```
-- Only requirements allowed
require("hdf5")
require("optim")
require("nn")
require("nnx")
require("cunn")
require("cutorch")
-- require("cudnn")

cmd = torch.CmdLine()

UNSEEN = -1

-- Cmd Args
cmd:option('-datafile', '', 'data file')
cmd:option('-lm', 'nn', 'classifier to use')
cmd:option('-alpha', 0.01, 'Laplace smoothing coefficient')
cmd:option('-eta', 0.05, 'learning rate')
cmd:option('-nepochs', 3, 'number of training epochs')
cmd:option('-mb', 32, 'minibatch size')
cmd:option('-k', 1, 'ratio of noise for NCE')
cmd:option('-gpu', 0, 'whether to use gpu for training')

-- Helper function which converts a tensor to a key for table lookup
function tensor_to_key(t)
  local key = ''
  local table = torch.totable(t)
  for k, v in pairs(table) do
    key = key .. tostring(v) .. ','
  end
  return string.sub(key, 1, -2) -- Remove trailing comma
end

function count_based(smoothing)
  -- Specializd function for building unigram probability distribution (no context)
  function build_unigram()
    local unigrams = {}
    for i = 1, train_y:size(1) do
      local wi = train_y[i]
      if unigrams[wi] == nil then
        unigrams[wi] = 1
      else
        unigrams[wi] = unigrams[wi] + 1
      end
    end

    for wi, count in pairs(unigrams) do
```

```
    unigrams[wi] = count / train_y:size(1)
  end

  return unigrams
end

print('Building unigram distribution...')
local unigram = build_unigram()

-- Normalized probability distribution of unigrams
function p_unigram(wi)
  p = unigram[wi]
  if p == nil then -- Never before seen unigram, go with uniform dist
    p = (1.0 / nwords)
  end
  return p
end

-- Raw frequency of unigrams in dataset
function freq_unigram(wi)
  return p_unigram(wi) * train_y:size(1)
end

-- Create co-occurrence dictionary mapping contexts to following words
function build_ngram(x, y)
  print('Building p(w_ilw_i-n+1,...,w_i-1)...')
  local ngram = {}
  for i = 1, x:size(1) do
    local ctx = tensor_to_key(x[i])
    local wi = y[i]
    local val = ngram[ctx]
    if val == nil then
      ngram[ctx] = {}
      ngram[ctx][wi] = 1
    else
      local innerval = ngram[ctx][wi]
      if innerval == nil then
        ngram[ctx][wi] = 1
      else
        ngram[ctx][wi] = ngram[ctx][wi] + 1
      end
    end

    if i % 100000 == 0 then
      print('Processed ' .. i .. ' training examples.')
    end
  end
```

```
  return ngram
end

-- Renormalize probabilities for each ngram
function normalize_ngram(ngram)
  print('Renormalizing ngram probabilities...')
  for ctx, wis in pairs(ngram) do
    local sum = 0
    local seen = 0 -- To calculate the number of unseen wi in given ctx
    for wi, tot in pairs(wis) do
      sum = sum + tot
      seen = seen + 1
    end

    local unseen = nwords - seen
    -- Smoothing + normalization
    if smoothing == 'mle' then
      -- How to predict unseen without smoothing for mle??
      sum = sum + (unseen * (1.0 / nwords)) -- Uniform assumption
      for wi, tot in pairs(wis) do
        ngram[ctx][wi] = ngram[ctx][wi] / sum
      end
      ngram[ctx][UNSEEN] = (1.0 / nwords) / sum
    elseif smoothing == 'lap' then
      sum = sum + (nwords * alpha) -- Add probability mass for every wi (including
unseen)
      for wi, tot in pairs(wis) do
        ngram[ctx][wi] = (ngram[ctx][wi] + alpha) / sum
      end
      ngram[ctx][UNSEEN] = alpha / sum
    end
  end
end

function p_ngram(ngram, ctx, wi)
  local ctxd = ngram[ctx]
  local p = 0
  if ctxd == nil then -- Never before seen context, go with unigram
    p = p_unigram(wi)
    -- p = (1 / nwords)
  else
    p = ngram[ctx][wi]
    if p == nil then -- We've seen this context before, just not this wi
      p = ngram[ctx][UNSEEN]
    end
  end
  return p
```

```
end

-- Count total occurrences of context and unique types within context
function lambda_for_ctx(ngram, ctx, word)
  local unique = 0
  local total = 0
  local ctx_wi_freq = 0
  if ngram[ctx] == nil then
    return 0, 0, 0, 0 -- By defn seems like should be 1, but intuitively should be 0
  else
    for wi, tot in pairs(ngram[ctx]) do
      unique = unique + 1
      total = total + tot
      if wi == word then ctx_wi_freq = tot end
    end

    local lambda = 1 - (unique / (unique + total))
    return lambda, unique, total, ctx_wi_freq
  end
end

-- Calculates interpolated probability based on occurrences of the specific
-- context/word pair, unique words in context, total words in context, and
-- the probability of the 1-order lower context
function p_wb(ctx_wi_freq, unique, total, p_lower)
  if unique + total == 0 then -- Prevent / 0 in denom
    return 0
  else
    local num = ctx_wi_freq + unique * p_lower
    local denom = unique + total
    return num / denom
  end
end

function interpolated_p_wb(trigram, bigram, x, wi)
  if trigram ~= nil then -- Trigram, bigram, unigram interpolation
    local ctx3 = tensor_to_key(x)
    local ctx2 = tensor_to_key(torch.Tensor({x[1]}))

    local l3, unq3, tot3, ctx_wi_freq3 = lambda_for_ctx(trigram, ctx3, wi)
    local l2, unq2, tot2, ctx_wi_freq2 = lambda_for_ctx(bigram, ctx2, wi)
    local l1 = nwords / (nwords + freq_unigram(wi))
    local all = l3 + l2 + l1 -- Ensure lambdas form complex combination
    l3 = l3 / all
    l2 = l2 / all
    l1 = l1 / all
```

```
    p1 = p_unigram(wi)
    p2 = p_wb(ctx_wi_freq2, unq2, tot2, p1)
    p3 = p_wb(ctx_wi_freq3, unq3, tot3, p2)

    return (l3 * p3) + (l2 * p2) + (l1 * p1)
  else -- Bigram, unigram interpolation
    local ctx = tensor_to_key(x)

    local l2, unq2, tot2, ctx_wi_freq2 = lambda_for_ctx(bigram, ctx, wi)
    local l1 = nwords / (nwords + freq_unigram(wi))
    local all = l2 + l1 -- Ensure lambdas form complex combination
    l2 = l2 / all
    l1 = l1 / all

    p1 = p_unigram(wi)
    p2 = p_wb(ctx_wi_freq2, unq2, tot2, p1)

    return (l2 * p2) + (l1 * p1)
  end
end

-- Perplexity = exponential(avg negative conditional-log-likelihood)
function perplexity(ngram, x, y)
  local sum = 0
  for i = 1, x:size(1) do
    local ctx = tensor_to_key(x[i])
    local wi = y[i]
    local p = p_ngram(ngram, ctx, wi)

    sum = sum + math.log(p)
  end

  local nll = (-1.0 / x:size(1)) * sum
  return math.exp(nll)
end

-- Perplexity = exponential(avg negative conditional-log-likelihood,
-- interpolated alongside all lower order ngrams)
function wb_perplexity(trigram, bigram, x, y)
  local sum = 0
  for i = 1, x:size(1) do
    p_interp = interpolated_p_wb(trigram, bigram, x[i], y[i])
    sum = sum + math.log(p_interp)
  end

  local nll = (-1.0 / x:size(1)) * sum
  return math.exp(nll)
```

```
  end

  if smoothing ~= 'wb' then
    print('Building count based model (ngram=' .. ngram .. ', smoothing=' .. smoothing ..
')...')

    local ngram = build_ngram(train_x, train_y)
    normalize_ngram(ngram)

    print('Calculating perplexity on train...')
    local perp = perplexity(ngram, train_x, train_y)
    print('Training perplexity: ' .. perp)

    print('Calculating perplexity on valid...')
    perp = perplexity(ngram, valid_x, valid_y)
    print('Validation perplexity: ' .. perp)
  else
    print('Building count based model with Witten-Bell smoothing (max ngram=' ..
ngram .. ')...')

    local bigram = build_ngram(bigram_train_x, bigram_train_y)
    -- normalize_ngram(bigram)

    local trigram = nil
    if interp == 3 then
      trigram = build_ngram(trigram_train_x, trigram_train_y)
      -- normalize_ngram(trigram)
    end

    print('Calculating perplexity on train...')
    local perp = wb_perplexity(trigram, bigram, train_x, train_y)
    print('Training perplexity: ' .. perp)

    print('Calculating preplexity on valid...')
    perp = wb_perplexity(trigram, bigram, valid_x, valid_y)
    print('Validation perplexity: ' .. perp)
  end
end

function nnlm(structure)
  local embedding_size = 50
  local din = embedding_size * dwin
  local dhid = 100
  local dout = nwords

  print('\nBuilding neural language model with hyperparameters:')
  print('Learning rate (eta): ' .. eta)
```

```
print('Number of epochs (nepochs): ' .. n_epochs)
print('Mini-batch size (mb): ' .. batch_size)
print('Context window (dwin): ' .. dwin)
print('Embedding size (embed): ' .. embedding_size)
print('Vocabulary size (|V|): ' .. nwords)

-- For hierarchical softmax
function build_softmax_tree()
  hierarchy = {}
  -- n_buckets = math.ceil(math.sqrt(nwords))

  -- Randomly sort words into two layer tree
  -- for i = 1, n_buckets do
  --   local words = torch.range((i - 1) * n_buckets + 1, i * n_buckets):int()
  --   hierarchy[i] = words
  -- end

  n_buckets = math.ceil(math.sqrt(nwords))
  root = n_buckets + 1
  hierarchy[root] = torch.IntTensor{nwords + 1, 1, 2}
  for i = 1, n_buckets do
    hierarchy[i] = torch.range((i - 1) * n_buckets + 3, i * n_buckets + 2):int()
  end

  return hierarchy
end

function tree_for_word(w)
  return math.ceil(w / n_buckets)
end

local model = nn.Sequential()

if structure == 'nn' then
  -- Lookup table concats embeddings for words in context
  input_embedding = nn.LookupTable(nwords, embedding_size)
  model:add(input_embedding)
  model:add(nn.View(din))

  -- Linear, tanh, linear
  model:add(nn.Linear(din, dhid))
  model:add(nn.Tanh())
  model:add(nn.Linear(dhid, dout))

  model:add(nn.LogSoftMax())
  nll = nn.ClassNLLCriterion()
  print('Using softmax output.')
```

```
  if gpu == 1 then
    model:cuda()
    nll = nll:cuda()
    print('Using gpu accelerated training.')
  end
elseif structure == 'hsm' then
  build_softmax_tree()

  -- Parallel table to forward target directly to tree softmax module
  local para = nn.ParallelTable()
  local inp = nn.Sequential()
  input_embedding = nn.LookupTable(nwords, embedding_size)
  inp:add(input_embedding)
  inp:add(nn.View(din))
  inp:add(nn.Linear(din, dhid))
  inp:add(nn.HardTanh())
  local out = nn.Identity()

  para:add(inp)
  para:add(out)
  model:add(para)

  model:add(nn.SoftMaxTree(dhid, hierarchy, root, false, true, true))
  nll = nn.TreeNLLCriterion()
  print('Using hierarchical softmax output.')
elseif structure == 'nce' then
  -- NB: currently incomplete
  local w1 = nn.Sequential()

  w1:add(nn.LookupTable(nwords, embedding_size))
  w1:add(nn.Reshape(din))

  -- Linear (no bias), tanh
  local lin1 = nn.Linear(din, dhid)
  lin1.bias = false
  w1:add(lin1)
  w1:add(nn.Tanh())

  local w2 = nn.Sequential()
  w2:add(nn.LookupTable(nwords, embedding_size))

  -- local lin2 = nn.Linear(dhid, dout)
  -- lin2.bias = false
  -- model:add(lin2)

  print('Using noise contrastive estimation.')
end
```

```
  params, gradParams = model:getParameters()
  if gpu == 1 then
    params = params:cuda()
    gradParams = gradParams:cuda()
  end

  function train(e)
    -- Package selected dataset into minibatches
    local n_train_batches = math.floor(train_x:size(1) / batch_size) - 1

    order = torch.randperm(train_x:size(1))
    -- subsets = torch.range(1,word_order:size(1)-subset_size, subset_size)

    print('\nBeginning epoch ' .. e .. ' training: ' .. n_train_batches .. ' minibatches of size
' .. batch_size .. '.')
    for i = 1, n_train_batches do
      local batch_start = torch.random(i * batch_size + 1, (i + 1) * batch_size)
      local batch_end = math.min((batch_start + batch_size - 1), order:size(1))

      curr_batch = order[{{batch_start, batch_end}}]:long()
      local x = train_x:index(1, curr_batch)
      local y = train_y:index(1, curr_batch)

      if gpu == 1 then
        x = x:cuda()
        y = y:cuda()
      end

      -- Compute forward and backward pass (predictions + gradient updates)
      function run_minibatch(p)
        if params ~= p then
          params:copy(p)
        end

        -- Accumulate gradients from scratch each minibatch
        gradParams:zero()
        local loss = 0

        if structure == 'nn' then
          -- Forward pass
          local preds = model:forward(x)
          loss = nll:forward(preds, y)

          -- Backward pass
          local dLdpreds = nll:backward(preds, y)
          model:backward(preds, dLdpreds)
```

```
      elseif structure == 'hsm' then
        -- Forward pass (requires input + target)
        local preds = model:forward({x, y})
        loss = nll:forward(preds, y)

        -- Backward pass
        local dLdpreds = nll:backward(preds, y)
        -- model:backward(preds, dLdpreds)
        model:backward(x, dLdpreds)
      end

      if i % 2000 == 0 then
        print('Completed ' .. i .. ' minibatches.')
        -- print('Loss after ' .. batch_end .. ' examples: ' .. loss)
      end

      return loss, gradParams
    end

    options = {
      learningRate = eta,
      learningRateDecay = 0.0001
      -- alpha = 0.95 -- For rmsprop
      -- momentum = 0.5
    }

    -- Use optim package for minibatch sgd
    collectgarbage()
    optim.sgd(run_minibatch, params, options)
    -- optim.adagrad(run_minibatch, params, options)
    -- optim.rmsprop(run_minibatch, params, options) -- Slower
  end

  -- Renormalize input embeddings after each epoch (max l2 norm = 1)
  -- local threshold = 1
  -- input_embedding.weight:renorm(2, 2, threshold):cuda()
end

function test(x, y)
  local preds = model:forward(x)
  local loss = nll:forward(preds, y)
  local perp = math.exp(loss)
  if structure == 'nn' then
    local max, yhat = preds:max(2)
    local correct = yhat:int():eq(y):double():mean() * 100
    return correct, loss, perp
  else
```

```
    return 0, 0, perp
  end
end

function hsm_perp(x, y)
  -- Each pred is the likelihood of a leaf class (y)
  -- To generate distribution need to provide all desired
  -- classes and renormalize
  local preds = model:forward({x, y})
  local loss = nll:forward(preds, y)
  local perp = math.exp(loss)

  return perp
end

function valid_acc()
  if structure == 'hsm' then
    -- return test({valid_x, valid_y}, valid_y)
    return 0, 0, hsm_perp(valid_x, valid_y)
  elseif gpu == 0 then
    return test(valid_x, valid_y)
  else
    return test(valid_x:cuda(), valid_y:cuda())
  end
end

function train_acc()
  if structure == 'hsm' then
    return test({train_x, train_y}, train_y)
  else
    return test(train_x, train_y)
  end
end

function predict_kaggle()
  local fl = torch.DiskFile('training_output/newest_predictions_model=' .. lm ..
',dataset=' .. datafile .. '.txt', 'w')
  fl:writeString('ID') -- Header row
  for i = 1, test_dist[1]:size(1) do
    fl:writeString(',Class' .. i)
  end
  fl:writeString('\n')

  local preds = model:forward(test_x)
  for i = 1, preds:size(1) do
    local targets = test_dist[i]
    local i_preds = torch.Tensor(targets:size(1))
```

```
      for j = 1, targets:size(1) do
        i_preds[j] = preds[i][targets[j]]
      end
      i_preds = i_preds / i_preds:sum()

      fl:writeString(tostring(i))
      for j = 1, i_preds:size(1) do
        fl:writeString(',' .. i_preds[j])
      end
      fl:writeString('\n')
    end

    fl:close()
  end

  local inita, initl, initp = valid_acc()
  print('Validation perplexity before training: ' .. initp .. '.')
  print('Beginning training...')
  local vloss = torch.DoubleTensor(n_epochs) -- valid/train loss/accuracy/time
  local tloss = torch.DoubleTensor(n_epochs)
  local vacc = torch.DoubleTensor(n_epochs)
  local tacc = torch.DoubleTensor(n_epochs)
  local etime = torch.DoubleTensor(n_epochs)

  for i = 1, n_epochs do
    local timer = torch.Timer()
    train(i)

    local va, vl, vp = valid_acc()
    -- local ta, tl, tp = train_acc()

    -- vloss[i] = vl
    -- tloss[i] = tl
    vacc[i] = vp
    -- tacc[i] = tp
    etime[i] = timer:time().real

    print('Epoch ' .. i .. ' training completed in ' .. timer:time().real .. ' seconds.')
    print('Validation perplexity after epoch ' .. i .. ': ' .. vp .. '.')

    -- Logging
    local flr = torch.DiskFile('training_output/epoch=' .. i .. ',model=' .. lm .. ',dataset=' ..
datafile .. '.txt', 'w')
    for j = 1, i do
      -- flr:writeString(vacc[j] .. ',' .. tacc[j] .. ',' .. vloss[j] .. ','  .. tloss[j] .. ','  .. etime[j] .. '\n')
      -- flr:writeString(tacc[j] .. ',' .. vacc[j] .. ','  .. etime[j] .. '\n')
      flr:writeString(vacc[j] .. ','  .. etime[j] .. '\n')
```

```
    end
    flr:close()
  end

  -- Kaggle predictions
  predict_kaggle()

  -- Export lookup table weights
  local f = hdf5.open('embed_export50_hsm.hdf5', 'w')
  f:write('/embed', input_embedding.weight)
  f:close()

  local ta, tl, tp = train_acc()
  print('Final training perplexity after all epochs: ' .. tp)
end

function main()
  -- Parse input params
  opt = cmd:parse(arg)
  datafile = opt.datafile
  lm = opt.lm
  alpha = opt.alpha
  eta = opt.eta
  n_epochs = opt.nepochs
  batch_size = opt.mb
  k = opt.k
  gpu = opt.gpu

  local f = hdf5.open(opt.datafile, 'r')
  nwords = f:read('nwords'):all():long()[1]
  nclasses = f:read('nclasses'):all():long()[1]
  ngram = f:read('ngram'):all():long()[1]
  dwin = ngram - 1

  -- Split training and validation data
  train_x = f:read('train_input'):all()
  train_y = f:read('train_output'):all()
  valid_x = f:read('valid_input'):all()
  valid_y = f:read('valid_output'):all()
  test_x = f:read('test_input'):all()
  test_dist = f:read('test_output'):all()

  -- Format datasets for Witten-Bell
  if lm == 'wb' then
    if string.find(opt.datafile, '3') then
      interp = 3
      -- Need to load bigrams
```

```
      local f2 = hdf5.open('PTB_2gram.hdf5', 'r')
      bigram_train_x = f2:read('train_input'):all()
      bigram_train_y = f2:read('train_output'):all()
      bigram_valid_x = f2:read('valid_input'):all()
      bigram_valid_y = f2:read('valid_output'):all()

      trigram_train_x = train_x
      trigram_train_y = train_y
      trigram_valid_x = valid_x
      trigram_valid_y = valid_y
    else
      interp = 2
      bigram_train_x = train_x
      bigram_train_y = train_y
      bigram_valid_x = valid_x
      bigram_valid_y = valid_y
    end
  end

  if lm == 'mle' or lm == 'lap' or lm == 'wb' then
    count_based(lm)
  else
    nnlm(lm)
  end
end

main()
```