Exercise for MA-INF 2218 Video Analytics SS2
Weakly supervised action learning

Mayara E. Bonani
Ismail Wahdan

This is a report of the assignment 05 of the course Video Analytics.

Utils.py

- **get_transitions_prior(class2index)**

Before the model implementation, we implemented a function to read the Grammars in the file "utils.py" in the function get_transitions_prior(class2index). In this function we add a possible transition with a small weight to each possible transition in A. A usually has transitions between subaction itself and next subaction, with weight 0.8 meaning a substate tends to return itself, and 0.2 the possibility it moves on to the next subactions. Using the grammar, we observe which actions come after each other. For connection A1 -> A2, we connect the last subaction of action A1 to the first subaction of A2 with a small weight 0.2 and normalize each row, because we don't know how many connections there are after each action.

- **get_subaction_alignment(alignment)**

This function uniformly generates subactions over the alignment of actions. Example for 16 frames of two actions, each will get 4 subactions:
[0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1] -> [0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7]
Where subactions [0-3] belong to action 0 and subactions [4-7] belong to action 1 .

Moreover, the two functions **collate_fn_padd_training(batch)** and **collate_fn_padd_test(batch)** are used to allow dataloader to load batches of videos, since videos have variable number of frames. However, we decided to simply use a batch of 1 video, because the algorithms we are using are not designed for parallel processing.

Dataset.py

- **class Dataset(torch.utils.data.Dataset)**

The constructor of this class simply reads a list of path of the videos, as well as create maps **class2index** and **index2class** e.g SIL -> 0 and 0 -> SIL, to facilitate reading other files like grammar and ground truth.

- **__getitem__:** For the training dataset, the function returns both videos and their initial uniform alignment, as well the transcripts (ordered lists of actions in video) used in training. For testing, it just returns the video and the actual ground truth alignment.

Model.py:

The classes contained in the model.py will be explained together with the implementation provided for each task.

1. Train a weakly supervised system using HMMs+GMMs for five iterations.

The class HMM and GMM implement the Hidden Markov Model and the Gaussian Mixture Model respectively.

HMM:

For each action, there is a HMM instance. The HMM can be defined as a tuple of:

- pi: a vector of length n_states, probability of starting in a certain state
- A: probability of transition between two states
- An observation model (GMM, MLP, RNN) that will be trained in parallel and used to calculate b, the observation probability.

Initialization:

- **pi** is initialized uniformly over the first subaction of each action. It can also be implemented to depend on the grammar file and frequency of each action at the start.
- **A** is initialized as described above in **get_transitions_prior(class2index)**

We will now explain the functions inside the class HMM.

- **get_alpha(self, frames, b_i_t)**

The function has as input the frames (equivalent to O1, …,Ot in the formula below) and the observation probability calculated by GMM or the other helper model used (b_j(O_t)) function.

Alpha is a helper function used in the forward-backward algorithm.

Alpha is the conditional probability of seeing the sequence of observations up to time point t, and ending at state i at time t, given the current HMM) model.

$$\alpha_t(i) = P(O_1, O_2, O_3, \ldots, O_t, q_t = S_i \mid \lambda)$$

Inductively we have:

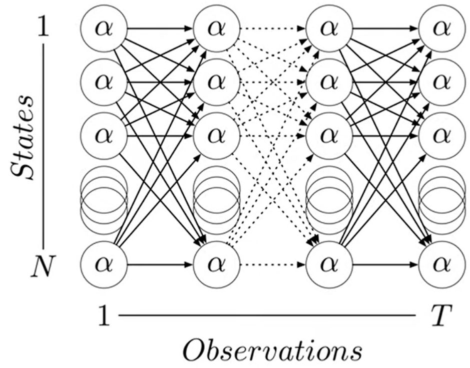
- base case: $\alpha_1(i) = \pi_i b_i(O_1) \quad 1 \leq i \leq N$
- inductive step:

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(O_{t+1}) \qquad \begin{array}{l} 1 \leq t \leq T-1 \\ 1 \leq j \leq N \end{array}$$

The final step is defined as follows:

$$P(O \mid \lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

The image below shows the graph that represents the HMM for N states and T observations



- **get_beta(self, frames, b_i_t)**

The function has as input the frames and the observation probability calculated by GMM or any other helper model, input described the same way as above.

The Beta is the probability of observing the next sequence of observations after t till last time point T, given we are currently at state i and the current HMM model.

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \cdots O_T \mid q_t = Si, \lambda)$$

Inductively we have:

- base case: $\beta_T(i) = 1 \qquad 1 \leq i \leq N$
- inductive step:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)$$

$$t = T-1, T-2, \cdots, 1 \quad 1 \leq i \leq N$$

- **get_gamma(self,frames , b_i_t)**

It implements the following formula and returns gamma:

$$\gamma_i(t) = p(Q_t = i|O, \lambda) \ = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$

- **get_eta(self,frames , b_i_t)**

It implements the following formula and returns eta:

$$\xi_{ij}(t) = \frac{p(Q_t = i, Q_{t+1} = j, O|\lambda)}{p(O|\lambda)}$$

$$= \frac{\alpha_i(t) a_{ij} b_j(o_{t+1}) \beta_j(t+1)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) a_{ij} b_j(o_{t+1}) \beta_j(t+1)}$$

- **update_pi(self)**

$$\pi_i = \frac{\sum_{e=1}^{E} \gamma_i^e(1)}{E}$$

- **update_A(self, frames)**

$$\tilde{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

- **learn(self, frames, b_i_t)**
  The learn function performs sequentially the following steps:
  - alpha_i_t = self.get_alpha(frames, b_i_t)
  - beta_i_t = self.get_beta(frames, b_i_t)
  - gamma_i_t = self.get_gamma(frames, b_i_t)
  - eta_i_j_t = self.get_eta(frames, b_i_t)
  - self.update_pi()
  - self.update_A(frames)

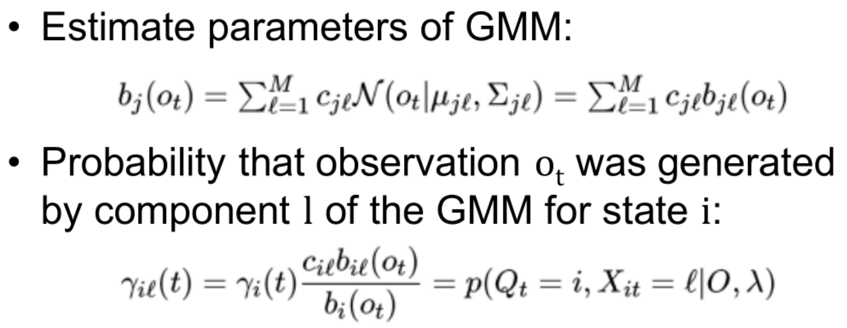
GMM:
Initialization:

- Mean: random frames
- Covariance Matrix: unit matrices

We will now explain the functions inside the class GMM.

- **train(self, alignment, frames, gamma)**

The training is based on the equations provided in the lecture:



- Estimate parameters of GMM:

$$b_j(o_t) = \sum_{\ell=1}^{M} c_{j\ell} \mathcal{N}(o_t|\mu_{j\ell}, \Sigma_{j\ell}) = \sum_{\ell=1}^{M} c_{j\ell} b_{j\ell}(o_t)$$

- Probability that observation o_t was generated by component l of the GMM for state i:

$$\gamma_{i\ell}(t) = \gamma_i(t) \frac{c_{i\ell} b_{i\ell}(o_t)}{b_i(o_t)} = p(Q_t = i, X_{it} = \ell|O, \lambda)$$

The input is list of alignment as ground truth of S list of frames as observations.
The gamma is calculated by Baum-Welch algorithm from HMM of shape [state_dim, T(#nframes)], as it is shown in the equation above.

- **get_b_i_t(self,frames)**

It returns a Tensor of shape [T, state_dim] where rows are normalised probability distribution (vector size state_dim) for observation over states resulting from the gaussian mixtures, simply put p(s|x)

- **get_b_j_l_t(self,frames)**

It returns a Tensor with shape (T,state_dim, M) with normalized rows, each representing predictions according only to component l number of Gaussian components for this state

2. Replace the GMM by a simple multi-layer perceptron (MLP) with one hidden layer of 64 rectified linear units and a softmax output layer with one unit for each HMM state.

In this case, we use the class SimpleMLP to provide the observational probabilities.
We have an MLP with 2 hidden layers. The input dimension is 64, the hidden dimension is 64, output dimension is 192. We use a ReLu function as activation function for the hidden layers and a softmax for the output. The model tries to learn p(s|x).

3. Replace the simple neural network by a recurrent network using gated recurrent units (GRUs). As input, the GRUs receive subsequences of 21 frames each (centered around the current frame).

We implemented the GRU in the class RNN. We used the Pytorch class torch.nn.GRU for its implementation, and for the fully connected layer, we use a ReLu as activation function. The model tries to learn p(s|x) (for x the current center frame)

4. Extra functions in main.py

- **get_new_alignment(model_output,hmm):**

using p(s|x) and the trained hmm, calculate the new alignment using equation 10 from paper:

$$\hat{s}(t) = \arg\max_{s(t)} \left\{ p(\mathbf{x}_1^T|\mathbf{a}_1^N) \right\} \tag{9}$$

$$= \arg\max_{s(t)} \left\{ \prod_{t=1}^{T} p\big(x_t|s(t)\big) \cdot p\big(s(t)|s(t-1)\big) \right\}. \tag{10}$$

where p(x|s) is calculated from p(s|x) in function **get_p_x_s(model_output,hmm)** using bayes:

$$p(x_t|s) = \mathrm{const} \cdot \frac{p(s|x_t)}{p(s)}.$$

- **Infer functions:**

The inference in paper is done for action lists not alignments, therefore we just use the same induction method from above for getting the alignment using p(s|x) and the HMM

- **Evaluate functions:**

Straight forward accuracy measurement.

Regarding Results:

The operation of training an HMM is highly sequential and can neither optimize parallelization techniques used in deep learning, nor use fast GPUs. The training has very large RAM and CPU needs that crash our machines when running. Moreover, while using loops is great for understanding, it is impractical for performance. We struggle with vectorization due to the complexity of the formulas, so we decided to keep the loops in the submission for the ease of understanding the code.

However, if we would have valid results we would expect that:

- The Model with GMM performs the worst, because GMMs are basically an unsupervised learning algorithm. They do not really utilize the alignments.
- MLP would perform better because it utilizes the alignments, and is able to generalize better over all videos.
- RNN would perform best, because it has a lot more context information, in addition to utilizing the alignments.

External Reference:

[1] https://www.youtube.com/watch?v=gYma8Gw38Os - Prof. Patterson lectures