HW4: Word Segmentation

Virgile Audi
vaudi@g.harvard.edu
Nicolas Drizard
nicolasdrizard@g.harvard.edu
Github: virgodi/cs287/HW5

April 17, 2016

1 Introduction

The goal of this assignement is to tackle the NLP task of identifying and labeling contiguous segments of text. We will use sequence models and a dynamic programming method to find the best scoring sequence.

2 Problem Description

The idea is here to label continuous sequence of words with BIO tagging of different entities. The entities are the following:

1. PER: a person
2. LOC: a location
3. ORG: an organization
4. MISC:

Furthermore, this tagging method identifies the continuous group of words belonging to the same entity: the prefix B stop the current tag and begins a new one whereas the prefix I continues adding to the previous tag. However, in our solution we just cared about predicting the entity tag and then we were grouping the contiguous predictions into the same entity because the training text does not contain any B-tag.

3 Model and Algorithms

We used three different methods to solve this problem. The first two are the equivalent of first the Naive Bayes and second the logistic regression from text classification tasks. The last one introduces a customized way to train a neural architecture for this task.

3.1 Hidden Markov Model

We implement here a standard first order hidden Markov Model. The hidden states are the tags and the observed states are the features we built (word counts, capitalization...). The model can be represented with the following graphical model and requires two distribution: emission and transition.

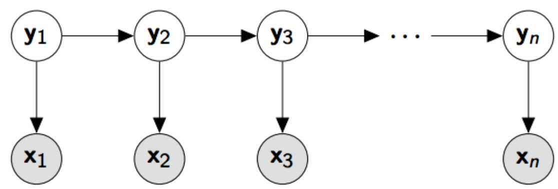


Figure 1: Graphical model of 1st order HMM with one feature

We represent the two distrubitions with multinomial as they model feature counts. As a result, we can infer them simply with the maximum likelihood estimator:

$$p(x_i = \delta(f) | y_i = \delta(c)) = \frac{F_{f,c}}{F_{.,c}}$$

$$p(y_i = \delta(c_i) | y_{i-1} = \delta(c_{i-1})) = \frac{T_{c_{i-1},c_i}}{T_{c_{i-1},.}}$$

with T_{c_{i-1},c_i} the counts of class c_{i-1} preceding class c_i and $F_{f,c}$ the counts of emission f with class c.

If we consider multiple features, then we still assume that the feature are indepent with each other (it's the main assumption in the Naive Bayes approach also). Only the emission distribution is changed and we can combine the probability together:

$$p(x_i = (\delta(f_1), \delta(f_2)) | y_i = \delta(c)) = p(x_i = \delta(f_1) | y_i = \delta(c)) p(x_i = \delta(f_2) | y_i = \delta(c)) = \frac{F_{f_1,c}}{F_{.,c}} \frac{F_{f_2,c}}{F_{.,c}}$$

3.2 Maximum-Entropy Markov Model

Next, we implemented a Maximum-Entropy Markov Model. The objective of the MEMM is to evaluate at each time step a distribution over the possible tags using features of the current word, denoted as $feat(x_i)$ and the tag of the previous word, c_{i-1}, using multi-class logistic regression, i.e.

$$p(\mathbf{y}_i | \mathbf{y}_{i-1}, feat(x_i)) = \text{softmax}([feat(x_i), c_{i-1}]\mathbf{W} + \mathbf{b})$$

3.3 Viterbi algorithm

The search algorithm that we implemented is the dynamic programming algorithm named after Andrew Viterbi. Its main difference with a greedy approach is that it evaluates at every step and for every previous state, the best possible next step. This guarantees a solution closer to the true optimal solution. The pseudo-code of the algorithm is given by:

procedure VITERBIWITHBP
  $\pi \in \mathbb{R}^{n+1 \times \mathcal{C}}$ initialized to $-\infty$
  $bp \in \mathcal{C}^{n \times \mathcal{C}}$ initialized to ϵ
  $\pi[0, \langle s \rangle] = 0$
  for $i = 1$ to n **do**
    for $c_{i-1} \in \mathcal{C}$ **do**
      compute $\hat{\boldsymbol{y}}(c_{i-1})$
      for $c_i \in \mathcal{C}$ **do**
        $score = \pi[i-1, c_{i-1}] + \log \hat{\boldsymbol{y}}(c_{i-1})_{c_i}$
        if $score > \pi[i, c_i]$ **then**
          $\pi[i, c_i] = score$
          $bp[i, c_i] = c_{i-1}$
  return sequence from bp

3.4 Structured Perceptron

The final model, we implemented is the structure perceptron train algorithm. The way the model is trained uses the Viterbi search algorithm, presented above. At each epoch, we uses Viterbi to predict the highest scored sequence given the state of the model. We can then find the timesteps where the actual sequence for the given sentence and the predicted one differ and compute at each of these time steps, the gradient of a hinge type loss. These gradients have a -1 entry on the true class for this given word, and a 1 on the predicted class by the model. We can then propagate these gradients in the network, and update the weights with a learning rate that can be tuned.

The model itself is similar to the model of the MEMM without the final logsoftmax layer.

4 Experiments

4.1 Feature Engineering

The original paper suggests several features to use. We focus on the word counts and a capitalization feature. We defined our capitalization feature as follow:

1. 1 : word in low caps;
2. 2 : whole word in caps;
3. 3 : first letter in cap;
4. 4 : one cap in the word;

5. 5 : other

We then produced an embedding of the word counts using a pre-trained version.

We also used the Python "pattern.en" package to extract Part-of-Speach (PoS) features. The packages generates 41 features to which we added special feature for the opening and closing tabs <s> and $< \backslash$s$>$.

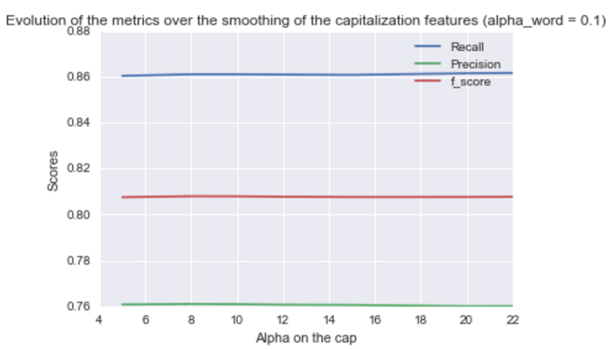
4.2 Model Evaluation

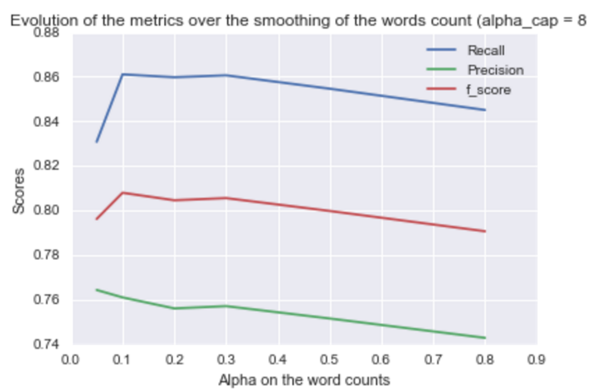
As used in the Kaggle competition, we used the f-score with the precision and recall measure to evaluate our model while tuning the hyperparameters. A positive prediction stands for a label (in the notation of the task, everything which is not the **O** tag):

1. recall: ratio of the true positive predictions among the positives tags in the correct sequence
2. precision: ratio of the true positive predictions among the positive predictions,
3. f-score (with $\beta = 1$): harmonic mean of the precision and the recall, i.e. $f_1 = \frac{2pr}{p+r}$

4.3 Hidden Markov Model

There is only the smoothing parameter α and eventually feature selection here to tune here. We evaluate the impact of adding more features and run experiments with different alpha values to tune them . One important details is to make sure to use a specific smoothing parameter for each distribution, i.e a smoothing parameter may be applied to the transition matrix but also to the emission matricx of each different feature. Each of this distribution has a different tail and need a different smoothing. For instance, the transition matrix need a very small α (around 0.1) because we are pretty confident in it but the capitalizations feature need one much bigger (around 20) because the counts are already high.





We notice that the model is less sensitive to the changes of the smoothing parameter on the capitalization feature as on the word counts. This is pretty reasonable as the feature coutns are much higher in the capitalization feature than in the word counts. Tuning this parameter provides a model with a f-score of **0.808**. Using only the word counts features provide a best f-score of **0.764**.

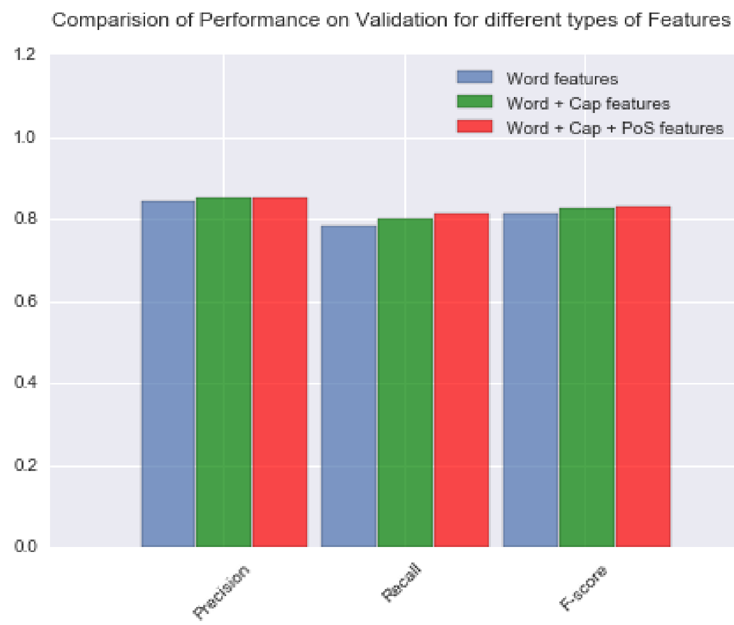
We obtained a Kaggle score on the test set of :

$$K_{HMM} = 0.48365$$

4.4 Maximum-Entropy Markov Model

We coded the MEMM using the nn module and trained using stochastic gradient descent. We also used the Glove embedggins using a lookup table. As for the HMM, we used two different sets of features, i.e. the words and the words and capitalisation of the words. We also added the Part of Speech features that were evaluated using the python package ”pattern.en” in order to gain some time. We observed that the training algorithm converges quite rapidely, and that if adding caps to the features helped decrease the loss, the impact was not as strong as expected. On the other hand, adding PoS features impacted greatly the loss. Nevertheless, we trained the model on 20 epochs in order to learn the embeddings for the <s> and < \s> ”words” added during pre-processing.

We evaluated the performance of these two models using the f-score presented above:



Adding extra features yielded better results on both Precision and Recall and therefore on the f-score. But as we expected from the small differences in loss, we did not observe an important increase on the f-score using cap features. We were nevertheless surprised to see that the impact on loss using PoS features did not translate on the f-score. These results were later confirmed on the test set, as the kaggle score obtained for these two models were:

$$K_{nocaps} = 0.52057 \quad \text{and} \quad K_{caps} = 0.55482 \quad K_{PoS} = 0.57121$$

which are both slightly better than the results of the HMM.

4.5 Structured Perceptron

5 Conclusion

This segmentation task gave us the opportunity to implement different recurrent neural network architectures but also to compare them with more traditionnal method. Whereas the count based and even the simple neural network models are pretty fast to train they still provide interesting results. The results provided by the three variants of RNN were interesting to illustrate the influence of gates and memory in such networks. The gated reccurent network ended as the best model on this task. One future work could be to stack more layers to our reccurent architecture or to implement a network with a dynamic memory part to give more flexibility in how the model uses the information it already processed.

Appendices

Preprocessing:

```
#!/usr/bin/env python

"""NER Preprocessing
"""

import numpy as np
import h5py
import argparse
import sys
import re
import codecs

# Your preprocessing, features construction, and word2vec code.


FILE_PATHS = {"CONLL": ("data/train.num.txt",
                        "data/dev.num.txt",
                        "data/test.num.txt",
                        "data/tags.txt")}
args = {}


def main(arguments):
    global args
    parser = argparse.ArgumentParser(
        description=__doc__,
        formatter_class=argparse.RawDescriptionHelpFormatter)
    parser.add_argument('dataset', help="Data set",
                        type=str)
    args = parser.parse_args(arguments)
    dataset = args.dataset
    train, valid, test, tag_dict = FILE_PATHS[dataset]

    filename = args.dataset + '.hdf5'
    with h5py.File(filename, "w") as f:
        f['train_input'] = train_input
        f['train_output'] = train_output
        if valid:
            f['valid_input'] = valid_input
            f['valid_output'] = valid_output
        if test:
```

```
            f['test_input'] = test_input
        f['nfeatures'] = np.array([V], dtype=np.int32)
        f['nclasses'] = np.array([C], dtype=np.int32)


if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))
```

Hidden Markov Model:

```
-- Documentation:
-- ---------- How to call it from the command line?
-- For example:
-- $ th count_based.lua -N 5
-- Other argument possible (see below)
--
-- ---------- Is there an Output?
-- By default, the predictions on the test set are saved in hdf5 format as classifier .. opt.f

-- Only requirements allowed
require("hdf5")
require 'helper.lua';

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-datafile', 'data/words_feature.hdf5',
           'Datafile with features in hdf5 format')
cmd:option('-alpha_t', 0.1, 'Smoothing parameter alpha in the transition counts')
cmd:option('-alpha_w', 2, 'Smoothing parameter alpha in the word counts')
cmd:option('-alpha_c', 20, 'Smoothing parameter alpha in the caps counts')
cmd:option('-test', 0, 'Boolean (as int) to ask for a prediction on test, will be saved in submission in hdf5 format')
cmd:option('-datafile_test', 'submission/v_seq_hmm', 'Smoothing parameter alpha in the word counts')
cmd:option('-nfeatures', 2, 'Number of type of features to use')
cmd:option('-cv', 0, 'Boolean (as int) to run a cross validation pipeline')



-- Formating as log-probability and smoothing the input
```

```lua
function format_matrix(matrix, alpha)
    local formatted_matrix = matrix:clone():type('torch.DoubleTensor')
    formatted_matrix:add(alpha)
    -- Normalize
    local norm_mat = torch.expandAs(formatted_matrix:sum(1), formatted_matrix)
    formatted_matrix:cdiv(norm_mat)
    return formatted_matrix:log()
end

-- log-scores of transition and emission
-- corresponds to the vector y in the lecture notes
-- i: timestep for the computed score
function score_hmm(observations, i, emissions, transition, C, nfeatures)
    local observation_emission = torch.zeros(C)
    for k=1,nfeatures do
        observation_emission:add(emissions[k][observations[{i,k}]])
    end
    observation_emission = observation_emission:view(C, 1):expand(C, C)
    -- NOTE: allocates a new Tensor
    return observation_emission + transition
end

-- Viterbi algorithm.
-- observations: a sequence of observations, represented as integers
-- logscore: the edge scoring function over classes and observations in a history-based model
function viterbi(observations, logscore, emissions, transition, nfeatures)
    local y
    -- Formating tensors
    local initial = torch.zeros(transition:size(2), 1)
    -- initial started with a start of sentence: <t>
    initial[{8,1}] = 1
    initial:log()

    -- number of classes
    C = initial:size(1)
    local n = observations:size(1)
    local max_table = torch.Tensor(n, C)
    local backpointer_table = torch.Tensor(n, C)

    -- first timestep
    -- the initial most likely paths are the initial state distribution
```

```lua
    -- NOTE: another unnecessary Tensor allocation here
    local init_pred = initial:clone()
    for i=1,nfeatures do
        init_pred:add(emissions[i][observations[{1,i}]])
    end
    local maxes, backpointers = init_pred:max(2)
    max_table[1] = maxes

    -- remaining timesteps ("forwarding" the maxes)
    for i=2,n do
        -- precompute edge scores
        y = logscore(observations, i, emissions, transition, C, nfeatures)
        scores = y + maxes:view(1, C):expand(C, C)

        -- compute new maxes (NOTE: another unnecessary Tensor allocation here)
        maxes, backpointers = scores:max(2)

        -- record
        max_table[i] = maxes
        backpointer_table[i] = backpointers
    end
    -- follow backpointers to recover max 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
end

-- Prediction pipeline
function predict(observations, emissions, transition, alphas, nfeatures)
    -- Formating model parameters (log and alpha smoothing)
    -- Alphas is a tensor : {alpha_t, alpha_w, alpha_c}
    emissions_cleaned = {}
    for i=1,nfeatures do
        emissions_cleaned[i] = format_matrix(emissions[i], alphas[i+1])
    end
    local transition_cleaned = format_matrix(transition, alphas[1])

    return viterbi(observations, score_hmm, emissions_cleaned,
```

```
        transition_cleaned, nfeatures)
end

-- Cross validation pipeline
function cross_validation(observations, emissions, transitions,
    true_classes,
                          alphas_table, alpha_t)
    -- alphas_table is a table of tensor with the range of parameters
        to use
    -- Current implementation for 2 features only
    -- alphas_table = {alpha_w_tensor, alpha_c_tensor}
    -- Return a tensor with first columns the alpha value and last the
        score for each
    local nfeatures = #alphas_table
    local v_seq_dev, precision, recall, f
    local alphas = torch.DoubleTensor(3)
    local size1 = alphas_table[1]:size(1)
    local size2 = alphas_table[2]:size(1)
    local num_evaluations = size1*size2

    -- Columns for 2 features are (alphas_w_value, alphas_c_value,
        f_score, precision, recall)
    local scores = torch.DoubleTensor(num_evaluations, nfeatures+3)

    for i=1,size1 do
        alpha_w = alphas_table[1][i]
        for k=1,size2 do
            alpha_c = alphas_table[2][k]

            alphas:copy(torch.Tensor({alpha_t, alpha_w, alpha_c}))
            v_seq_dev = predict(observations, emissions, transition,
                alphas, nfeatures)
            precision, recall = compute_score(v_seq_dev, true_classes)
            f = f_score(precision, recall)

            -- Filling the scores tensor
            scores[{(i-1)*size2+k, 1}] = alpha_w
            scores[{(i-1)*size2+k, 2}] = alpha_c
            scores[{(i-1)*size2+k, 3}] = f
            scores[{(i-1)*size2+k, 4}] = precision
            scores[{(i-1)*size2+k, 5}] = recall
        end
    end

    return scores
```

```
end


function main()
    -- Parse input params
    opt = cmd:parse(arg)

    -- Reading file from pre-processing
    myFile = hdf5.open(opt.datafile,'r')
    data = myFile:all()
    emission_w = data['emission_w']
    emission_c = data['emission_c']
    -- Table of emission tensor (one tensor per feature)
    emissions = {emission_w, emission_c}
    -- Assertion on number of features
    nfeatures = opt.nfeatures
    if nfeatures > #emissions then
        error('Too many features specified')
    end
    print('Number of features used: '..nfeatures)
    transition = data['transition']
    input_matrix_train = data['input_matrix_train']
    input_matrix_dev = data['input_matrix_dev']
    input_matrix_test = data['input_matrix_test']
    myFile:close()

    -- Parameters:
    true_classes = input_matrix_dev:narrow(2,5,1):clone():view(input_matrix_dev:size(1))
    -- contain in each column feature observation
    -- (same order as the feature emission tensor in the emissoins table)
    observations = input_matrix_dev:narrow(2,3,nfeatures):clone()
    -- Alpha parameter
    alphas = torch.Tensor({opt.alpha_t, opt.alpha_w, opt.alpha_c})

    -- Prediction on dev
    v_seq_dev = predict(observations, emissions, transition, alphas, nfeatures)
    precision, recall = compute_score(v_seq_dev, true_classes)
    f = f_score(precision, recall)

    print('Prediction on dev')
    print('Precision is : '..precision)
    print('Recall is : '..recall)
```

```
    print('F score (beta = 1) is : '..f)

    -- Cross validation
    if (opt.cv == 1) then
        alphas_table = {}
        -- alpha_w
        alphas_table[1] = torch.Tensor({0.05, 0.1, 0.2, 0.3, 0.5, 0.8})
        -- alpha_c
        alphas_table[2] = torch.Tensor({5, 8, 10, 12, 15, 20, 22})

        scores = cross_validation(observations, emissions, transitions, true_classes,
                                  alphas_table, opt.alpha_t)
        print(scores)

        -- Saving the score
        myFile = hdf5.open('plot_scores.hdf5', 'w')
        myFile:write('scores', scores)
        myFile:close()
        print('CV on dev saved in '..'plot_scores.hdf5')
    end

    -- Prediction on test
    if (opt.test == 1) then
        print('Prediction on test')
        observations_test = input_matrix_test:narrow(2,3,nfeatures):clone()
        v_seq_test = predict(observations_test, emissions, transition, alphas, nfeatures)
        -- Saving predicted sequence on test
        myFile = hdf5.open(opt.datafile_test, 'w')
        myFile:write('v_seq_test', v_seq_test)
        myFile:write('v_seq_dev', v_seq_dev)
        myFile:close()
        print('Sequence predicted on test saved in '..opt.datafile_test)
    end

end

main()
```

Max-Entropy Markov Model:

```
require 'hdf5';
require 'nn';
```

```
require 'helper.lua';

-- Loading data
myFile = hdf5.open('../data/MM_data_pos.hdf5','r')
data = myFile:all()
input_matrix_train_pos = data['input_matrix_train_pos']
input_matrix_dev_pos = data['input_matrix_dev_pos']
input_matrix_test_pos = data['input_matrix_test_pos']
myFile:close()

nwords = input_matrix_train_pos:size(1)
train_output = input_matrix_train_pos:narrow(2,59)
train_input_pos = torch.Tensor(nwords-1,1+9+5+43)
train_input_pos:narrow(2,1,1):copy(input_matrix_train_pos:narrow(2,1,1):narrow(1,2,nwords-1))
train_input_pos:narrow(2,2,9):copy(input_matrix_train_pos:narrow(2,2,9):narrow(1,1,nwords-1))
train_input_pos:narrow(2,11,5):copy(input_matrix_train_pos:narrow(2,11,5):narrow(1,1,nwords-1))
train_input_pos:narrow(2,16,43):copy(input_matrix_train_pos:narrow(2,16,43):narrow(1,1,nwords-1))

observations_dev = input_matrix_dev_pos:narrow(2,1,1):clone()
dev_feat = input_matrix_dev_pos:narrow(2,11, 5 + 43)
dev_true_classes = input_matrix_dev_pos:narrow(2, 59,1):squeeze()

observations_test_pos = input_matrix_test_pos:narrow(2,1,1)
observations_test_feat = input_matrix_test_pos:narrow(2,2,5+43)

-- Defining the model

model = nn.Sequential()
t1_pos = nn.ParallelTable()

t1_pos_1 = nn.Sequential()
t1_pos_1:add(LT)
t1_pos_1:add(nn.View(-1,50))

t1_pos_2 = nn.Identity()

t1_pos:add(t1_pos_1)
t1_pos:add(t1_pos_2)

model:add(t1_pos)
model:add(nn.JoinTable(2))
```

```
model:add(nn.Linear(50 + 9 + 5 + 43,9))
model:add(nn.LogSoftMax())

-- Training function:


function train_model_cap(train_input, train_output, model, criterion, din, nclass, eta, nEpochs, batchSize)
    -- Train the model with a mini batch SGD
    -- standard parameters are
    -- nEpochs = 1
    -- batchSize = 32
    -- eta = 0.01
    local loss = torch.Tensor(nEpochs)

    -- To store the loss
    local av_L = 0

    -- Memory allocation
    local inputs_batch = torch.DoubleTensor(batchSize, din)
    local targets_batch = torch.DoubleTensor(batchSize)
    local outputs = torch.DoubleTensor(batchSize, nclass)
    local  df_do = torch.DoubleTensor(batchSize, nclass)

    for i = 1, nEpochs do
        -- timing the epoch
        timer = torch.Timer()
        av_L = 0

        -- mini batch loop
        for t = 1, train_input:size(1), batchSize do
            -- Mini batch data
            current_batch_size = math.min(batchSize,train_input:size(1)-t)

            inputs_batch:narrow(1,1,current_batch_size):copy(train_input:narrow(1,t,current_batch_size))

            targets_batch:narrow(1,1,current_batch_size):copy(train_output:narrow(1,t,current_batch_size))

            -- reset gradients
            model:zeroGradParameters()

```

```
            -- Forward pass (selection of inputs_batch in case the batch is not full, ie last batch)
            outputs:narrow(1,1,current_batch_size):copy(model:forward({inputs_batch:narrow(1,1,current_batch_size):narrow(2,1,1),
            inputs_batch:narrow(1,1,current_batch_size):narrow(2,2,din-1)}))
            -- Average loss computation
            f = criterion:forward(outputs:narrow(1,1,current_batch_size), targets_batch:narrow(1,1,current_batch_size))

            av_L = av_L +f

            -- Backward pass
            df_do:narrow(1,1,current_batch_size):copy(criterion:backward(outputs:narrow(1,1,current_batch_size), targets_batch:narrow(1,1,current_batch_size)))
            model:backward({inputs_batch:narrow(1,1,current_batch_size):narrow(2,1,1), inputs_batch:narrow(1,1,current_batch_size):narrow(2,2,din-1)},
            df_do:narrow(1,1,current_batch_size))

            model:updateParameters(eta)

        end

        print('Epoch '..i..': '..timer:time().real)
        loss[i] = av_L/math.floor(train_input:size(1)/batchSize)
        print('Average Loss: '.. loss[i])

    end

    return loss
end

-- Viterbi for MEMM:

-- Evaluates the matrix of scores for all possible tags for the previous word, using the word features at timestep i

function compute_logscore_extrafeat(observations, feat, i, model, C)
    local y = torch.zeros(C,C)
    local hot_1 = torch.zeros(C+feat:size(2))
    for j = 1, C do
        hot_1:zero()
```

```
            hot_1[j] = 1
            hot_1:narrow(1,10,feat:size(2)):copy(feat:narrow(1,i,1))
            y:narrow(1,j,1):copy(model:forward({observations[i]:view(1,1), hot_1:view(1,C+feat:size(2))}))
        end
        return y
end

-- Evaluates the highest scoring sequence:
function viterbi_extrafeat(observations, feat, compute_logscore, model, C)

        local y = torch.zeros(C,C)
        -- Formating tensors
        local initial = torch.zeros(C, 1)
        -- initial started with a start of sentence: <t>

        initial[{8,1}] = 1
        initial:log()

        -- number of classes
        local n = observations:size(1)
        local max_table = torch.Tensor(n, C)
        local backpointer_table = torch.Tensor(n, C)
        -- first timestep
        -- the initial most likely paths are the initial state distribution
        local maxes, backpointers = (initial + compute_logscore_extrafeat(observations, feat, 1, model, C)[8]):max(2)
        max_table[1] = maxes
        -- remaining timesteps ("forwarding" the maxes)
        for i=2,n do
            -- precompute edge scores

            y:copy(compute_logscore_extrafeat(observations, feat, i, model, C))
            scores = y:transpose(1,2) + maxes:view(1, C):expand(C, C)

            -- compute new maxes
            maxes, backpointers = scores:max(2)

            -- record
            max_table[i] = maxes
            backpointer_table[i] = backpointers
        end
        -- follow backpointers to recover max 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
end

-- Train Model

loss_pos = train_model_cap(train_input_pos, train_output,
    ultimate_t_pos, criterion, 1 + 9 + 5 + 43, 9, 0.1, 20, 32)

-- Evaluate performance on dev set:

cl_pos_dev = viterbi_extrafeat(observations_dev, dev_feat,
    compute_logscore_extrafeat, ultimate_t_pos, 9)
f = f_score(cl_pos_dev, dev_true_classes)

-- Predict on test:

v_seq_test_pos = viterbi_extrafeat(observations_test_pos,
    observations_test_feat, compute_logscore_extrafeat, ultimate_t_pos,
    9)

-- Saving predicted sequence on test
myFile = hdf5.open('../submission/v_seq_test_mem_pos', 'w')
myFile:write('v_seq_test', v_seq_test_pos)
myFile:close()
```

Helper:

```
-- function to evaluate the predicted sequence
-- need to compute precision and recall (class 1 stands for negative
    class)
function compute_score(predicted_classes, true_classes)
    local n = predicted_classes:size(1)
    local right_pred = 0
    local positive_true = 0
    local positive_pred = 0
    for i=1,n do
        if predicted_classes[i] > 1 then
            positive_pred = positive_pred + 1
        end
        if true_classes[i] > 1 then
```

```
            positive_true = positive_true + 1
        end
        if (true_classes[i] == predicted_classes[i]) and true_classes[i] > 1 then
            right_pred = right_pred + 1
        end
    end
    local precision = right_pred/positive_pred
    local recall = right_pred/positive_true
    return precision, recall
end

function f_score(predicted_classes, true_classes)
    local p,r = compute_score(predicted_classes, true_classes)
    print('Precision: '..p)
    print ('Recall: '..r)
    print ('f-score: '..2*p*r/(p+r))
    return 2*p*r/(p+r)
end
```