [ deterministic_eval(plugin_parent_values(graph[2],X)) for X in cX_list ]

    return samples
```

2.2 Results

Here are the results from running MH Gibbs for programs 1 through 4:

```
==== running program 1 =====
Program 1 mean:  6.4470267
Program 1 variance:  0.026307987
Total run time:  26.21826195716858
```

```
 ==== running program 2 =====
Program 2 slope mean is:  1.6526821
Program 2 bias mean is:  1.3684516
Program 2 slope variance is:  0.07386868
Program 2 bias variance is:  0.61285996
Total run time:  55.65074110031128
```

```
 ==== running program 3 ====
Program 3 mean is:  0.9986666666666667
Program 3 variance is:  0.001331555555555556
total run time:  579.4399008750916
```

```
 ==== running program 4 ====
Program 4 mean is:  2.5e-05
Program 4 variance is:  2.4999375e-05
total run time:  575.306489944458
```

2.2.1 Program 1

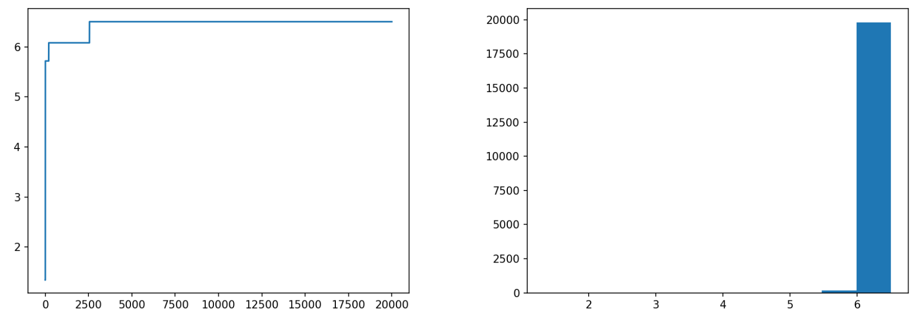

Figure 1: trace plot (left) and histogram (right)

2.2.2 Program 2

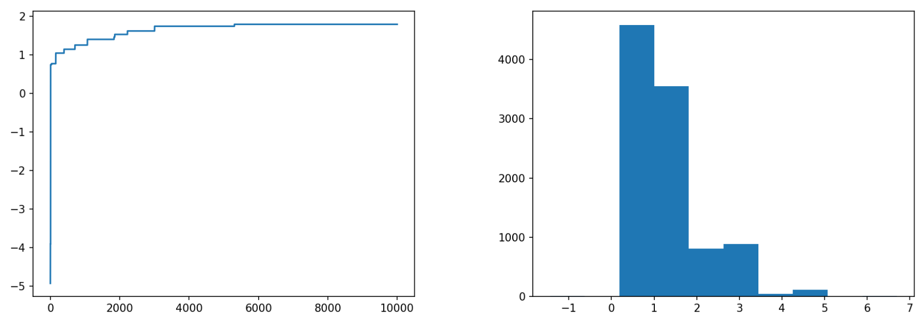

Figure 2: Program 2 (bias): trace plot (left) and histogram (right)

2.2.3 Program 3

2.2.4 Program 4

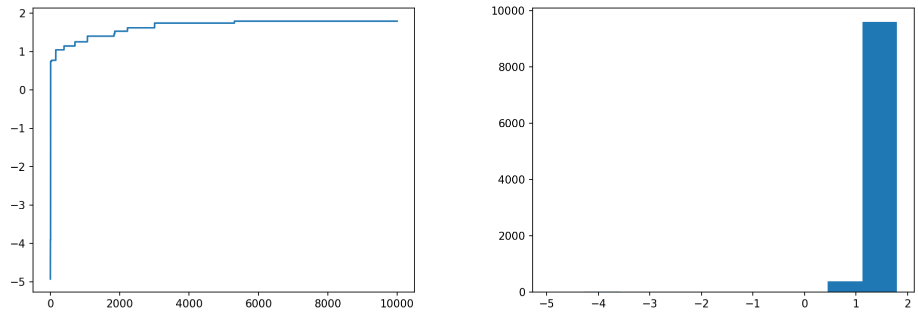

Figure 3: Program 2 (slope): trace plot (left) and histogram (right)

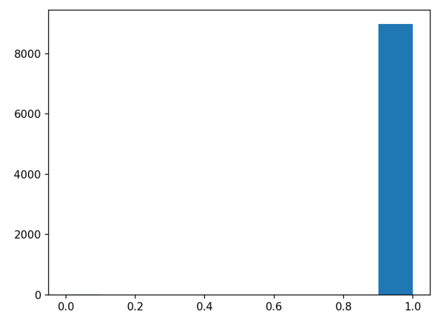

Figure 4: Program 3 histogram

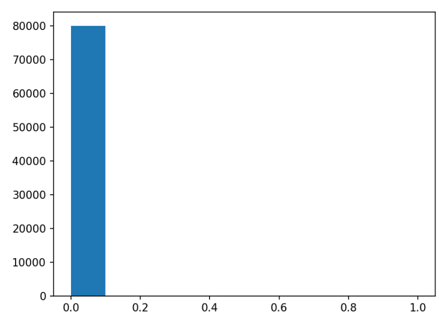

Figure 5: Program 4 histogram