CPSC532W Homework 5

Justin Reiher
Student ID: 37291151
CWL: reiher

Link to public repository for homework 5:

`https://github.com/justinreiher/probProg_Fall2021/tree/main/CS532-HW5`

The HOPPL is implemented following Peter Norvig's tutorial `https://norvig.com/lispy.html` very closely. Particularly the `Procedure` and `Environment` classes:

```
class Env(dict):
    "An environment: a dictionary of ('var':val) paris with and
    outer Env."
    def __init__(self,params=(),args=(),outer=None):
        self.update(zip(params,args))
        self.outer = outer
    def get(self,var):
        "Find the innermost Env where var appears."
        return self[var] if (var in self) else self.outer.get(var)

class Procedure(object):
    "A user-defined FOPPL procedure."
    def __init__(self,params,body,env):
        self.params, self.body, self.env = params,body,env
    def __call__(self,*args):
        return evaluate(self.body, Env(self.params, args, self.env)
    )
```

The evaluator itself likewise follows the format and style in the tutorial augmented with `sample` and `observe` where `observe` in this case does nothing interesting other than return the observed value:

```
def evaluate(exp, env=None): #TODO: add sigma, or something
    # if the environment is not set, then get the standard
    environment, and add
    # sigma to this environment
    if env is None:
        env = standard_env()
        env = env.update({'sig':''})

    if isinstance(exp,Symbol):  #variable reference
        e = env.get(exp)
        if e == None:
            e = exp
```

```
            return e
    elif not isinstance(exp,List): #constant case
            return torch.tensor(float(exp))

    op, *args = exp
    if op == 'if':
            (test,conseq,alt) = args
            exp = (conseq if evaluate(test,env) else alt)
            return evaluate(exp,env)
    elif op == 'fn': #procedure definition
            (params,body) = args
            return Procedure(params,body,env)
    elif op == 'sample':
            v = evaluate(args[0],env)
            d = evaluate(args[1],env)
            return d.sample()
    elif op == 'observe':
            v = evaluate(args[0],env)
            d = evaluate(args[1],env)
            c = evaluate(args[2],env)
            return c
    else:
            proc = evaluate(op,env)
            vals = [evaluate(arg,env) for arg in args]
            return proc(*vals)

    return
```

The primitives in the HOPPL are implemented with `lambda` expressions where possible (again following Peter Norvig's tutorial example). The environment primitive implemented are:

```
env = { 'sqrt': lambda _,a: torch.sqrt(a),
        '+': lambda _,a,b: torch.add(a,b),
        '-': lambda _,a,b: torch.sub(a,b),
        '/': lambda _,a,b: torch.div(a,b),
        '*': lambda _,a,b: torch.mul(a,b),
        'exp': lambda _,a: torch.exp(a),
        'abs': lambda _,a: torch.abs(a),
        'log': lambda _,a: torch.log(a),
        '>': lambda _,a,b: torch.greater(a,b),
        '<': lambda _,a,b: torch.less(a,b),
        '=': lambda _,a,b: torch.equal(a,b),
        'or': lambda _,a,b: a or b,
        'and': lambda _,a,b: a and b,
        'empty?': lambda _,a: len(a)==0,
        'true': torch.tensor(1.0),
        'false': torch.tensor(0.0),
        'normal': Normal,
        'uniform': Uniform,
        'uniform-continuous': Uniform,
        'exponential': lambda _,x: dist.Exponential(x),
        'beta': lambda _,a,b: dist.Beta(a,b),
        'discrete': Categorical,
        'dirichlet': Dirichlet,
        'flip': Bernoulli,
        'gamma': Gamma,
        'dirac': None,
```

```
        'vector': Vector,
        'mat-transpose': lambda _, M: M.t(),
        'mat-add': lambda _,a,b: torch.add(a,b),
        'mat-mul': lambda _,a,b: torch.matmul(a,b),
        'mat-repmat': lambda _,M,a,b: M.repeat(int(a),int(b)),
        'mat-tanh': lambda _,a: torch.tanh(a),
        'get': Get,
        'put': Put,
        'first': First,
        'second': Second,
        'rest': Rest,
        'last': Last,
        'append': Append,
        'hash-map': HashMap,
        'cons': Cons,
        'conj': Conj,
        'peek': lambda _,d: d[-1],
        'push-address': push_addr}
```

The lambda expressions all have as a first parameter the address which at this stage does nothing. Functions that required particular implementations are as shown:

```
def push_addr(alpha, value):
    return alpha + value

def Vector(*args):
    addr = args[0]
    if len(args[1:]) == 0:
        return []
    return torch.stack([i for i in args[1:]])

def Get(*args):

    addr,d,ind = args
    addr = args[0]
    d = args[1]
    ind = args[2]
    if type(ind) == type(torch.tensor(1.)):
        ind = int(ind)
    return d[ind]

def Put(*args):
    addr,d,ind,val = args
    if type(d) == dict:
        dRet = d.copy()
    else:
        dRet = d.clone()
    if type(ind) == type(torch.tensor(1.)):
        ind = int(ind)
    dRet[ind] = val
    return dRet

def First(*args):
    addr = args[0]
    vec = args[1]
    return vec[0]

```

```python
def Second(*args):
    addr = args[0]
    vec = args[1]
    return vec[1]

def Last(*args):
    addr = args[0]
    vec = args[1]
    return vec[len(vec)-1]

def Rest(*args):
    addr = args[0]
    vec = args[1]
    return vec[1:]

def Append(*args):
    addr = args[0]
    vec = args[1]
    val = args[2]
    return torch.cat((vec,torch.tensor([val])),0)

def dirac(*args):
    addr = args[0]
    sigma = 0.1
    return Normal(d,sigma)

def HashMap(*args):
    addr = args[0]
    kvp = args[1:]
    retD = {}
    i = 0
    while(i < len(kvp)-1):
        if type(kvp[i]) == type(torch.tensor(1.)):
            retD[int(kvp[i])] = kvp[i+1]
        else:
            retD[kvp[i]] =  kvp[i+1]
        i += 2
    return retD

def Cons(*args):
    addr,l,val = args
    return torch.cat((torch.tensor([val]),torch.tensor(l)),0)

def Conj(*args):
    addr,l1,val = args
    return torch.cat((l1,torch.tensor([val])),0)
```

Any changes to data structures are done on copies to not modify any original data. `Pyrsistent` provides that functional but is not extensively used here. The distribution objects that were provided in homework 4 are used and augmented with the address variable where implemented:

```python
class Normal(dist.Normal):

    def __init__(self, alpha, loc, scale):

        if scale > 20.:
```

```python
            self.optim_scale = scale.clone().detach().requires_grad_()
        else:
            self.optim_scale = torch.log(torch.exp(scale) - 1).clone().detach().requires_grad_()


        super().__init__(loc, torch.nn.functional.softplus(self.optim_scale))

    def Parameters(self):
        """Return a list of parameters for the distribution"""
        return [self.loc, self.optim_scale]

    def make_copy_with_grads(self):
        """
        Return a copy  of the distribution, with parameters that require_grad
        """

        ps = [p.clone().detach().requires_grad_() for p in self.Parameters()]

        return Normal(*ps)

    def log_prob(self, x):

        self.scale = torch.nn.functional.softplus(self.optim_scale)

        return super().log_prob(x)

class Bernoulli(dist.Bernoulli):

    def __init__(self,alpha, probs=None, logits=None):
        if logits is None and probs is None:
            raise ValueError('set probs or logits')
        elif logits is None:
            if type(probs) is float:
                probs = torch.tensor(probs)
            logits = torch.log(probs/(1-probs)) ##will fail if probs = 0
        #
        super().__init__(logits = logits)

    def Parameters(self):
        """Return a list of parameters for the distribution"""
        return [self.logits]

    def make_copy_with_grads(self):
        """
        Return a copy  of the distribution, with parameters that require_grad
        """
        logits = [p.clone().detach().requires_grad_() for p in self.Parameters()][0]

        return Bernoulli(logits = logits)
```

```
class Categorical(dist.Categorical):

    def __init__(self,alpha, probs=None, logits=None, validate_args=None):

        if (probs is None) == (logits is None):
            raise ValueError("Either `probs` or `logits` must be specified, but not both.")
        if probs is not None:
            if probs.dim() < 1:
                raise ValueError("`probs` parameter must be at least one-dimensional.")
            probs = probs / probs.sum(-1, keepdim=True)
            logits = dist.utils.probs_to_logits(probs)
        else:
            if logits.dim() < 1:
                raise ValueError("`logits` parameter must be at least one-dimensional.")
            # Normalize
            logits = logits - logits.logsumexp(dim=-1, keepdim=True)
        super().__init__(logits = logits)
        self.logits = logits.clone().detach().requires_grad_()
        self._param = self.logits

    def Parameters(self):
        """Return a list of parameters for the distribution"""
        return [self.logits]

    def make_copy_with_grads(self):
        """
        Return a copy  of the distribution, with parameters that require_grad
        """
        logits = [p.clone().detach().requires_grad_() for p in self.Parameters()][0]

        return Categorical(logits = logits)

class Dirichlet(dist.Dirichlet):

    def __init__(self, alpha,concentration):
        #NOTE: logits automatically get added
        super().__init__(concentration)

    def Parameters(self):
        """Return a list of parameters for the distribution"""
        return [self.concentration]

    def make_copy_with_grads(self):
        """
        Return a copy  of the distribution, with parameters that require_grad
        """

        concentration = [p.clone().detach().requires_grad_() for p
```

```
    in self.Parameters()][0]

        return Dirichlet(concentration)

class Gamma(dist.Gamma):

    def __init__(self, alpha,concentration, rate, copy=False):
        if rate > 20. or copy:
            self.optim_rate = rate.clone().detach().requires_grad_
    ()
        else:
            self.optim_rate = torch.log(torch.exp(rate) - 1).clone
    ().detach().requires_grad_()


        super().__init__(concentration, torch.nn.functional.
    softplus(self.optim_rate))

    def Parameters(self):
        """Return a list of parameters for the distribution"""
        return [self.concentration, self.optim_rate]

    def make_copy_with_grads(self):
        """
        Return a copy  of the distribution, with parameters that
    require_grad
        """

        concentration,rate = [p.clone().detach().requires_grad_()
    for p in self.Parameters()]
        return Gamma(concentration, rate, copy = True)

    def log_prob(self, x):

        self.rate = torch.nn.functional.softplus(self.optim_rate)

        return super().log_prob(x)

class Uniform:

    def __init__(self,alpha,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)

```

```python
    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))))
```

1 Program 1: Deterministic and Probabilistic Tests

Output demonstrating all tests pass:

```
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
FOPPL Tests passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
Test passed
/home/justin/Research/Research/ProbProg/CS532-HW5/primitives.py:244: UserWarning: To copy co
return torch.cat((torch.tensor([val]),torch.tensor(l)),0)
Test passed
All deterministic tests passed
('normal', 5, 1.4142136)
p value 0.4392251209556768
('beta', 2.0, 5.0)
p value 0.20362142284502927
```

```
('exponential', 0.0, 5.0)
p value 0.4026319736237799
('normal', 5.3, 3.2)
p value 0.6117760761512494
('normalmix', 0.1, -1, 0.3, 0.9, 1, 0.3)
p value 0.42402962493527685
('normal', 0, 1.44)
p value 0.018257659736088516
All probabilistic tests passed
```

The warning in the HOPPL test 12:

```
/home/justin/Research/Research/ProbProg/CS532-HW5/primitives.py:244: UserWarning: To copy co
return torch.cat((torch.tensor([val]),torch.tensor(l)),0)
```

is telling me that I should not call `torch.tensor(l)` on a tensor object that already exists. However the behaviour is correct, which is to say that if the list is not a `torch.tensor(l)` it will create one, if it already exists then it returns the same list.

2 Running Programs

All programs are run with 10k samples and the results are shown below

2.1 Program 1

Output from running program 1:

```
Sample of prior of program 1:
Elapsed time for program  1 .daphne is:  0:05:11.798638  seconds
Mean of samples:  tensor(99.1019)
Variance of samples:  tensor(9923.6123)
```

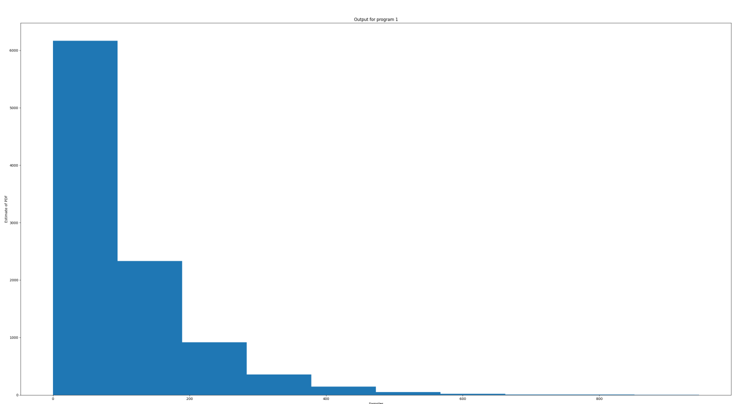

Figure 1: Histogram for Program 1

2.2 Program 2

Output from running program 2:

```
Sample of prior of program 2:
Elapsed time for program  2 .daphne is:  0:00:29.480438  seconds
Mean of samples:  tensor(0.9736)
Variance of samples:  tensor(5.0724)
```

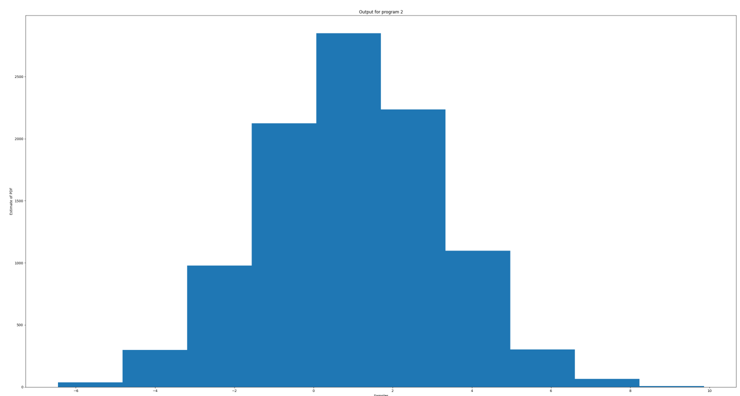

Figure 2: Histogram for Program 2

2.3 Program 3

Output from running program 3:

```
Sample of prior of program 3:
Elapsed time for program  3 .daphne is:  0:02:03.474374  seconds
```

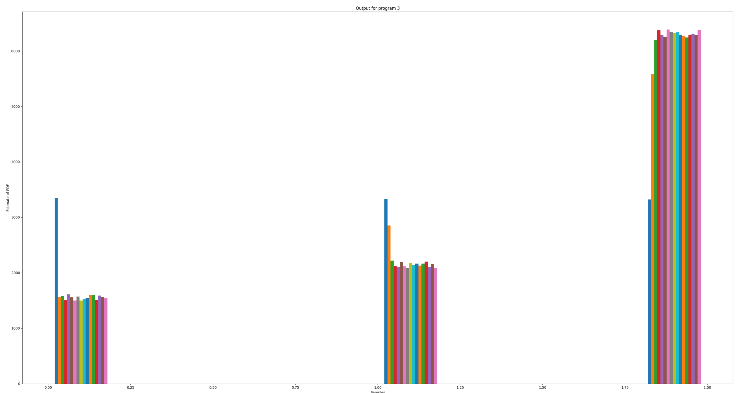

Figure 3: Histogram for Program 3