```
-- Only requirements allowed
require('hdf5')
require('nn')
-- require('cunn')
-- require('cutorch')

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-datafile', '', 'data file')
cmd:option('-lm', 'hmm', 'classifier to use')
cmd:option('-alpha', 0.01, 'Laplace smoothing coefficient')
cmd:option('-eta', 1, 'learning rate')
cmd:option('-nepochs', 15, 'number of training epochs')
cmd:option('-mb', 32, 'minibatch size')
cmd:option('-gpu', 0, 'whether to use gpu for training')
cmd:option('-avg', 0, 'whether to use weight averaging for sp training')

START = 1
STOP = 2
START_TAG = 8
STOP_TAG = 9

local tags = {}
tags[2] = 'PER'
tags[3] = 'LOC'
tags[4] = 'ORG'
tags[5] = 'MISC'
tags[6] = 'MISC'
tags[7] = 'LOC'

----------
-- Misc
----------

-- Finds the nth index of a given value in a tensor
function find_nth(t, val, n)
  local count = 0
  for i = 1, t:size(1) do
    if t[i] == val then
      count = count + 1
      if count == n then return i end
    end
  end
  return -1
end
```

```
-- Finds the first index of a given value in a tensor
function find_first(t, val)
  return find_nth(t, val, 1)
end

-- Removes excess repeated labels from end of a generic sequence
function chop(seq)
  local label = seq[seq:size(1)]
  local stop = find_first(seq, label)
  return seq[{{1, stop}}]
end

-- Removes start and stop labels from ends of a sequence
function slap(seq)
  return seq[{{2, seq:size(1) - 1}}]
end

-- Removes last value from a sequence
function chop_last(seq)
  return seq[{{1, seq:size(1) - 1}}]
end

-- Computes F score
function f_score(preds, y, beta)
  local cor_pos = 0
  local all_pos = 0
  local tru_pos = 0
  for i = 1, preds:size(1) do
    if preds[i] > 1 then
      all_pos = all_pos + 1
      if preds[i] == y[i] then cor_pos = cor_pos + 1 end
    end
    if y[i] > 1 then tru_pos = tru_pos + 1 end
  end

  local p = cor_pos / all_pos
  local r = cor_pos / tru_pos

  -- If all_pos or tru_pos are 0, then effectively we have 0/0
  -- which should be mapped to a precision or recall of 1
  if all_pos == 0 then p = 1 end
  if tru_pos == 0 then r = 1 end

  local b2 = beta * beta
  if b2 * p + r == 0 then return 0 end
  return (1 + b2) * ((p * r) / (b2 * p + r))
end
```

```
-- Logging
local function save_performance(name, t, v, flog)
  if flog == nil then flog = torch.zeros(nepochs) end
  local f = torch.DiskFile('training_output/' .. name .. '.txt', 'w')
  for j = 1, v:size(1) do
    f:writeString(t[j] .. ','  .. v[j] .. ',' .. flog[j] .. '\n')
  end
  f:close()
end

-- Kaggle predictions
local function format_kaggle(seq)
  local cur_tag = ''
  local output = ''
  for i = 1, seq:size(1) do
    local t = tags[seq[i]]
    if t == nil then
      cur_tag = 'O'
    elseif t ~= cur_tag then
      if output == '' then -- First tag
        cur_tag = t
        output = output .. cur_tag .. '-' .. i
      else -- Next new tag
        cur_tag = t
        output = output .. ' ' .. t .. '-' .. i
      end
    else -- Continuing current tag
      output = output .. '-' .. i
    end
  end
  return output
end

function save_kaggle(preds)
  local f = torch.DiskFile('training_output/kaggle_preds_model=' .. lm .. '.txt', 'w')
  f:writeString('ID,Labels\n') -- Header row

  for i = 1, #preds do
    f:writeString(tostring(i) .. ',' .. preds[i])
    f:writeString('\n')
  end

  f:close()
end

----------
```

```
-- Search
----------

-- Viterbi algorithm
-- observations: a sequence of observations, represented as integers
-- observations_t: a sequence of previous classes (only for memm, sp)
-- logscore: the edge scoring function over classes and observations in a
-- history-based model
function viterbi(observations, observations_t, logscore, initial_emission)
  local initial = torch.zeros(nclasses, 1)
  initial[START_TAG] = 1 -- Initial transition dist (always begins with start)
  initial:log()

  local n = observations:size(1)
  local max_table = torch.Tensor(n, nclasses)
  local backpointer_table = torch.Tensor(n, nclasses)

  -- First timestep
  -- Initial most likely paths are the initial state distribution
  local maxes, backpointers = (initial + initial_emission):max(2)
  if observations_t ~= nil then
    maxes, backpointers = initial_emission:max(2)
  end

  max_table[1] = maxes

  -- Remaining timesteps
  for i = 2, n do
    local y = logscore(observations, observations_t, i)
    local scores = y + maxes:view(1, nclasses):expand(nclasses, nclasses)
    maxes, backpointers = scores:max(2)
    max_table[i] = maxes
    backpointer_table[i] = backpointers
  end

  -- Use backpointers to recover optimal path
  local classes = torch.Tensor(n)
  maxes, classes[n] = maxes:max(1)
  for i = n, 2, -1 do
    classes[i-1] = backpointer_table[{i, classes[i]}]
  end

  return classes:int()
end

----------
-- HMM
```

```
-- Build simple HMM using the class to class transition distribution combined
-- with the class to word emission distribution
function hmm()
  print('Building HMM...')
  local transition = torch.zeros(nclasses, nclasses) -- p(y_{i}|y_{i-1})
  local emission = torch.zeros(nwords, nclasses) -- p(x_{i}|y_{i})

  for i = 1, train_x:size(1) do
    local x = chop(train_x[i])
    local y = chop(train_y[i])

    for j = 1, x:size(1) do
      local word = x[j]
      local class = y[j]
      emission[word][class] = emission[word][class] + 1

      if j < x:size(1) then
        transition[y[j+1]][class] = transition[y[j+1]][class] + 1
      end
    end
  end

  -- Normalize and log() transition probabilities
  transition = transition + alpha
  local sums = torch.sum(transition, 1)
  sums = sums:expand(nclasses, nclasses)
  transition:cdiv(sums):log()

  -- Normalize and log() emission probabilities
  emission = emission + alpha
  sums = torch.sum(emission, 1)
  sums = sums:expand(nwords, nclasses)
  emission:cdiv(sums):log()

  -- Log-scores of transition and emission
  -- i: timestep for the computed score
  function hmm_score(observations, observations_t, i)
    local observation_emission = emission[observations[i]]:view(nclasses, 1):
      expand(nclasses, nclasses)
    return observation_emission + transition
  end

  -- Predict optimal sequences in validation using viterbi
  print('Computing F1 score on validation...')
  local totalf = 0
```

```
  for i = 1, valid_x:size(1) do
    local x = chop(valid_x[i])
    local y = slap(chop(valid_y[i])) -- SLAP CHOP!!!
    local initial_emission = emission[x[1]]
    local classes = slap(viterbi(x, nil, hmm_score, initial_emission))
    local f = f_score(classes, y, 1) -- F1
    totalf = totalf + f
  end

  local validf = totalf / valid_x:size(1)
  print('Validation F1 score: ' .. validf .. '.')

  -- Predict optimal sequences on kaggle test using viterbi
  print('Computing Kaggle sequences...')
  local preds = {}
  for i = 1, test_x_w_s:size(1) do
    local x = chop(test_x_w_s[i]:view(test_x_w_s[i]:size(1)))
    local x = torch.LongTensor({1}):cat(x)
    local initial_emission = emission[x[1]]
    local classes = slap(viterbi(x, nil, hmm_score, initial_emission))
    preds[i] = format_kaggle(classes)
  end

  save_kaggle(preds)
  print('All done!')
end

----------
-- MEMM/SP
----------

function memm(lm)
  local name = 'Structured Perceptron'
  if lm == 'memm' then name = 'MEMM' end
  print('\nBuilding ' .. name .. ' with hyperparameters:')
  print('Learning rate (eta): ' .. eta)
  print('Number of epochs (nepochs): ' .. nepochs)
  print('Mini-batch size (mb): ' .. batch_size)

  local model = nn.Sequential()

  -- Word input
  local sparse_W_w = nn.LookupTable(nwords, nclasses)
  local W_w = nn.Sequential():add(sparse_W_w):add(nn.Sum(2))

  -- Tag input
  local sparse_W_t = nn.LookupTable(nclasses, nclasses)
```

```
local W_t = nn.Sequential():add(sparse_W_t):add(nn.Reshape(nclasses))

local par = nn.ParallelTable()
par:add(W_w)
par:add(W_t)

model:add(par)
model:add(nn.CAddTable())
if lm == 'memm' then
  model:add(nn.LogSoftMax())
else
  -- Zero lt weights
  sparse_W_w.weight:zero()
  sparse_W_t.weight:zero()
  if avg == 1 then
    print('Using cross-epoch weight averaging.')
  end
end

local nll = nn.ClassNLLCriterion()
local params, gradParams = model:getParameters()

if gpu == 1 then
  model:cuda()
  nll = nll:cuda()
  params = params:cuda()
  gradParams = gradParams:cuda()
  print('Using gpu accelerated training.')
end

-- Logging
local vacc = torch.DoubleTensor(nepochs)
local tacc = torch.DoubleTensor(nepochs)
local flog = torch.DoubleTensor(nepochs)

-- Scoring for viterbi
function memm_score(x_w, x_t, i)
  local w = torch.Tensor({{x_w[i]}})
  local t = torch.Tensor({{x_t[i]}})
  local score = model:forward({w, t}):view(nclasses, 1):expand(nclasses, nclasses)

  -- Prevent predicting start or stop tokens
  score:select(1, START_TAG):csub(100000)
  score:select(1, STOP_TAG):csub(100000)
  return score
end
```

```
function sp_score(x_w, x_t, i)
  local w = x_w[i]:view(1, 1):expand(nclasses, 1)
  local t = torch.range(1, nclasses):view(nclasses, 1) -- Calculate across all classes
  local score = model:forward({w, t})

  score:select(1, START_TAG):csub(100000)
  score:select(1, STOP_TAG):csub(100000)
  return score
end

function sp_score_prev(x_w, x_t, i)
  local w = torch.Tensor({{x_w[i][1]}})
  local t = torch.Tensor({{x_t[i][1]}})
  local score = model:forward({w, t}):view(nclasses, 1):expand(nclasses, nclasses)

  score:select(1, START_TAG):csub(100000)
  score:select(1, STOP_TAG):csub(100000)
  return score
end

function test(x_w, x_t, y)
  local preds = model:forward({x_w, x_t})
  local max, yhat = preds:max(2)

  local correct = yhat:int():eq(y):double():mean() * 100
  return correct
end
print(model)

-- F1 calculation
function f1()
  local totalf = 0
  for i = 1, valid_x_w_s:size(1) do
    local x_w = chop(valid_x_w_s[i]:view(valid_x_w_s[i]:size(1)))
    local x_t = chop(valid_x_t_s[i]:view(valid_x_t_s[i]:size(1)))
    x_t = chop_last(x_t) -- Remove final tag
    local y = chop_last(chop(valid_y_memm_s[i]:view(valid_y_memm_s[i]:size(1))))

    local classes = torch.Tensor()
    if lm == 'memm' then
      local initial_score = memm_score(x_w, x_t, 1):select(2,1)
      initial_score = initial_score:view(initial_score:size(1), 1)
      classes = chop_last(viterbi(x_w, x_t, memm_score, initial_score))
    else
      local w1 = torch.Tensor({{x_w[1]}})
      local initial_score = model:forward({w1, torch.Tensor({{START_TAG}})})
      initial_score = initial_score:view(initial_score:size(2), 1)
```

```
      x_w = x_w:view(x_w:size(1), 1)
      x_t = x_t:view(x_t:size(1), 1)
      classes = chop_last(viterbi(x_w, x_t, sp_score_prev, initial_score))
    end

    local f = f_score(classes, y, 1) -- F1
    totalf = totalf + f
  end
  return totalf / valid_x_w_s:size(1)
end

-- Train
local n_train_batches = train_x:size(1) / batch_size
local prev_acc = math.huge

if lm == 'memm' then
  -- MEMM Train
  for e = 1, nepochs do

    print('\nBeginning epoch ' .. e .. ' training: ' .. n_train_batches ..
      ' minibatches of size ' .. batch_size .. '.')
    local loss = 0
    local timer = torch.Timer()

    for j = 1, n_train_batches do
      model:zeroGradParameters()
      local x_w = train_x_w:narrow(1, (j - 1) * batch_size + 1, batch_size)
      local x_t = train_x_t:narrow(1, (j - 1) * batch_size + 1, batch_size)
      local y = train_y_memm:narrow(1, (j - 1) * batch_size + 1, batch_size)

      if gpu == 1 then
        x_w = x_w:cuda()
        x_t = x_t:cuda()
        y = y:cuda()
      end

      local preds = model:forward({x_w, x_t})
      loss = loss + nll:forward(preds, y)

      local dLdpreds = nll:backward(preds, y)
      model:backward({x_w, x_t}, dLdpreds)
      model:updateParameters(eta)
    end

    -- If accuracy isn't increasing from previous epoch, halve eta
    -- Only do this towards the end of training (be patient)
    local acc = test(valid_x_w, valid_x_t, valid_y_memm)
```

```
    if e > nepochs - 10 and  acc > prev_acc then
      eta = eta / 2
      print('Reducing learning rate to ' .. eta .. '.')
    end
    prev_acc = acc

    print('Epoch ' .. e .. ' training completed in ' .. timer:time().real ..
      ' seconds.')
    print('Validation accuracy after epoch ' .. e .. ': ' .. acc .. '.')

    local train_acc = test(train_x_w, train_x_t, train_y_memm)
    print('Train accuracy after epoch ' .. e .. ': ' .. train_acc .. '.')

    vacc[e] = acc
    tacc[e] = train_acc

    -- Test F1 on validation
    if e % 10 == 0 then
      print('\nCalculating validation F1...')
      local score = f1()
      print('Validation F1 score: ' .. score .. '.')
      flog[e] = score
    end
  end
else
  -- Structured Perceptron Train
  local prev_w_weight = torch.Tensor()
  local prev_t_weight = torch.Tensor()
  for e = 1, nepochs do

    -- Weight averaging across epochs
    if avg == 1 then
      local w_size = sparse_W_w.weight:size()
      local t_size = sparse_W_t.weight:size()
      prev_w_weight = torch.Tensor(w_size[1], w_size[2])
      prev_t_weight = torch.Tensor(t_size[1], t_size[2])

      prev_w_weight:copy(sparse_W_w.weight)
      prev_t_weight:copy(sparse_W_t.weight)
    end

    print('\nBeginning epoch ' .. e .. ' training.')
    local loss = 0
    local timer = torch.Timer()
    for j = 1, train_x_w_s:size(1) do
      model:zeroGradParameters()
      local x_w = chop(train_x_w_s[j]:view(train_x_w_s[j]:size(1)))
```

```
      local x_t = chop(train_x_t_s[j]:view(train_x_t_s[j]:size(1)))
      x_t = chop_last(x_t) -- Remove final tag
      local y = chop(train_y_memm_s[j]:view(train_y_memm_s[j]:size(1)))

      -- Enforce column structure
      x_w = x_w:view(x_w:size(1), 1)
      x_t = x_t:view(x_t:size(1), 1)
      -- y = y:view(y:size(1), 1)

      -- Find initial optimal sequence according to viterbi
      local initial_score = model:forward({x_w[1]:view(1, 1),
torch.Tensor({{START_TAG}})})
      initial_score = initial_score:view(initial_score:size(2), 1)
      local viterbi_preds = viterbi(x_w, x_t, sp_score, initial_score)

      -- Compare viterbi preds to gold sequence
      -- Create gradient by hand for backprop
      for i = 1, viterbi_preds:size(1) do
        if viterbi_preds[i] ~= tonumber(y[i]) then
          local w = x_w[i]:view(1, 1)

          -- Forward over predicted class rather than one seen in dataset
          local t = torch.Tensor({{START_TAG}})
          if i > 1 then
            t = torch.Tensor({{viterbi_preds[i-1]}})
          end

          local grad = torch.zeros(nclasses)
          local new_preds = model:forward({w, t})
          local max, pred = new_preds:max(2)

          local gold = y[i]
          local bad = pred[1][1]
          -- local bad = viterbi_preds[i]

          grad[gold] = -1
          grad[bad] = 1
          model:backward({w, t}, grad)
        end
      end

      model:updateParameters(eta)
    end

    -- Set lookup table weights to be a running average from previous
    -- Use a memory coefficient to decide how quickly to forget old weights
    -- remember = 0: vanilla sgd
```

```
      -- remember = 1: no learning
      if avg == 1 then
        remember = 0.85
        local forget = 1 - remember
        sparse_W_w.weight:mul(forget):add(remember, prev_w_weight)
        sparse_W_t.weight:mul(forget):add(remember, prev_t_weight)
      end

      -- If accuracy isn't increasing from previous epoch, halve eta
      -- Only do this towards the end of training (be patient)
      local acc = test(valid_x_w, valid_x_t, valid_y_memm)
      prev_acc = acc

      print('Epoch ' .. e .. ' training completed in ' .. timer:time().real ..
        ' seconds.')
      print('Validation accuracy after epoch ' .. e .. ': ' .. acc .. '.')

      local train_acc = test(train_x_w, train_x_t, train_y_memm)
      print('Train accuracy after epoch ' .. e .. ': ' .. train_acc .. '.')

      vacc[e] = acc
      tacc[e] = train_acc

      -- Test F1 on validation
      print('\nCalculating validation F1...')
      local score = f1()
      print('Validation F1 score: ' .. score .. '.')
      flog[e] = score
    end
  end

  -- Save to logfile
  if remember == nil then remember = 0 end
  local name = 'model=' .. lm .. ',f1=' .. flog[nepochs] .. ',mem=' .. remember
    .. ',eta=' .. eta
  save_performance(name, tacc, vacc, flog)
end

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

```
  gpu = opt.gpu
  avg = opt.avg

  local f = hdf5.open(opt.datafile, 'r')
  nclasses = f:read('nclasses'):all():long()[1]
  nwords = f:read('nwords'):all():long()[1]
  nfeatures = f:read('nfeatures'):all():long()[1]
  dwin = nfeatures / 2

   -- Split training, validation, test 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()

  train_x_w = f:read('train_input_w'):all()
  train_x_t = f:read('train_input_t'):all()
  train_y_memm = f:read('train_output_memm'):all()
  valid_x_w = f:read('valid_input_w'):all()
  valid_x_t = f:read('valid_input_t'):all()
  valid_y_memm = f:read('valid_output_memm'):all()

  -- Preserving sentence chunks
  train_x_w_s = f:read('train_input_w_s'):all()
  train_x_t_s = f:read('train_input_t_s'):all()
  train_y_memm_s = f:read('train_output_memm_s'):all()
  valid_x_w_s = f:read('valid_input_w_s'):all()
  valid_x_t_s = f:read('valid_input_t_s'):all()
  valid_y_memm_s = f:read('valid_output_memm_s'):all()

  test_x_w_s = f:read('test_input_w_s'):all()

  if lm == 'hmm' then
    hmm()
  elseif lm == 'memm' or lm == 'sp' then
    memm(lm)
  else
    print('No recognized classifier specified, bailing out.')
  end
end

main()
```