CPSC532W Homework 2

Justin Reiher
Student ID: 37291151
CWL: reiher

Link to public repository for homework 2:

`https://github.com/justinreiher/probProg_Fall2021/tree/main/CS532-HW2`

All code is written in Python using the Pytorch framework.

1 Evaluation Based Sampling

This part implements the semantics described in algorithm 6 from the textbook. There are some differences in the order in which the matching occurs. Particularly when variables get matched and bound to the local context. These will be highlighted in the following:

1.1 Global Procedures ρ

The evaluator starts with empty local context, sigma and procedure list ρ. When an abstract syntax tree is given to the evaluator it first breaks the tree into individual procedures and evaluates them in order. This is required as calls to procedures that are not yet defined would fail, and so this ordering is preserved.

The global context of the evaluator is stored in:

```
#global state
sig = []
var = {}
rho = []
```

Using a singleton like pattern to grab the procedure

```
    p = rho
    if p == []: #start with populating the procedure definitions,
    singleton like pattern used here
        for i in range(0,len(ast)):
            p.append(ast[i])
            ret,s = evaluate_program(p[i])
        p.clear() # after the program is run clear the procedure
    definitions
        s.clear() # reset the world state
        var.clear() # clear the variables
```

Note that after the evaluation is complete the global context is reset for the next program.

1.2 Defining the context in the host language

Python is the host language used to implement algorithm 6 from the textbook. The white list of primitive functions used are:

```
operations = {'sqrt': torch.sqrt,
              '+': torch.add,
              '-': torch.sub,
              '/': torch.div,
              '*': torch.mul,
              'exp': torch.exp,
              '>': torch.greater,
              '<': torch.less,
              'normal': dist.Normal,
              'uniform': dist.Uniform,
              'exponential': dist.Exponential,
              'beta': dist.Beta,
              'discrete':dist.Categorical,
              'vector': torch.stack,
              'mat-transpose': torch.t,
              'mat-add': torch.add,
              'mat-mul': torch.matmul,
              'mat-repmat': None,
              'mat-tanh': torch.tanh,
              'get': None,
              'put': None,
              'first': None,
              'last': None,
              'append': None,
              'hash-map': None}
```

Where *None* is used, these cases are handled individually where a Pytorch routine is not available.

Constant values of *ints* and *floats* are converted to Pytorch tensors:

```
        if type(e) == int or type(e) == float:
            ret = torch.tensor(float(e))
```

Matching to an *if* expression evaluates to:

```
        elif e[0] == 'if':
            testExp,s = evaluate_program(e[1])
            if testExp:
                ret,s = evaluate_program(e[2])
            else:
                ret,s = evaluate_program(e[3])
```

Updating local variable bindings via *let* expressions evaluate to, where the dictionary of variables are updated in *var*. After expression e_2 is complete the variables are destroyed to preserve scoping.

```
        elif e[0] == 'let':

            letBlock = e[1]
```

```
            var[letBlock[0]],s = evaluate_program(letBlock[1])
            ret,s = evaluate_program(e[2])
            del var[letBlock[0]] # get ride of the bound variable
    afterwards, it is no longer in scope after the let block is
    complete
```

The *sample* and *observe* evaluation are similar, where in this case the *observed* value y is ignored. If the distribution object cannot be resolved because we are defining a user-defined function, the distribution object returns an empty list.

```
        elif e[0] == 'sample':
            d = e[1] # get the distribution declaration
            try:
                Dobj,s = evaluate_program(d) # assign the
    distribution object
                if Dobj == []:
                    #variable not set, must be in a defn
                    ret = []
                else:
                    ret = Dobj.sample() #sample from that
    distribution object with the associated parameters
            except: #if the distribution object exists as a
    variable, then the parameters will have been set
                Dobj = var[d]
                ret = Dobj.sample()


        elif e[0] == 'observe':
            d = e[1] # get the distribution declaration
            y = e[2] # here I don't care about y in this assignment
    , but put here for the future
            try: #first try to get a distribution object from the
    known distributions directly
                 #otherwise the distribution object may be a
    variable
                Dobj,s = evaluate_program(d) #assign the
    distribution object
                if Dobj == []:
                    #variable not set, must be in a defn
                    ret = []
                else:
                    ret = Dobj.sample() #This is what we are doing
    now, but will be replaced later
            except:
                Dobj = var[d] #if retrieved from variable, then
    arguments already set
                ret = Dobj.sample()
```

When matching to a list of expressions, expressions e_1 to e_n are evaluated and stored in a list of arguments.

```
        elif type(e) == list:
            args = []
            for i in range(1,len(e)):
                r,s = evaluate_program(e[i]) #only interested in
    the returned expression
```

```
                args.append(r) #wrap each experession as a program
    that needs to be unpacked
```

Then we try to fetch the operation e_0 from the white list of primitive functions, where the white list dictionary of *None* are defined.

```
                args.append(r) #wrap each experession as a program
    that needs to be unpacked

            try: #determine if this is a built in primitive
    function

                assert(e[0] in operations.keys())

                op = operations[e[0]] #retrieve a built-in function
                if e[0] == 'vector': #this is a constructor and so
    the arguments are not broadcast
                    try:
                        ret = op(args)
                    except:
                        ret = args
                elif e[0] == 'get': #this is a getter method
                    #arguments takes the form, (dict,index)
                    myDict = args[0]
                    if type(e[2]) == int:
                        ret = myDict[e[2]] #here we want the non-
    tensor (integer or original key to index the dict)
                    else:
                        ret = myDict[int(args[1])]
                elif e[0] == 'put': #this is a setter method
                    #arguments takes the form, (dict,index,value)
                    argsCopy = args.copy() # if we don't do this,
    then the original arguments will be modified!
                    myDict = argsCopy[0]
                    if type(e[2]) == int:
                        myDict[e[2]] = args[2]
                    else:
                        myDict[int(args[1])] = args[2]
                    ret = myDict   #return the new dictionary
                elif e[0] == 'first':
                    #arguments take the form, (vec), return the
    first element
                    vec = args[0]
                    ret = vec[0]
                elif e[0] == 'last':
                    #arguments take the form, (vec), return the
    last element
                    vec = args[0]
                    ret = vec[len(vec)-1]
                elif e[0] == 'append':
                    #arguments take the form, (vec,value)
                    vec = args[0]
                    ret = torch.cat((vec,torch.tensor([args[1]]))
    ,0)
                elif e[0] == 'hash-map':
                    myDict = {} #need to build the hash-map with
    non-tensor keys, want to make use of the arguments collected
    above
```

```
                    i = 1
                    while(i < len(e)):
                        myDict[e[i]] = args[i] # this is damn confusing, but e[1] corresponds to args[0], so e[1] is key 1 and args[1] is e[2] which is the value we want
                        i += 2
                    ret = myDict
                elif e[0] == 'mat-repmat':
                    ret = e[1].repeat(e[2],e[3])
                elif e[0] == 'mat-transpose':
                    ret = op(e[1])
                    print(ret)
                    input('make sure I get here')
                else:
                    ret = op(*args)
```

If this fails it is assumed to then be a user-defined function. If it is a function definition then the variable names are added to the local context populated with empty values to begin with, the function body is wrapped in a *lambda* expression and stored in the local variable context *var*. When the *lambda* expression is called, the bound input variables are populated in the local context *var* and the function body is evaluated.

If the function is defined we apply the function with the input variables.

```
            except:   #if it does not match to a primitive function, then it must be a function definition
                if e[0] == 'defn':
                    #function definition format (defn func-name [func-args] [func-body] )
                    ret = None
                    for i in range(0,len(e[2])):
                        var[e[2][i]] = [] #place holder for the name of the arguments of the function
                        # this lambda expression binds variables to the constants in x and then runs the procedure
                    var[e[1]] = lambda x: [[var.update({e[2][i]:x[i]}) for i in range(0,len(x))], evaluate_program(e[3])] # the code for the function body

                else:
                    try:
                        # does the function already exists?
                        [_,(ret,s)] = var[e[0]](args)
                    # if not evaluate the program
                    except:
                        # this fall through is annoying and I would like it to be avoided really, not sure what to do
                        ret = []
```

If none of these cases match, as is the case when the list of expressions is the variable names of the function, the empty list is returned. (Not exactly happy with this)

Finally, if nothing matches to the above, the expression is assumed to be a variable and if it has been defined fetched from the local context *var*, otherwise the variable name is added with an empty value to the context.

```
            try:
                #if the variable has a binding return the binding
                ret = var[e]
            except:
                var[e] = []
                ret = var[e]
```

1.3 Results

All tests given pass - debugging on the more complicated programs was my take instead of writing more test cases. This likely means there some bugs that I have missed.

```
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.8501984267746799
('beta', 2.0, 5.0)
p value 0.003827349256147307
('exponential', 0.0, 5.0)
p value 0.6721877980957955
('normal', 5.3, 3.2)
p value 0.4106253276911206
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.5774036344195437
('normal', 0, 1.44)
p value 0.6460206016716383
All probabilistic tests passed
```

Running programs 1 through 4 the following histogram plots are generated after taking 1000 samples of these programs, the titles and descriptions describe what the histogram is representing. The title includes the mean and standard deviation. Only the HMM model does not include this, but there the title includes the time steps taken.

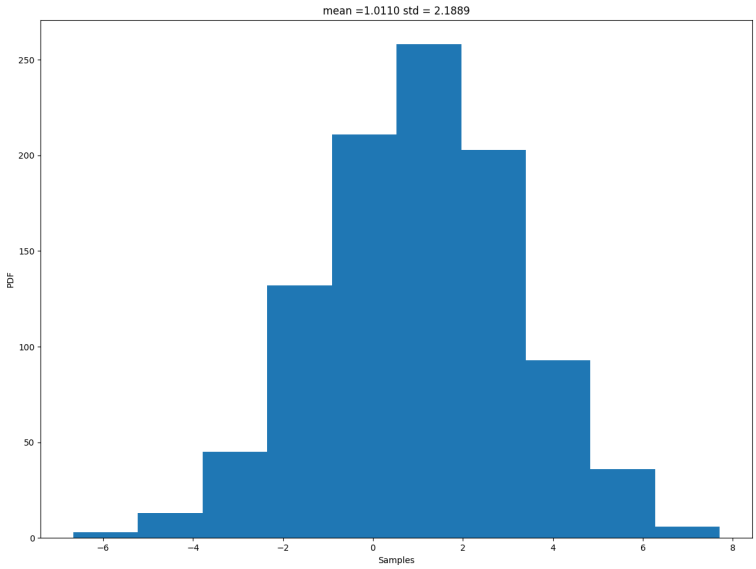

Figure 1: Program 1: The Gaussian unit test with mean: 1 std: $\sqrt{5}$

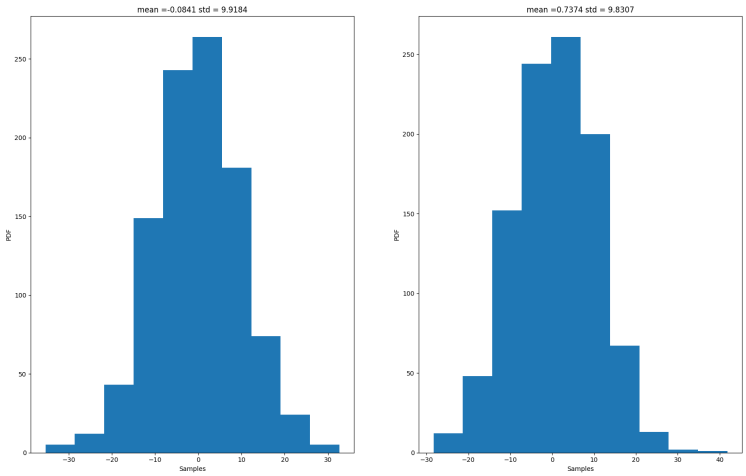

Figure 2: Program 2: The Bayesian linear regression with slope and bias mean:0 std: 10. The plot on the left is the slope and the one on the right is the bias

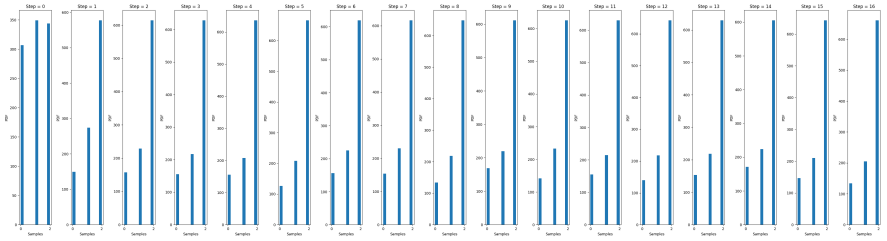

Figure 3: Program 3: The HMM distribution of states is shown at each time step

The histograms from time step 3 onwards the distribution of states appears to have converged. In step 0 as denoted in the program the distribution of states is roughly equal.

The Bayesian Neural Network returns the distribution for W_0, b_0, W_1 and b_1.

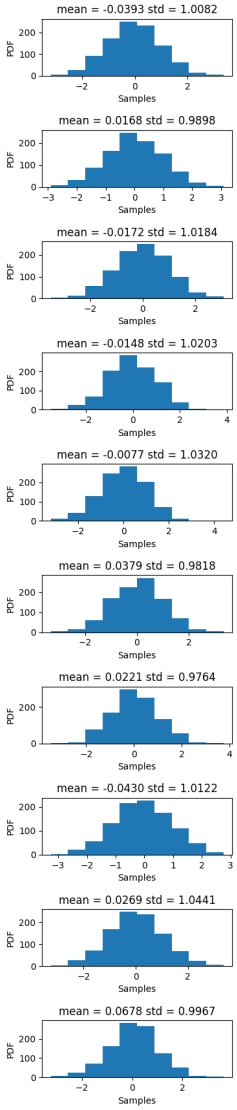

Figure 4: Program 4: The Bayesian Neural Network weights W_0 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of W_0 (zoom in to see mean and std)

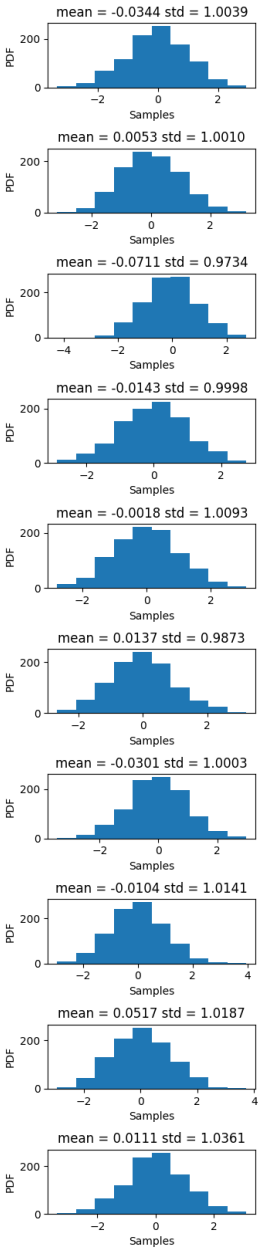

Figure 5: Program 4: The Bayesian Neural Network weights b_0 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of b_0 (zoom in to see mean and std)

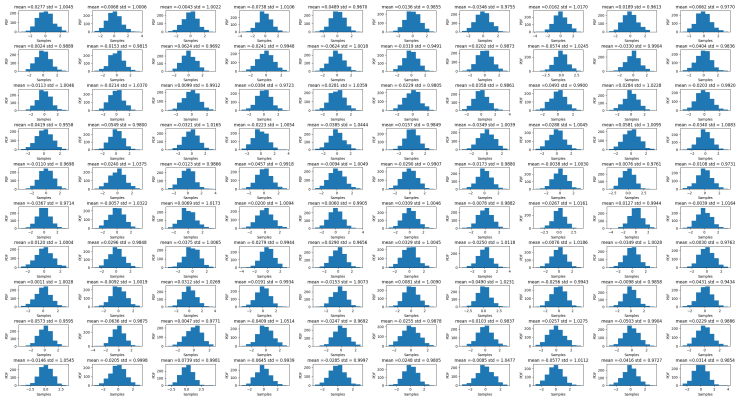

Figure 6: Program 4: The Bayesian Neural Network weights W_1. The histograms are oriented in the same configuration as the W_1 matrix i.e histogram i, j corresponds to element i, j of W_1 where W_1 is a 10×10 matrix. Here i corresponds to rows, j to columns.(Zoom in to see the mean and std)

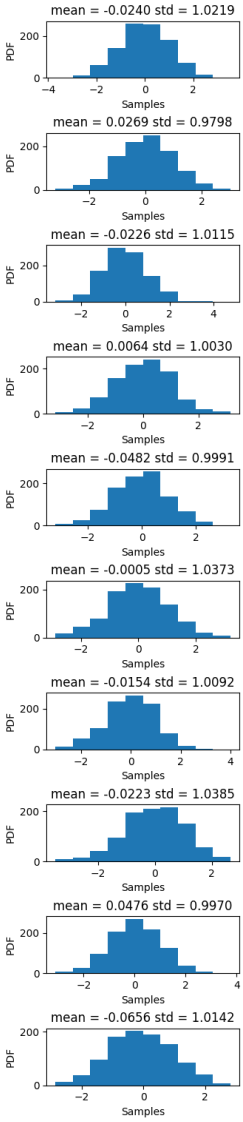

Figure 7: Program 4: The Bayesian Neural Network weights b_1 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of b_1 (zoom in to see mean and std)

2 Graph Based Sampling

The strategy in graph based sampling is a non-standard execution to achieve the same meaning as the evaluation based sampling. Similar to the previous part, Pytorch and tensors are used. The primitive functions implemented is the same:

```
env = { 'sqrt': torch.sqrt,
        '+': torch.add,
        '-': torch.sub,
        '/': torch.div,
        '*': torch.mul,
        'exp': torch.exp,
        '>': torch.greater,
        '<': torch.less,
        'normal': dist.Normal,
        'uniform': dist.Uniform,
        'exponential': dist.Exponential,
        'beta': dist.Beta,
        'discrete':dist.Categorical,
        'vector': torch.stack,
        'mat-transpose': torch.t,
        'mat-add': torch.add,
        'mat-mul': torch.matmul,
        'mat-repmat': None,
        'mat-tanh': torch.tanh,
        'get': None,
        'put': None,
        'first': None,
        'last': None,
        'append': None,
        'hash-map': None,
        'sample*': None,
        'observe*': None,
        'if': None}
```

The deterministic function evaluations makes calls to the primitive functions in:

```
from primitives import funcprimitives,bindingVars,topologicalSort
```

The deterministic evaluation is as follows:

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

And the *funcprimitives* are defined as:

```
def funcprimitives(e,op,args):
  if e == 'vector':
    try:
      ret = op(args)
    except:
      ret = args
  elif e == 'sample*':
    Dobj = args[0]
    ret = Dobj.sample()
  elif e == 'observe*':
    #for now we treat this as sample and ignore observed values
    Dobj = args[0]
    ret = Dobj.sample()
  elif e == 'if':
    if args[0]:
      ret = args[1]
    else:
      ret = args[2]
  elif e == 'get':
    ind = int(args[1])
    ret = args[0][ind]
  elif e == 'mat-transpose':
    ret = op(args[0])
  elif e == 'mat-repmat':
    ret = args[0].repeat(int(args[1]),int(args[2]))
  elif e == 'mat-mul':
    ret = op(*args)
  else:
    ret = op(*args)

  return ret
```

In order to construct the vectors and matrices properly, *torch.stack* is used to appropriately construct list of lists which allows us to define vectors and matrices. Similarly to the evaluation based sampling, *sample* and *observe* are treated the same. The control flow *if* is evaluated lazily here, but I acknowledge that when observes come into play this will need to be modified.

In order to do ancestral sampling, a topological sort of the vertices needs to be performed. I implemented Kahn's algorithm from:

`https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/`

Note a more concise implementation likely exists:

```
  #implementation of Khan's algorithm from:
  #https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/
  indegreeEdges = {}
  sortedNodes = []
  queue = []
  nodesVisited = 0
  #create dictionary of all the nodes and the number of ingoing edges
  for n in listOfVertices:
    indegreeEdges[n] = 0
```

```
  # go through all the nodes that have incoming edges and increment the
  # dictionary accordingly
  for n in listOfEdges:
    edges = listOfEdges[n]
    for e in edges:
      indegreeEdges[e] += 1

  # Find all nodes that have no incoming edges and put them in a queue
  for n in indegreeEdges:
    if indegreeEdges[n] == 0:
      queue.append(n)

  # if the queue starts empty, it means that there are no vertices that do not
  # have an incoming edge, in this case return the list of nodes
  if queue == []:
    return listOfVertices

  # while the queue is not empty, take a node off the queue (it has no incoming edges if it's on the queue)
  # add the node to the list of sorted nodes, increment the number of nodes visited,
  # visit the child nodes given from listOfEdges (if there are any)
  # when visiting a node decrement the indegree by 1 and check if it is 0, if so add it to the queue
  while(queue != []):
    n = queue.pop()
    nodesVisited += 1
    #if node n has neighbours get them, otherwise there are no neighbours to visit
    if n in listOfEdges.keys():
      neighbours = listOfEdges[n]
    else:
      neighbours = []
    sortedNodes.append(n)
    for v in neighbours:
      indegreeEdges[v] -= 1
      if indegreeEdges[v] == 0:
        queue.append(v)

  #verify that we have visited every node, otherwise something has gone wrong
  assert(nodesVisited == len(listOfVertices))

  return sortedNodes
```

Of note, if the sorting cannot be performed, then I return the list of vertices. Once the vertices are sorted, then I bind vertex nodes with their samples from the link functions in P by binding variables to values in *bindingVars* which recursively unpacks lists and binds variables to values:

```
def bindingVars(varDict,funcCode):
  evalFuncCode = funcCode.copy()
  for i in range(0,len(funcCode)):
    if type(evalFuncCode[i]) == list:
```

```
      evalFuncCode[i] = bindingVars(varDict,evalFuncCode[i])
    elif evalFuncCode[i] in varDict.keys():
      evalFuncCode[i] = float(varDict[evalFuncCode[i]]) #hacky for
    now
  return evalFuncCode
```

Thus to perform ancestral sampling the steps are extract vertices V and edges A from the graph. Topological sort V, get the link functions from P and execute the link function from P and store the results in a dictionary R. Sample each V in the sorted order making calls to P. Once each node V has a value, then the *bindingVars* function is called to return the expression of the program stored in E. *sample_from_joint* is shown:

```
    edges = listOfEdges[n]
    for e in edges:
      indegreeEdges[e] += 1

  # Find all nodes that have no incoming edges and put them in a
    queue
  for n in indegreeEdges:
    if indegreeEdges[n] == 0:
      queue.append(n)

  # if the queue starts empty, it means that there are no vertices
    that do not
  # have an incoming edge, in this case return the list of nodes
  if queue == []:
    return listOfVertices

  # while the queue is not empty, take a node off the queue (it has
     no incoming edges if it's on the queue)
  # add the node to the list of sorted nodes, increment the number
    of nodes visited,
  # visit the child nodes given from listOfEdges (if there are any)
  # when visiting a node decrement the indegree by 1 and check if
    it is 0, if so add it to the queue
  while(queue != []):
    n = queue.pop()
    nodesVisited += 1
    #if node n has neighbours get them, otherwise there are no
    neighbours to visit
    if n in listOfEdges.keys():
      neighbours = listOfEdges[n]
    else:
      neighbours = []
```

2.1 Results

Similar to the evaluation based sampling, the results are summarized here:.

```
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.7468519478136475
('beta', 2.0, 5.0)
p value 0.9760889086114111
('exponential', 0.0, 5.0)
p value 0.7425461404031451
('normal', 5.3, 3.2)
p value 0.319890341320916
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.3386100157743366
('normal', 0, 1.44)
p value 0.35515239218733097
All probabilistic tests passed
```

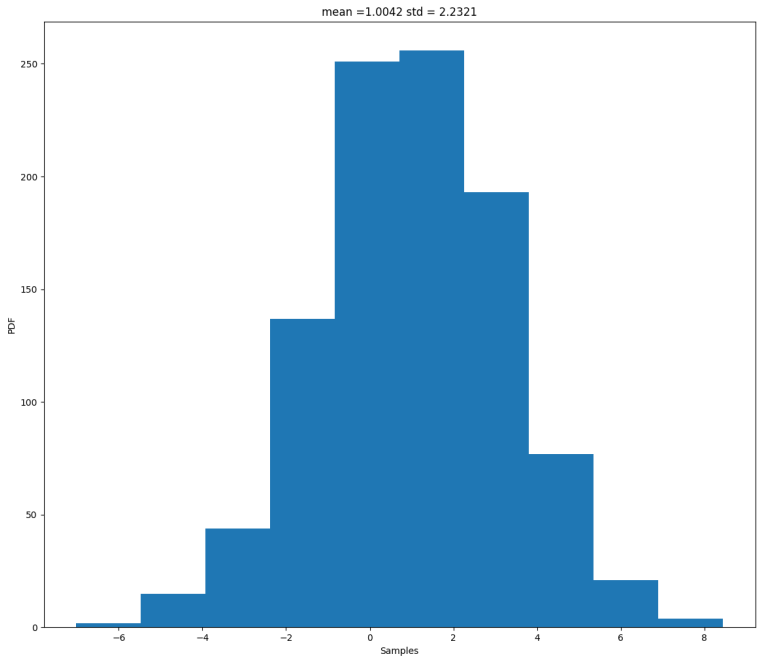

Figure 8: Program 1: The Gaussian unit test with mean: 1 std: $\sqrt{5}$

The histograms from the evaluation based and graph based sampling produce similar results, i.e. after time step 3 the distribution of states appears to have converged. In step 0 as denoted in the program the distribution of states is roughly equal.

The Bayesian Neural Network returns the distribution for W_0, b_0, W_1 and b_1.

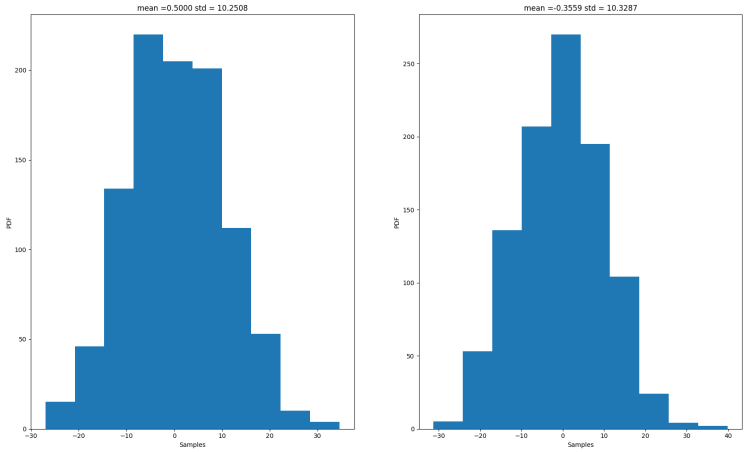

Figure 9: Program 2: The Bayesian linear regression with slope and bias mean:0 std: 10. The plot on the left is the slope and the one on the right is the bias

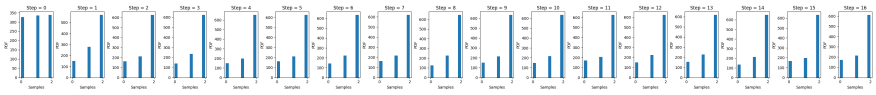

Figure 10: Program 3: The HMM distribution of states is shown at each time step

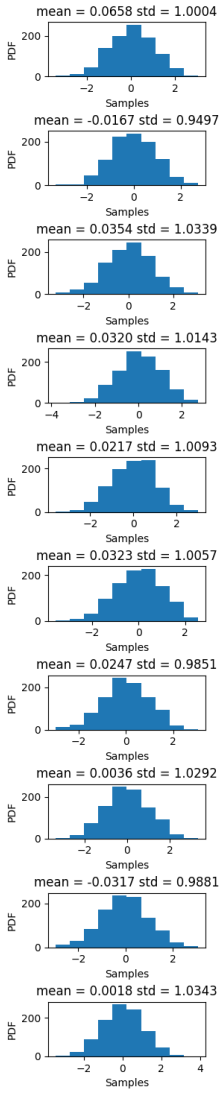

Figure 11: Program 4: The Bayesian Neural Network weights W_0 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of W_0 (zoom in to see mean and std)

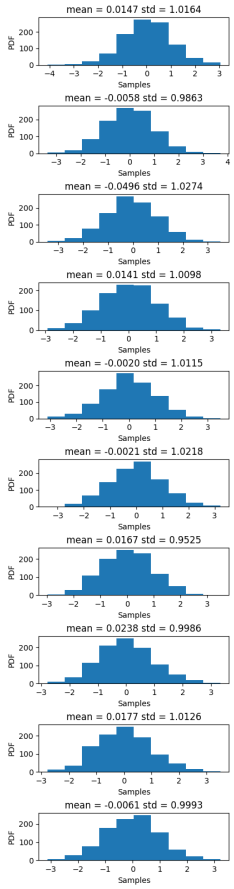

Figure 12: Program 4: The Bayesian Neural Network weights b_0 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of b_0 (zoom in to see mean and std)

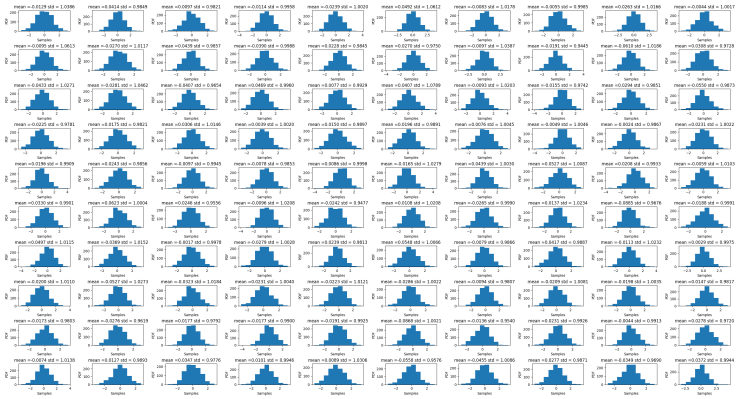

Figure 13: Program 4: The Bayesian Neural Network weights W_1. The histograms are oriented in the same configuration as the W_1 matrix i.e histogram i, j corresponds to element i, j of W_1 where W_1 is a 10×10 matrix. Here i corresponds to rows, j to columns.(Zoom in to see the mean and std)

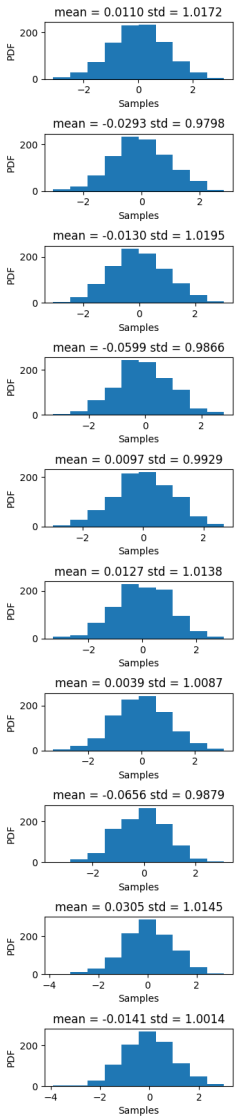

Figure 14: Program 4: The Bayesian Neural Network weights b_1 from top to bottom are the corresponding $i = 0$ to $i = 9$ histograms for the 10×1 vector of b_1 (zoom in to see mean and std)