```
require("hdf5")
require("optim")
require("nn")
require("cunn")
require("cutorch")
require("rnn")

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-datafile', '', 'data file')
cmd:option('-lm', 'nn', 'classifier to use')
cmd:option('-eta', 0.1, 'learning rate')
cmd:option('-nepochs', 15, 'number of training epochs')
cmd:option('-mb', 32, 'minibatch size')
cmd:option('-ngram', 3, 'ngram size to use for context')
cmd:option('-gpu', 0, 'whether to use gpu for training')

NOSPACE = 0
SPACE = 1
STOP = 28

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

-- 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

-- Helper function that 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

-- Helper function that finds the first index of a given value in a
```

```
tensor
function find_first(t, val)
  return find_nth(t, val, 1)
end

-- Helper function that finds the frequency of a given value in a
tensor
function find_all(t, val)
  local count = 0
  for i = 1, t:size(1) do
    if t[i] == val then
      count = count + 1
    end
  end
  return count
end

-- Backs off a context
function back_off_context(ctx)
  local idx = ctx:match'^.*(),' - 1
  return string.sub(ctx, 1, idx)
end

-- To renormalize embedding weights
function renorm(data, th)
  for i = 1, data:size(1) do
    local norm = data[i]:norm()
    if norm > th then
      data[i]:div(norm / th)
    end
  end
end

-- To renormalize grad params
function renorm_grad(data, th)
  local norm = data:norm()
  if norm > th then
    data:div(norm / th)
  end
end

-- Logging
local function save_performance(name, tperp, vperp)
  local f = torch.DiskFile('training_output/' .. name .. '.txt', 'w')
  for j = 1, vperp:size(1) do
    f:writeString(tperp[j] .. ','  .. vperp[j] .. '\n')
  end
  f:close()
end
```

```
-- Kaggle prediction
function predict_kaggle(preds)
  local f = torch.DiskFile('training_output/kaggle_preds_model=' ..
lm .. '.txt', 'w')
  f:writeString('ID,Count\n') -- Header row

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

  f:close()
end

----------
-- Sequence Search
----------

function greedy_search(seq, ngram, p_ngram, cb, cutoff)
  ngram = ngram - 1 -- ctx length = ngram - 1

  -- Assume that the first ngram chars are context from a previous
example
  local idx = ngram
  -- local score = 0
  while idx < seq:size(1) do
    local ctx = seq[{{idx-ngram+1, idx}}]
    local p_space = 0

    if cb then
      ctx = tensor_to_key(ctx)
      p_space = p_ngram(ctx, SPACE, true)
    else
      p_space = p_ngram(ctx, SPACE+1)
    end

    if p_space > cutoff then
      -- Need to insert a space when p(space) > 0.5
      local prev = seq[{{1, idx}}]
      local post = seq[{{idx+1, seq:size(1)}}]
      seq = prev:cat(torch.IntTensor({1})):cat(post)
    end
    idx = idx + 1
  end

  return find_all(seq, SPACE)
end

-- Bigram only for now
function viterbi_search(seq, ngram, p_ngram, cb, cutoff)
```

```
  ngram = 1 -- 2 - 1
  local head = torch.Tensor(nclasses)

  -- Assume that the first ngram chars are context from a previous
example
  local larger_score = 0
  local idx = ngram
  while idx < seq:size(1) do
    head[1] = larger_score
    head[2] = larger_score
    local ctx = seq[{{idx-ngram+1, idx}}]

    if cb then
      ctx = tensor_to_key(ctx)
      head[1] = head[1] + math.log(p_ngram(ctx, NOSPACE, true))
      head[2] = head[2] + math.log(p_ngram(ctx, SPACE, true))
    else
      head[1] = head[1] + math.log(p_ngram(ctx, NOSPACE+1))
      head[2] = head[2] + math.log(p_ngram(ctx, SPACE+1))
    end

    if head[2] > head[1] then
      -- Need to insert a space when head[2] -- p(space) -- is
preferred
      local prev = seq[{{1, idx}}]
      local post = seq[{{idx+1, seq:size(1)}}]
      seq = prev:cat(torch.IntTensor({1})):cat(post)
      larger_score = head[2]
    else
      larger_score = head[1]
    end
    idx = idx + 1
  end

  return find_all(seq, SPACE)
end

function predict(x, y, ngram, search, cb, cutoff)
  local se = 0

  -- Example 1 has no preceding ctx
  local stop = find_first(x[1], STOP) -- (STOP = </s>)
  local seq = x[{1, {1, stop}}] -- Chop off unnecessary </s>
  local spaces = search(seq, ngram, p_ngram, cb, cutoff)
  se = se + math.pow(spaces - y[1], 2)
  local prev_ctx = seq[{{stop - ngram + 1, stop - 1}}] -- Don't grab
last </s>
  prev_ctx:cat(torch.ones(1):int()) -- Add space

  local high = 0
```

```
  local low = 0

  for i = 2, x:size(1) do
    -- Append context from end of previous example
    seq = prev_ctx:cat(x[i])
    stop = find_first(seq, STOP)
    seq = seq[{{1, stop}}]
    spaces = search(seq, ngram, p_ngram, cb, cutoff)
    se = se + math.pow(spaces - y[i], 2)
    prev_ctx = seq[{{stop - ngram + 1, stop - 1}}]
    prev_ctx:cat(torch.ones(1):int()) -- Add space
    -- print(spaces, y[i])

    if spaces - y[i] > 0 then
      high = high + 1
    elseif y[i] - spaces > 0 then
      low = low + 1
    end
  end

  print('high: ' .. high .. ', low: ' .. low)
  return se / x:size(1)
end

----------
-- Count based model
----------

function count_based(max_ngram)
  ngrams = {}

  -- Calculate the space/no-space distribution
  function build_unigram(y)
    local total = 0
    local spaces = 0
    for i = 1, y:size(1) do
      local s = y[i]
      if s == 1 then
        spaces = spaces + 1
      end
      total = total + 1
    end

    local unigram = {}
    unigram[0] = (total - spaces) / total
    unigram[1] = spaces / total
    return unigram
  end

  -- Create co-occurrence dictionary mapping contexts to following
```

```
words
  function build_ngram(x, y, ngram)
    print('Building p(space|w_i-n+1,...,w_i-1) for i=' .. ngram ..
'...')
    ngram = ngram - 1 -- Shorten by 1 to get context length
    local grams = {}
    for i = ngram, x:size(1) do
      local ctx = tensor_to_key(x:narrow(1, i-ngram+1, ngram))
      local s = y[i]
      local val = grams[ctx]
      if val == nil then
        grams[ctx] = {}
        grams[ctx][s] = 1
      else
        local innerval = grams[ctx][s]
        if innerval == nil then
          grams[ctx][s] = 1
        else
          grams[ctx][s] = grams[ctx][s] + 1
        end
      end

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

  -- Renormalize probabilities for each ngram
  function normalize_ngram(ngram)
    print('Renormalizing ngram probabilities...')
    for ctx, ys in pairs(ngram) do
      -- Total number of spaces/non-spaces we've seen in this context
      local tot0 = ngram[ctx][0] or 0
      local tot1 = ngram[ctx][1] or 0
      local sum = tot0 + tot1

      -- Convert to space/non-space probabilities for each context
      for s, tot in pairs(ys) do
        ngram[ctx][s] = ngram[ctx][s] / sum
      end
    end
  end

  function p_ngram(ctx, s, seg)
    -- Select probability from longest valid context
    -- If no probability is established at this context, continue
backing off
    -- until one is found, all the way till unigram if necessary
```

```
    local found = false
    local g = max_ngram
    local p = 0

    -- Hack to ensure p(space) = 0 if the last ctx is a space (for
unseen ctx)
    -- Only use during segmentation
    local last = tonumber(string.sub(ctx, -1))
    if seg and last == SPACE then
      return 0.001 -- Don't actually return 0, need log probs
    end

    function back_off()
      g = g - 1
      if g == 1 then
        p = ngrams[g][s]
        found = true
      else
        ctx = back_off_context(ctx)
      end
    end

    while not found do
      ctxd = ngrams[g][ctx]
      if ctxd ~= nil then
        p = ngrams[g][ctx][s]
        if p ~= nil then
          found = true
        else
          back_off()
        end
      else
        back_off()
      end
    end

    return p
  end

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

      -- Prevent log(0)
      if p == 0 then
        p = 0.001
```

```
      end

      sum = sum + math.log(p)
    end

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

  print('Building count based model (ngram=' .. max_ngram .. ')...')
  -- Build all ngrams <= max ngram for backoff in unseen cases
  local unigram = build_unigram(train_y)
  ngrams[1] = unigram
  for i = max_ngram, 2, -1 do
    local ngram = build_ngram(train_x_cb, train_y_cb, i)
    normalize_ngram(ngram)
    ngrams[i] = ngram
  end

  print('Calculating greedy mse on validation segmentation...')
  local mse = predict(valid_kaggle_x, valid_kaggle_y, max_ngram,
greedy_search, true, 0.5)
  print('Validation segmentation mse: ' .. mse)

  print('Calculating Viterbi mse on validation segmentation...')
  -- Bigram only for now
  local mse = predict(valid_kaggle_x, valid_kaggle_y, 2,
viterbi_search, true)
  print('Validation segmentation mse: ' .. mse)

  print('Calculating perplexity on train...')
  local perp = perplexity(train_x_cb, train_y_cb, max_ngram - 1)
  print('Training perplexity: ' .. perp)

  print('Calculating perplexity on valid...')
  local perp = perplexity(valid_x_cb, valid_y_cb, max_ngram - 1)
  print('Validation perplexity: ' .. perp)
end

----------
-- NNLM
----------

function nnlm(dwin)
  local embedding_size = 100
  local din = embedding_size * dwin
  local dhid = 100
  local dout = nclasses

  print('\nBuilding neural language model with hyperparameters:')
```

```
print('Learning rate (eta): ' .. eta)
print('Number of epochs (nepochs): ' .. nepochs)
print('Mini-batch size (mb): ' .. batch_size)
print('Context window (dwin): ' .. dwin)
print('Embedding size (embed): ' .. embedding_size)

-- Build examples of ngram size dwin + 1
function build_ngram_windows(x_cb, y_cb)
  local ngram_x = torch.IntTensor(x_cb:size(1) - dwin, dwin)
  local ngram_y = torch.IntTensor(y_cb:size(1) - dwin)

  for i = dwin, x_cb:size(1) - 1 do
    local x = x_cb:narrow(1, i-dwin+1, dwin)
    local y = y_cb[i]
    ngram_x[{i-dwin+1, {1, dwin}}] = x
    ngram_y[i-dwin+1] = y + 1 -- Class labels can't = 0
  end

  return ngram_x, ngram_y
end

local train_x, train_y = build_ngram_windows(train_x_cb, train_y_cb)
local valid_x, valid_y = build_ngram_windows(valid_x_cb, valid_y_cb)
local model = nn.Sequential()

-- Lookup table concats embeddings for chars in context
input_embedding = nn.LookupTable(nletters, 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()

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

-- Perplexity check
function test(x, y)
  local preds = model:forward(x)
```

```
    local loss = nll:forward(preds, y)
    return math.exp(loss)
  end

  -- Probability for specific context
  function p_ngram(x, idx)
    local pred = model:forward(x)
    return math.exp(pred[idx])
  end

  -- Logging
  local vperp = torch.DoubleTensor(nepochs)
  local tperp = torch.DoubleTensor(nepochs)

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

  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 = train_x:narrow(1, (j - 1) * batch_size + 1,
batch_size)
      local y = train_y:narrow(1, (j - 1) * batch_size + 1,
batch_size)

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

      local preds = model:forward(x)
      loss = loss + nll:forward(preds, y)

      local dLdpreds = nll:backward(preds, y)
      model:backward(x, dLdpreds)
      model:updateParameters(eta)
    end

    -- If perplexity increases from previous epoch, halve eta
    local perp = test(valid_x, valid_y)
    if perp > prev_perp or math.abs(perp - prev_perp) < 0.002 then
      eta = eta / 2
      print('Reducing learning rate to ' .. eta .. '.')
```

```
    end
    prev_perp = perp

    -- Renormalize the weights of the lookup table
    renorm(input_embedding.weight, 1)

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

    local train_perp = test(train_x, train_y)
    print('Train perplexity after epoch ' .. e .. ': ' ..
train_perp .. '.')

    vperp[e] = perp
    tperp[e] = train_perp
  end

  print('\nCalculating greedy mse on validation segmentation...')
  local mse = predict(valid_kaggle_x, valid_kaggle_y, dwin+1,
greedy_search, false, 0.4)
  print('Validation segmentation mse: ' .. mse)

  -- Save to logfile
  local name = 'model=' .. lm .. ',dwin=' .. dwin .. ',dembed='
  .. embedding_size .. ',mse=' .. mse
  save_performance(name, tperp, vperp)
end

----------
-- RNN
----------

function rnn(structure)
  local embedding_size = 50

  local name = 'LSTM'
  if structure == 'gru' then name = 'GRU'
  elseif structure == 'stack' then name = 'Stacked LSTM' end

  print('\nBuilding ' .. name .. ' with hyperparameters:')
  print('Learning rate (eta): ' .. eta)
  print('Number of epochs (nepochs): ' .. nepochs)
  print('Mini-batch size (mb): ' .. batch_size)
  print('Sequence length (seq): ' .. seq_len)
  print('Embedding size (embed): ' .. embedding_size)

  -- Adjust targets from 0,1 -> 1,2
```

```
  -- NB BCECrit uses 0,1
  train_y = train_y + 1
  valid_y = valid_y + 1

  local model = nn.Sequential()

  -- Lookup table embeddings for chars in context
  input_embedding = nn.LookupTable(nletters, embedding_size)
  model:add(input_embedding)
  model:add(nn.SplitTable(1))

  if structure == 'gru' then
    model:add(nn.Sequencer(nn.GRU(embedding_size,
embedding_size)):remember('both'))
  elseif structure == 'stack' then
    model:add(nn.Sequencer(nn.FastLSTM(embedding_size,
embedding_size)):remember('both'))
    model:add(nn.Sequencer(nn.Dropout(0.5)))
    model:add(nn.Sequencer(nn.FastLSTM(embedding_size,
embedding_size)):remember('both'))
  else -- Single layer LSTM
    model:add(nn.Sequencer(nn.FastLSTM(embedding_size,
embedding_size)):remember('both'))
  end
  model:add(nn.Sequencer(nn.Linear(embedding_size, 2)))

  -- model:add(nn.Sequencer(nn.Sigmoid())
  -- nll = nn.SequencerCriterion(nn.BCECriterion())
  model:add(nn.Sequencer(nn.LogSoftMax()))
  nll = nn.SequencerCriterion(nn.ClassNLLCriterion())

  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

  -- Initialize params to uniform (-0.05, 0.05)
  torch.manualSeed(1234)
  local unif = torch.rand(params:size(1)) -- [0, 1)
  unif = unif / 10 -- [0, 0.1)
  unif = unif - 0.05 -- [-0.05, 0.05)
  params = params:copy(unif)

  -- Perplexity check
  function test(x, y)
    local loss = 0
```

```
    for j = 1, x:size(1) do
      local xj = x[j]:t()
      local yj = y[j]:t()
      local preds = model:forward(xj)
      loss = loss + nll:forward(preds, yj) / x:size(3) -- Divide by
seq len
    end
    loss = loss / x:size(1) -- Batch avg
    return math.exp(loss)
  end

  function find_nth_space(t, n, cutoff)
    local count = 0
    for i = 1, #t do
      if math.exp(t[i][2]) > cutoff then
        count = count + 1
        if count == n then return i end
      end
    end
    return -1
  end

  function rnn_greedy_search(seq)
    -- Pass sequence to rnn to recover first index of predicted space
    -- Save and input first estimated space into sequence
    -- Repeat

    -- model:evaluate()
    model:forget()
    local stop = find_first(seq, STOP) -- (STOP = </s>)
    if stop == -1 then stop = seq:size(1) end
    local seq = seq[{{1, stop}}] -- Chop off unnecessary </s>

    local cutoff = 0.3 -- 0.28
    local idx = 1
    local count = 1
    while idx > 0 do
      local preds = model:forward(seq)
      idx = find_nth_space(preds, count, cutoff)

      if idx > 0 then
        if idx+1 > seq:size(1) then -- Handle end prediction
          local prev = seq[{{1, idx}}]
          seq = prev:cat(torch.IntTensor({1}))
        else
          local prev = seq[{{1, idx}}]
          local post = seq[{{idx+1, seq:size(1)}}]
          seq = prev:cat(torch.IntTensor({1})):cat(post)
          count = count + 1
        end
```

```
      end
    end

    return count - 1
  end

  function rnn_predict(x, y, n_examples, test)
    if n_examples > x:size(1) or n_examples < 1 then n_examples =
x:size(1) end
    local preds = {}
    local se = 0
    local high = 0
    local low = 0
    for i = 1, n_examples do
      seq = x[i]
      spaces = rnn_greedy_search(seq)

      if not test then
        se = se + math.pow(spaces - y[i], 2)
        if spaces - y[i] > 0 then
          high = high + 1
        elseif y[i] - spaces > 0 then
          low = low + 1
        end
      end
      preds[i] = spaces
      -- print(spaces, y[i])
    end

    print('high: ' .. high .. ', low: ' .. low)
    if test then return preds end
    return (se / n_examples)
  end

  -- Logging
  local vperp = torch.DoubleTensor(nepochs)
  local tperp = torch.DoubleTensor(nepochs)

  -- Train
  local n_examples = train_x:size(1)
  local prev_perp = math.huge

  for e = 1, nepochs do
    model:training()
    print('\nBeginning epoch ' .. e .. ' training: ' .. n_examples ..
      ' minibatches of size ' .. batch_size .. '.')
    local loss = 0
    local timer = torch.Timer()

    for j = 1, n_examples do
```

```
      model:zeroGradParameters()
      local x = train_x[j]:t()
      local y = train_y[j]:t()

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

      local preds = model:forward(x)
      loss = loss + nll:forward(preds, y)

      local dLdpreds = nll:backward(preds, y)
      model:backward(x, dLdpreds)

      -- Normalize grad params to max norm = 5
      renorm_grad(gradParams, 5)
      model:updateParameters(eta)
    end

    -- If perplexity increases/stays same from previous epoch, halve
eta
    model:evaluate()
    local perp = test(valid_x, valid_y)
    if perp > prev_perp or math.abs(perp - prev_perp) < 0.002 then
      eta = eta / 2
      print('Reducing learning rate to ' .. eta .. '.')
    end
    prev_perp = perp

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

    local train_perp = test(train_x, train_y)
    print('Train perplexity after epoch ' .. e .. ': ' ..
train_perp .. '.')

    vperp[e] = perp
    tperp[e] = train_perp
  end

  print('\nCalculating greedy mse on validation segmentation...')
  local subset = 1000
  local mse = rnn_predict(valid_kaggle_x, valid_kaggle_y, subset,
false)
  print('Validation segmentation mse: ' .. mse)
```

```
  -- Save to logfile
  local name = 'model=' .. lm .. ',dembed=' .. embedding_size ..
',mse=' .. mse
  save_performance(name, tperp, vperp)

  -- Predict spaces for kaggle test
  -- local preds = rnn_predict(test_x, nil, -1, true)
  -- predict_kaggle()
end

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

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

  -- Split training, validation, test data
  train_x_cb = f:read('train_input_cb'):all()
  train_y_cb = f:read('train_output_cb'):all()
  valid_x_cb = f:read('valid_input_cb'):all()
  valid_y_cb = f:read('valid_output_cb'):all()

  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()
  valid_kaggle_x = f:read('valid_kaggle_input'):all()
  valid_kaggle_y = f:read('valid_kaggle_output'):all()
  test_x = f:read('test_input'):all()

  if lm == 'cb' then
    count_based(ngram)
  elseif lm == 'nn' then
    nnlm(ngram - 1)
  else
    batch_size = f:read('batch'):all():long()[1]
    seq_len = f:read('seq'):all():long()[1]
    rnn(lm)
  end

end
```

```
main()
```