Lab 8 : Hidden Markov Models

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

11/01/2022

Some general remarks

- In python, 1D arrays transposed are still 1D arrays. To compute $aa^\top$, you can use `np.outer`.
- Only vectorise operations after checking making sure it does what you think it does ! For loops are less error prove, although slower.
- Be careful with the indexing for the transition matrix, especially which N-1 terms they include.

Use cases

Hidden Markov Models are simply a discrete version of the LDS.
They are particularly suited in the case of discrete classes e.g. speech recognition, spelling correction...
The difference is that they allow for a bit more degrees of freedom regarding to the observation matrices.

Three basic problems for HMMs

1. Computing probabilities
   Given an observation sequence $x = x_1 \dots x_T$ and a model $\theta = (A, B, \pi)$, how to compute $P(x|\theta)$?
2. Decoding the latent states
   Given an observation sequence $x = x_1, \dots, x_T$ and a model $\theta = (A, B, \pi)$, how to compute the state sequence $Q = q_1, \dots, q_T$ that best explains the observations ?
3. Parameter estimation
   How to adjust the model parameters $\theta = (A, B, \pi)$ to maximize $P(x|\theta)$?

Computing a probability

Given θ and x, calculate $P(x \mid \theta)$.
We can write :

$$p(x, z) = p(z_0) \prod_{i=0}^{T-1} p(z_{i+1}|z_i) \prod_{i=1}^{T} p(x_i|z_i)$$

$$= \prod_{k=1}^{n} \pi_k^{z_{0,k}} \prod_{i=0}^{T-1} \prod_{j,k=1}^{n} A_{jk}^{z_{i+1,k} z_{i,j}} \prod_{i=1}^{T} \prod_{k=1}^{n} p(x_i|\phi_k)^{z_{i,k}}$$

$$P(x \mid \theta) = \sum_z P(x, z \mid \theta)$$

which is totally impractical.

A more efficient way to compute a probability

The order of the nodes does not matter, we only need to keep track of the probabilities of being in each state at each given time, then factor in the probability of observing the data point. Defining

$$\alpha(z_i) := P\left(x_{1:i}, z_i \mid \theta\right),$$

we have (everything is still conditioned on the parameters) :

$$\begin{aligned}
\alpha(z_i) &= \sum_{z_{i-1}} P(x_{1:i}, z_i, z_{i-1}) \\
&= \sum_{z_{i-1}} P(z_{i-1}) P(x_{1:i-1}|z_{i-1}) P(z_i|x_{1:i-1}, z_{i-1}) P(x_i|x_{1:i-1}, z_i) \\
&= \sum_{z_{i-1}} P(z_{i-1}) P(x_{1:i-1}|z_{i-1}) P(z_i|z_{i-1}) P(x_i|z_i) \\
&= P(x_i|z_i) \sum_{z_{i-1}} \alpha(z_{i-1}) P(z_i|z_{i-1})
\end{aligned}$$

A more efficient way to compute a probability

Finally,

$$\alpha(z_i) = P(x_1|z_1)P(z_1) = \prod_k (\pi_k P(x_1|\phi_k))^{z_{1k}}$$

$$\alpha(z_i) = P(x_i|z_i) \sum_{z_{i-1}} \alpha(z_{i-1}) P(z_i|z_{i-1})$$

$$P(x_{1:T}|\theta) = \sum_{z_T} \alpha(z_T).$$

Decoding : Viterbi algorithm

Given an observation sequence $x = x_1, \ldots, x_T$ and a model $\theta = (A, B, \pi)$, how to compute the state sequence $z = z_1, \ldots, z_T$ that best explains the observations ?

- A greedy decoding won't work; we want the sequence of states that maximize the likelihood of the data.
- As before, we don't care about how we arrived at a node. We just need their probability of generating the data sequence.
- We can again use dynamic programming, excepted that this time we use `max`.

Viterbi example

Figure 1: From link

Viterbi example

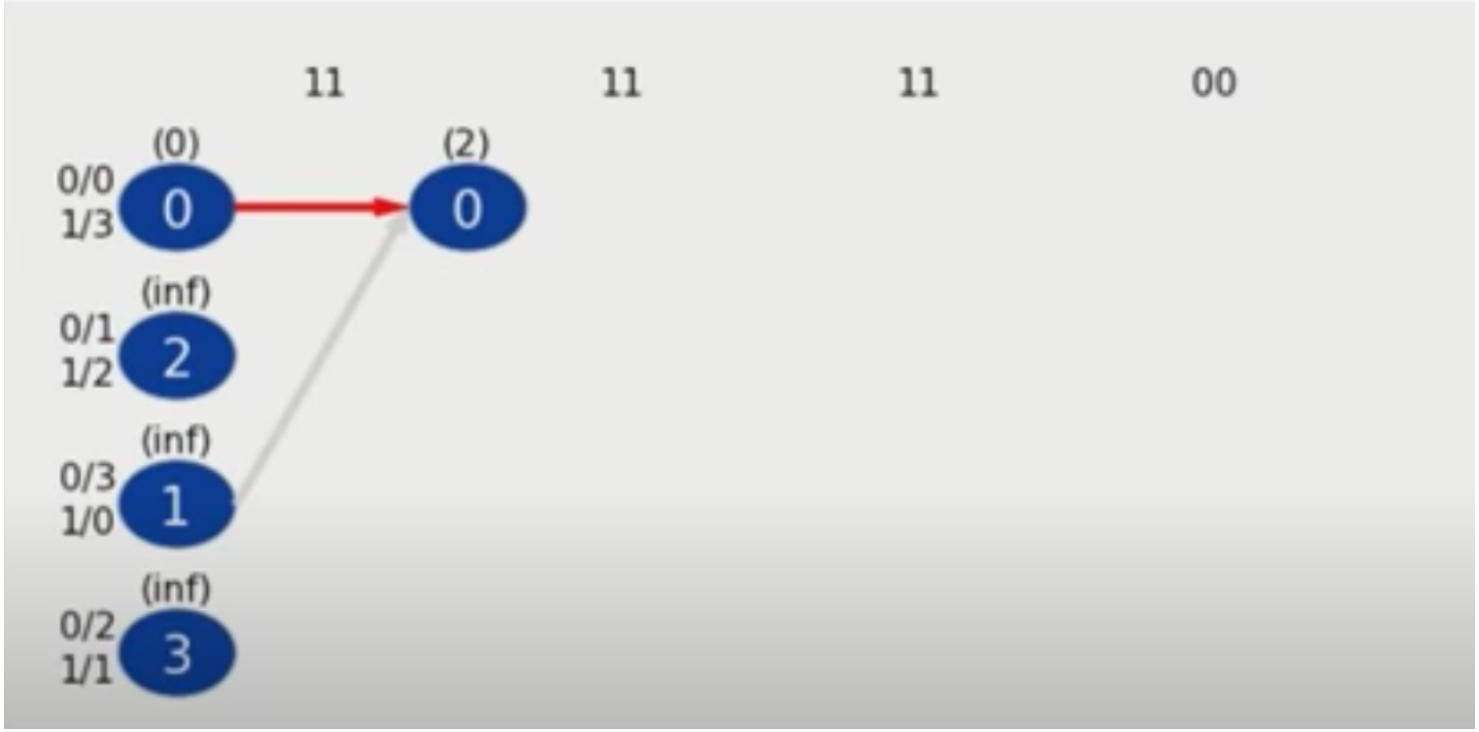

Figure 1: From link

Viterbi example

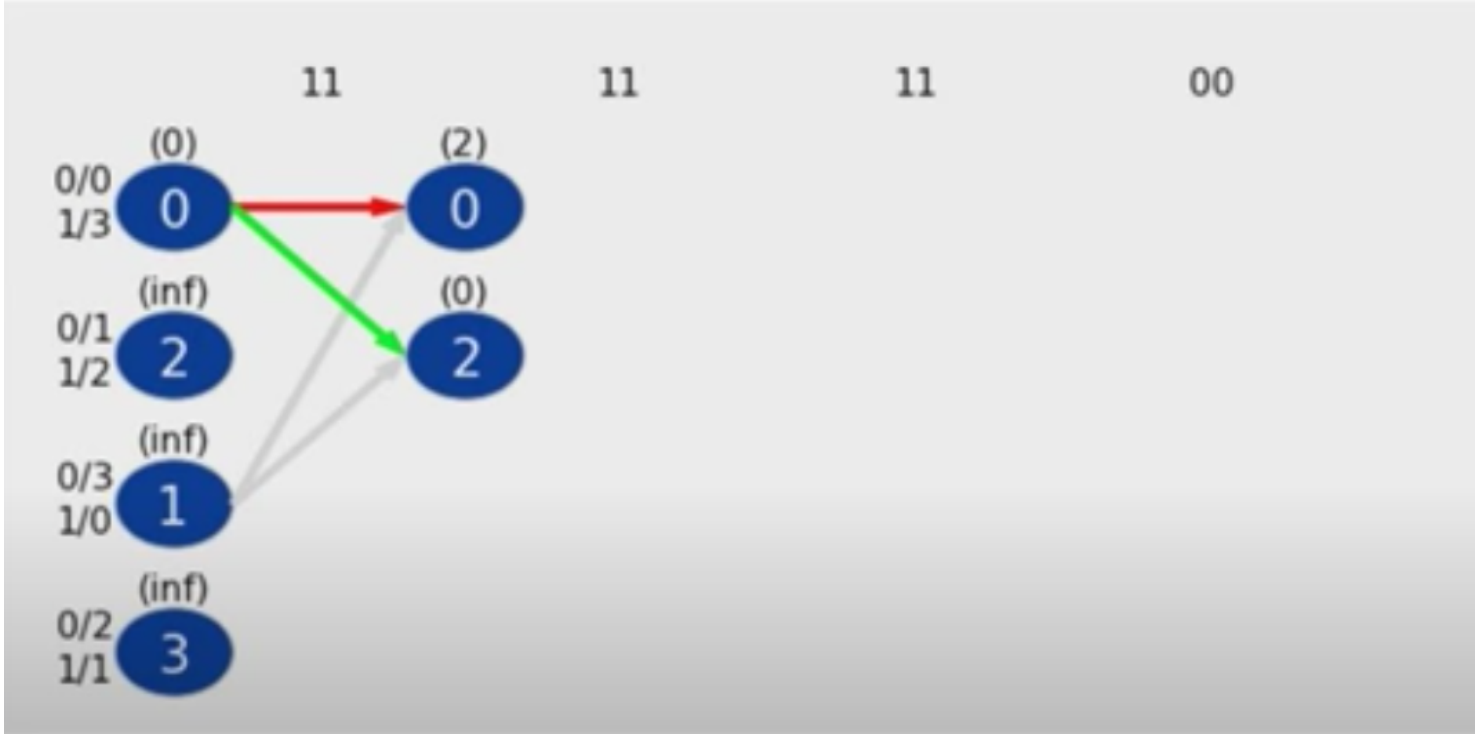

Figure 1: From link

Viterbi example

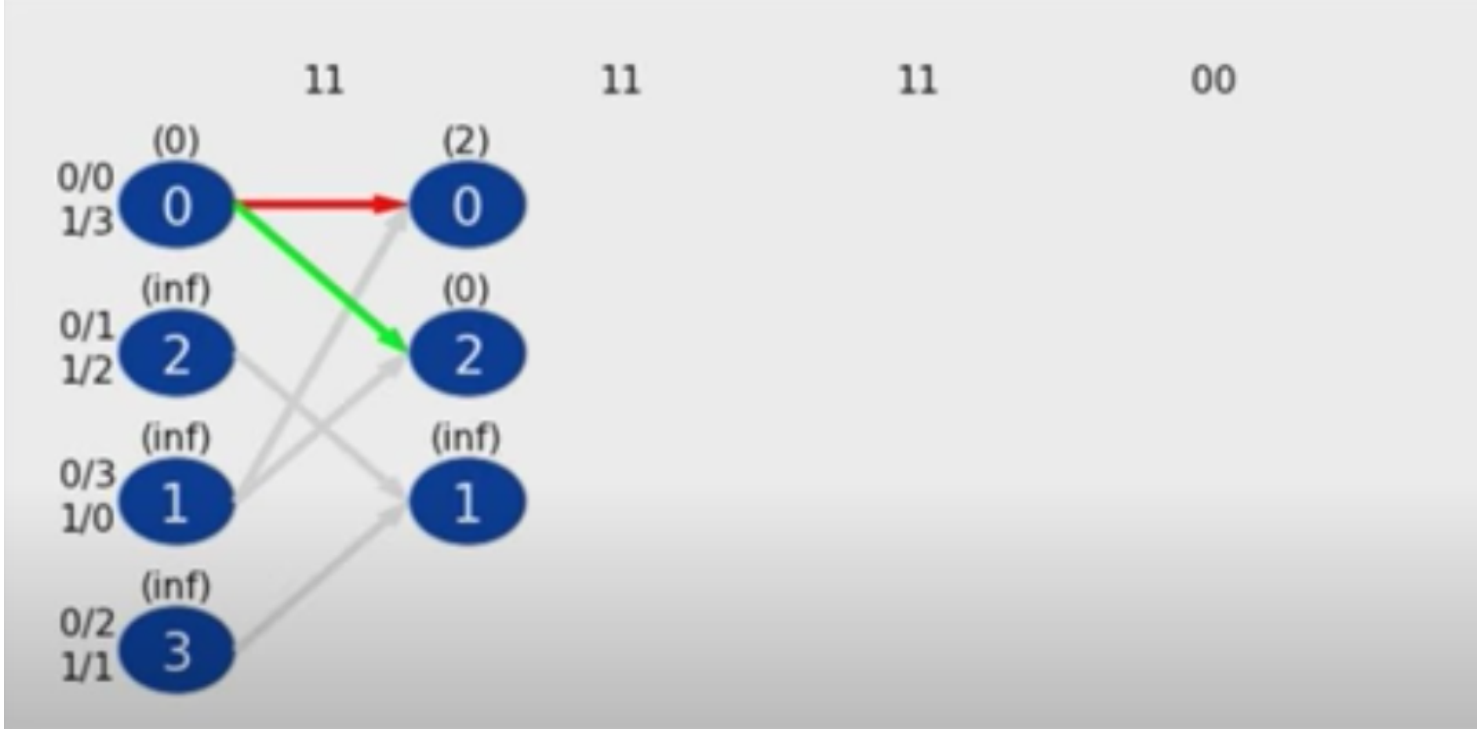

Figure 1: From link

Viterbi example

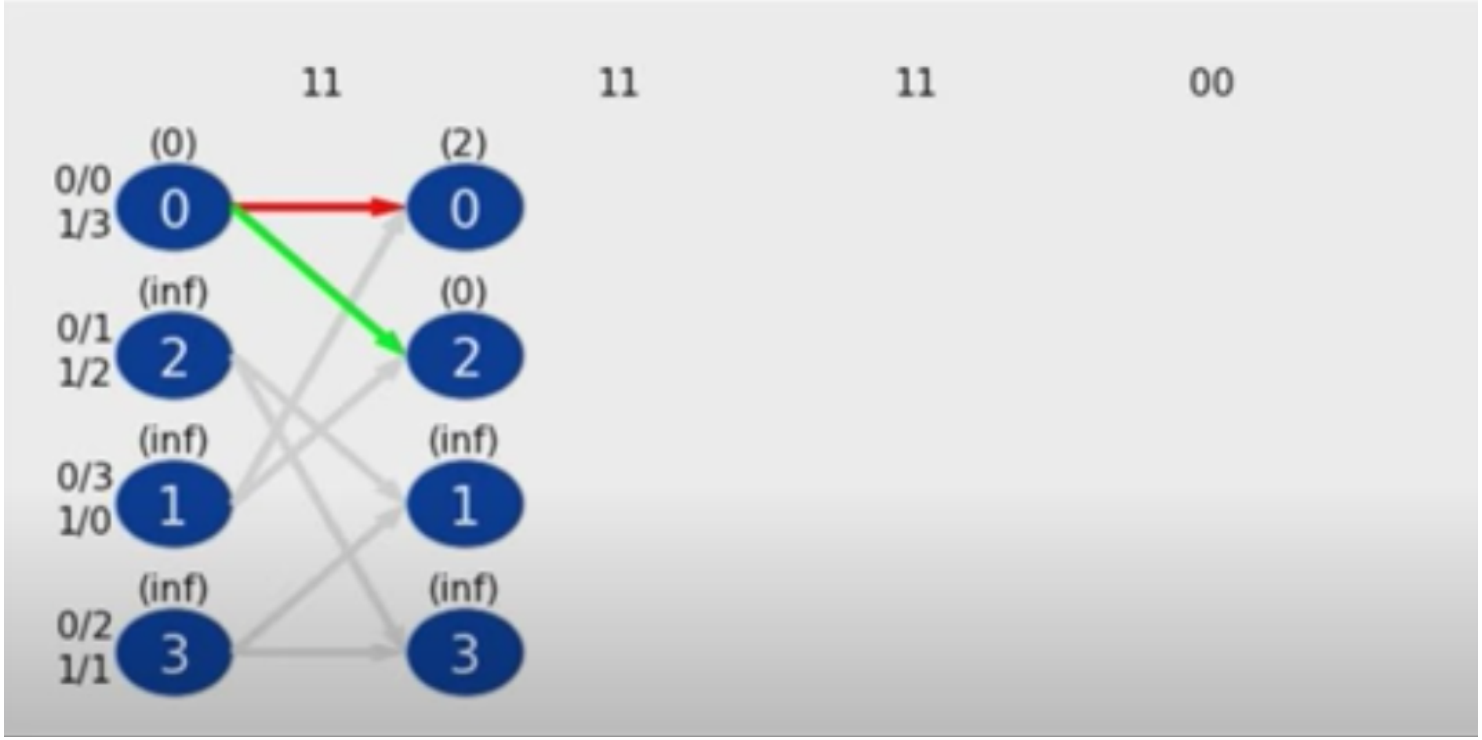

Figure 1: From link

Viterbi example

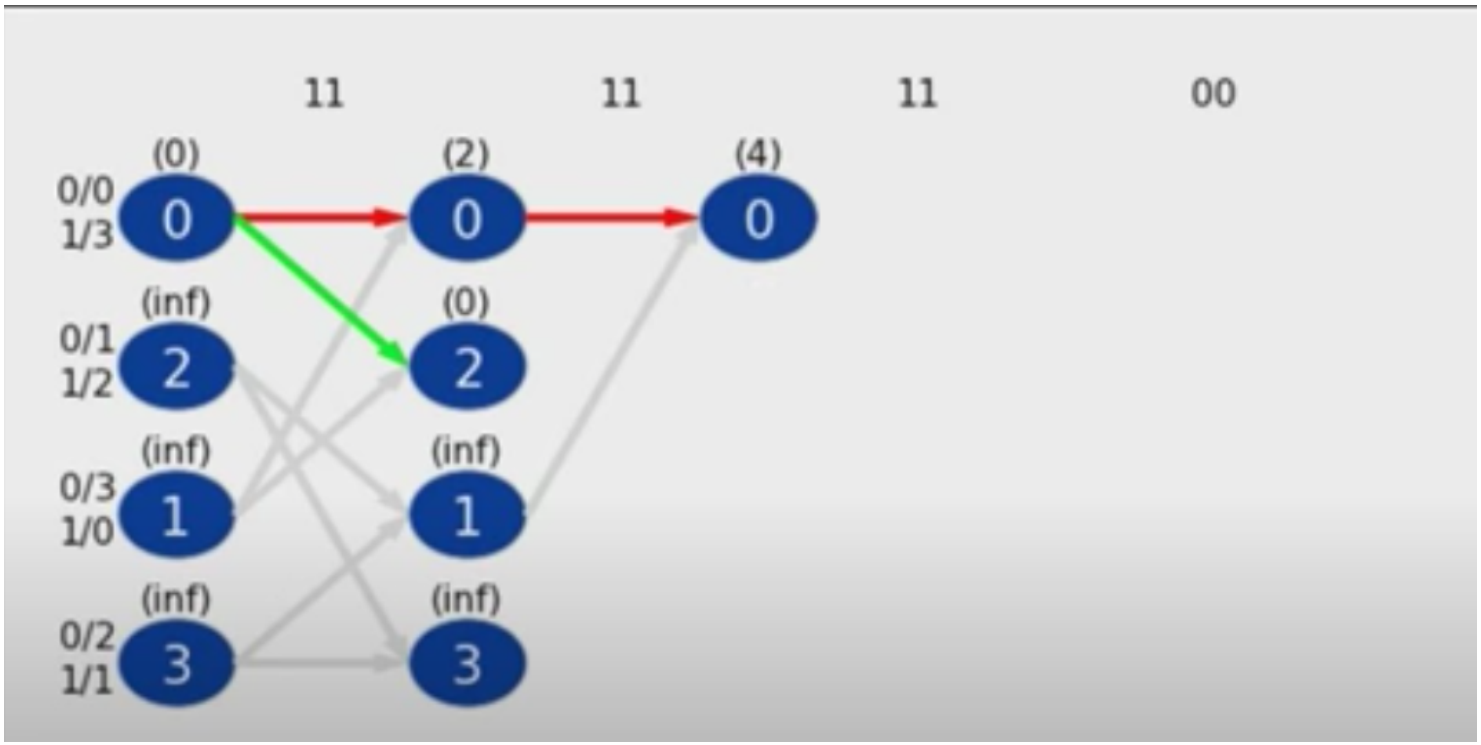

Figure 1: From link

Viterbi example

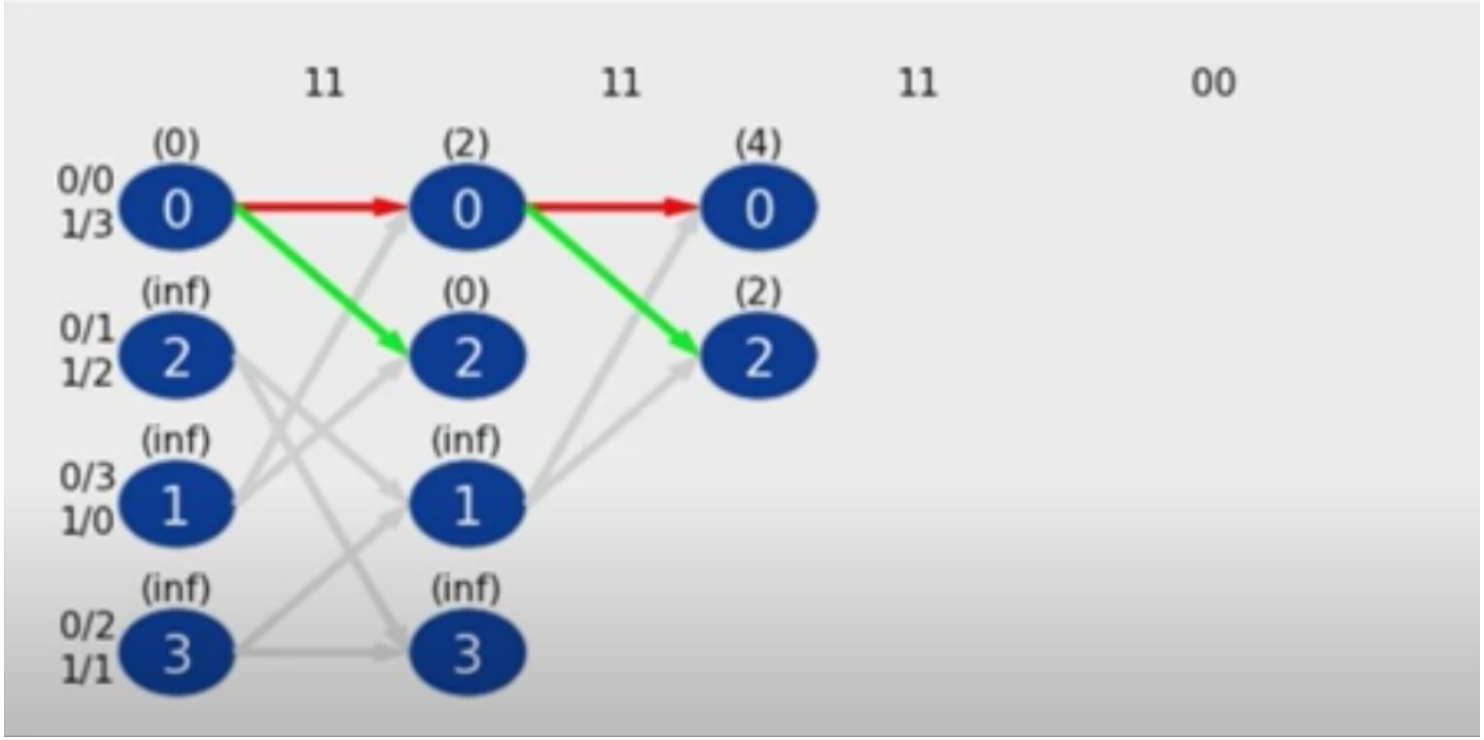

Figure 1: From link

Viterbi example

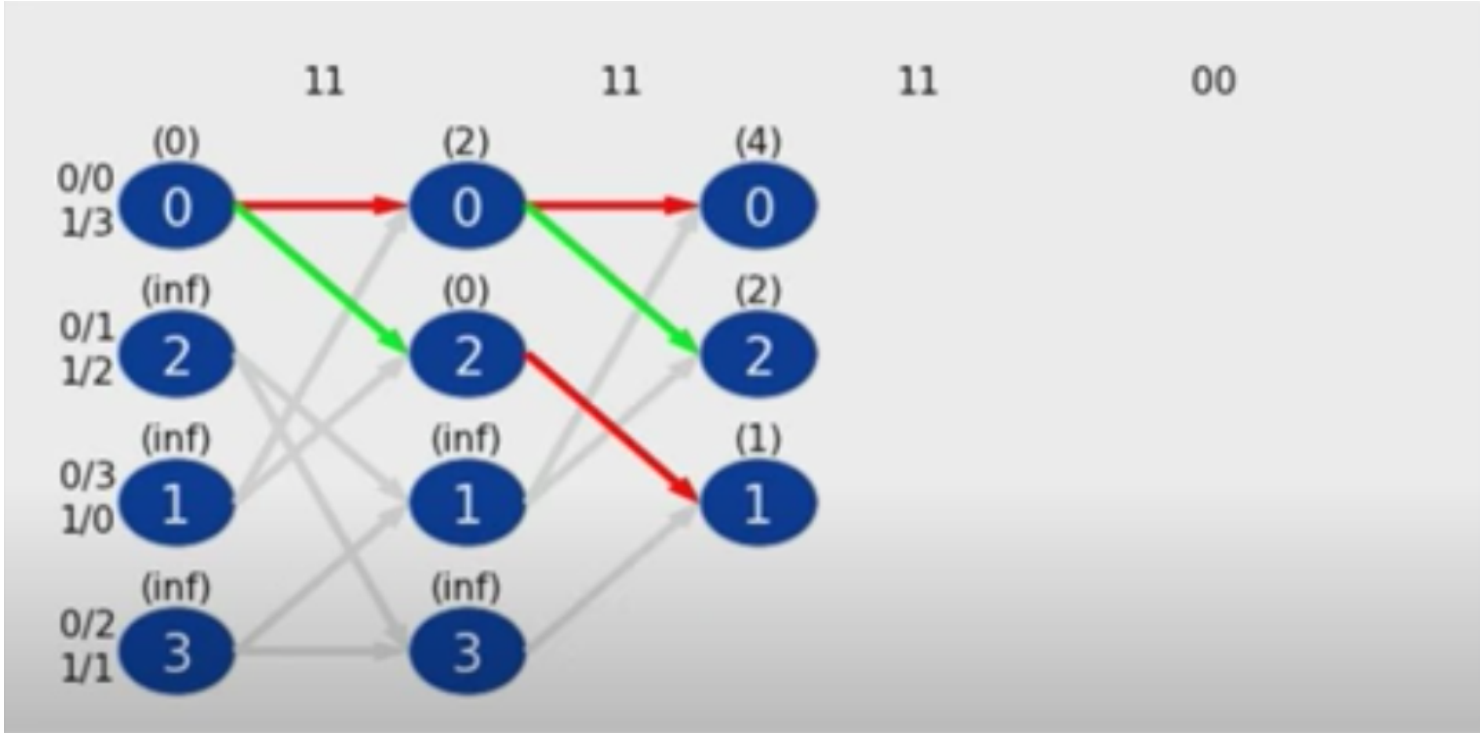

Figure 1: From link

Viterbi example

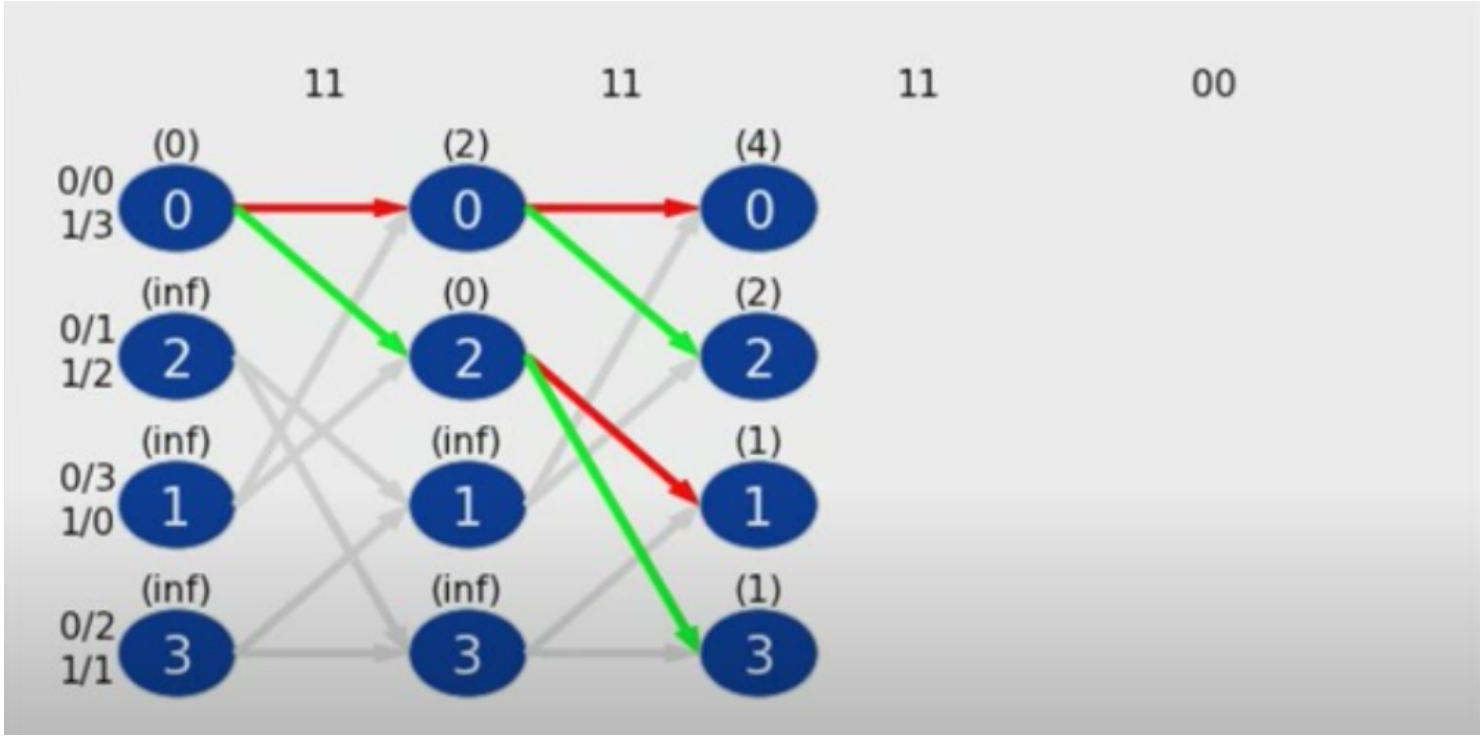

Figure 1: From link

Viterbi example

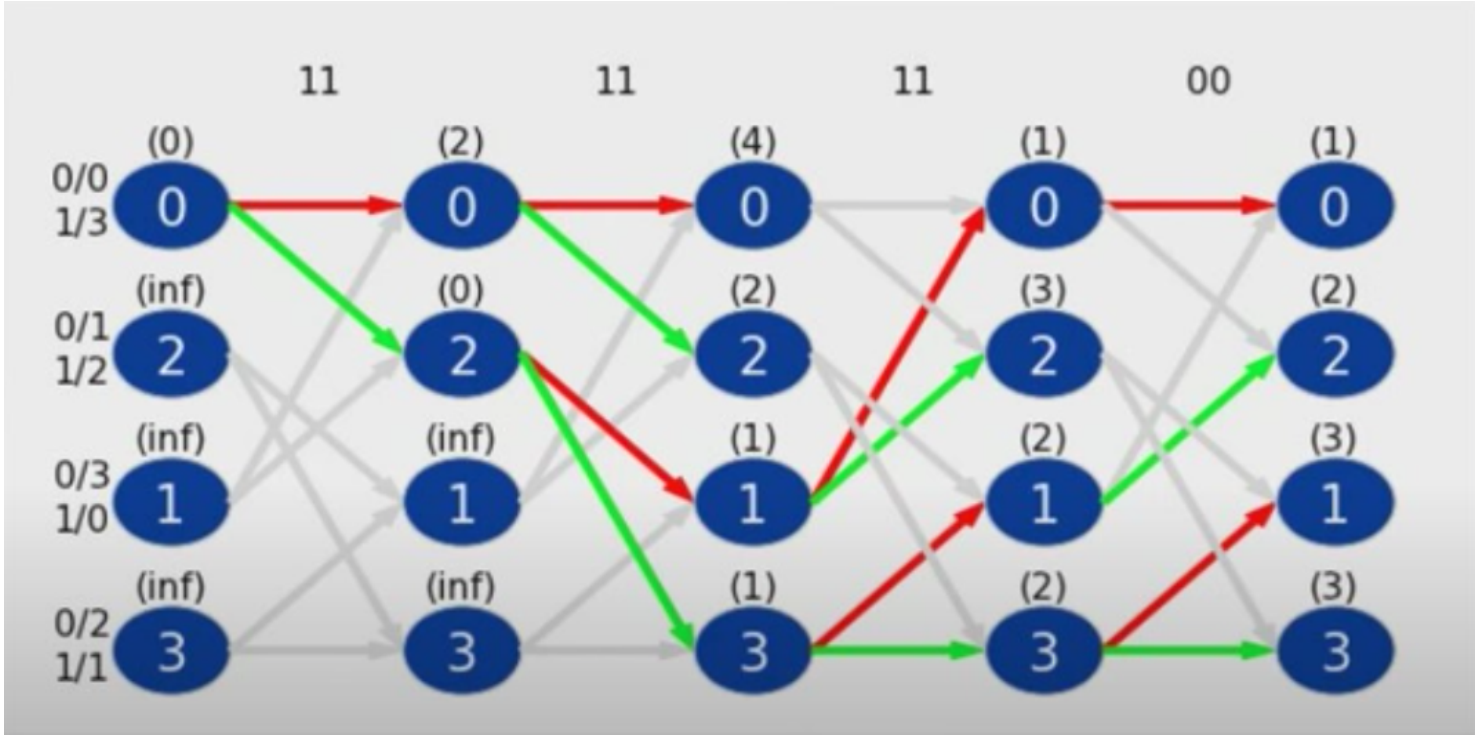

Figure 1: From link

Viterbi example

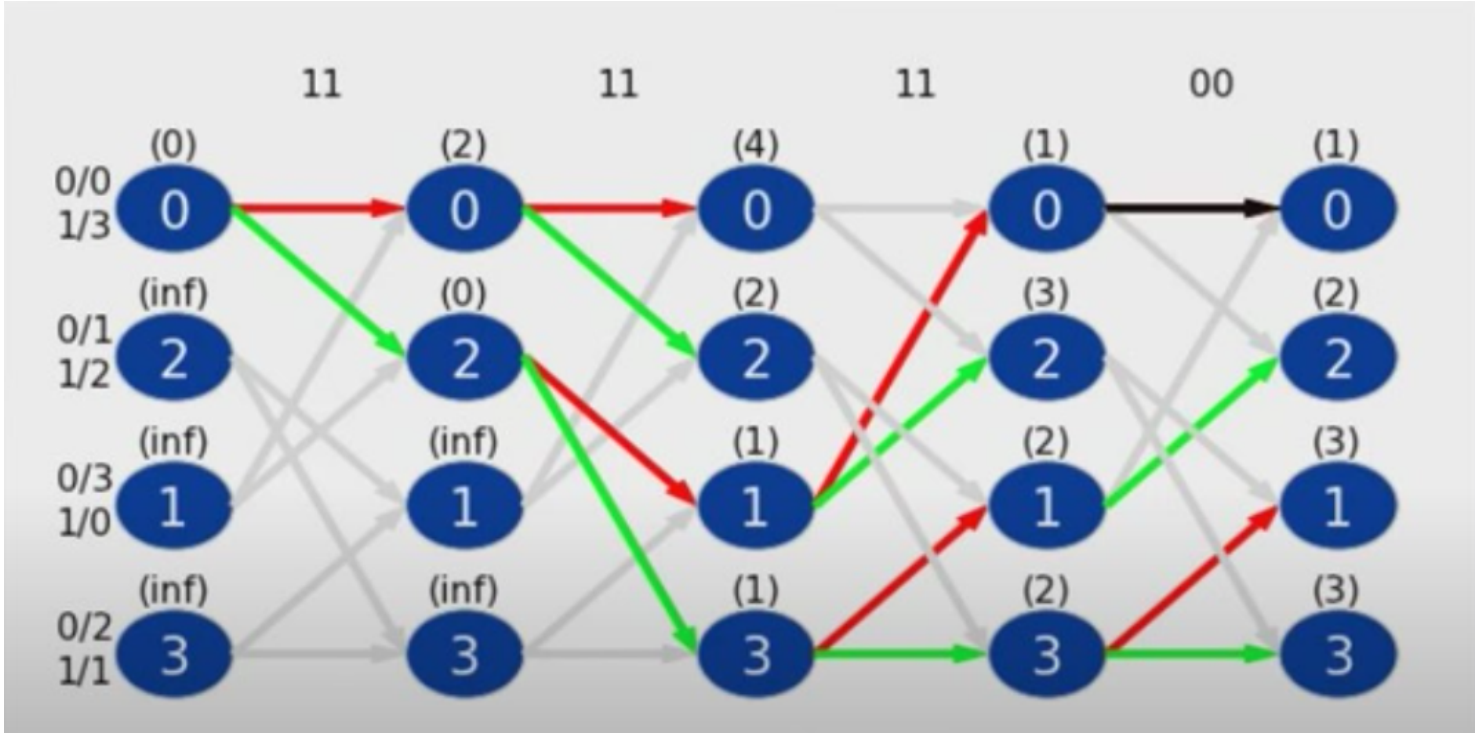

Figure 1: From link

Viterbi example

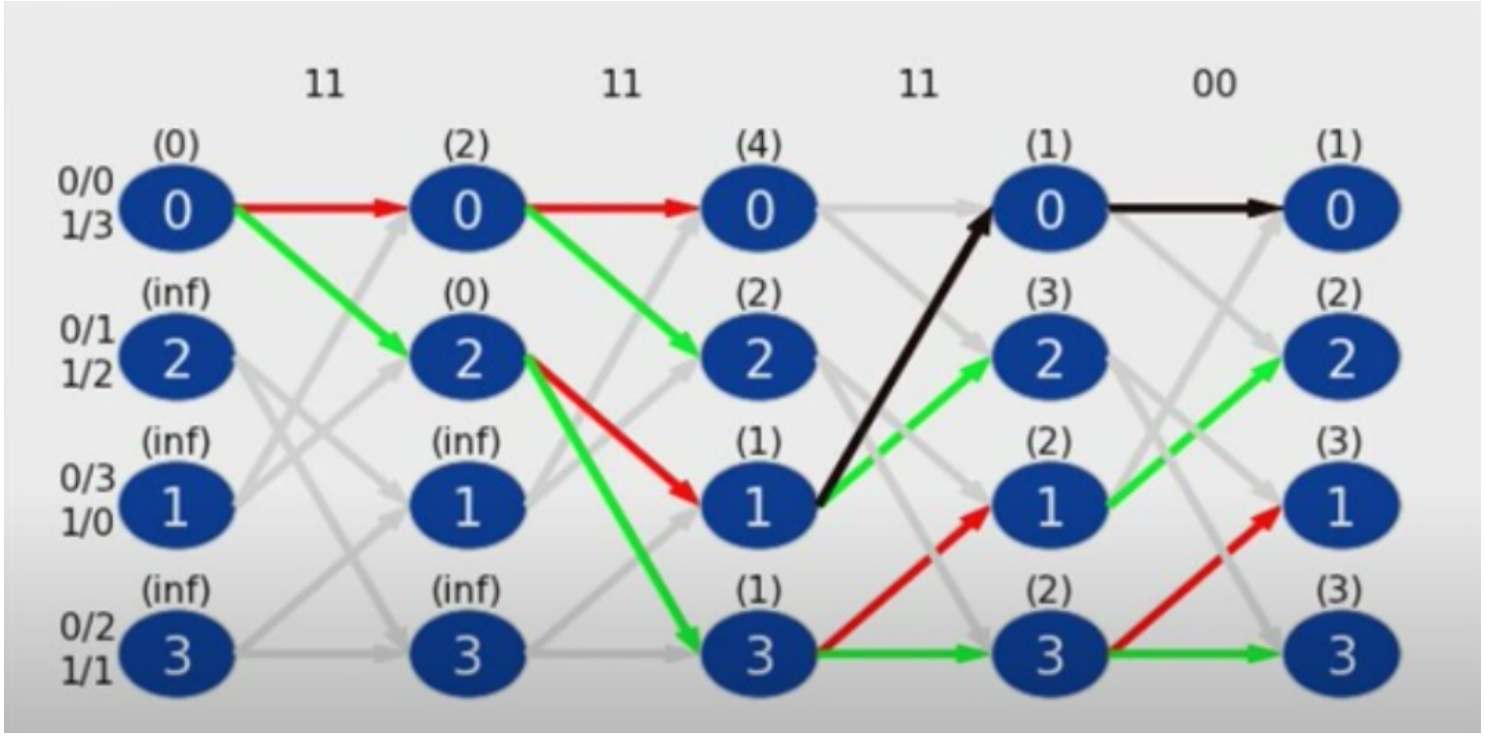

Figure 1: From link

Viterbi example

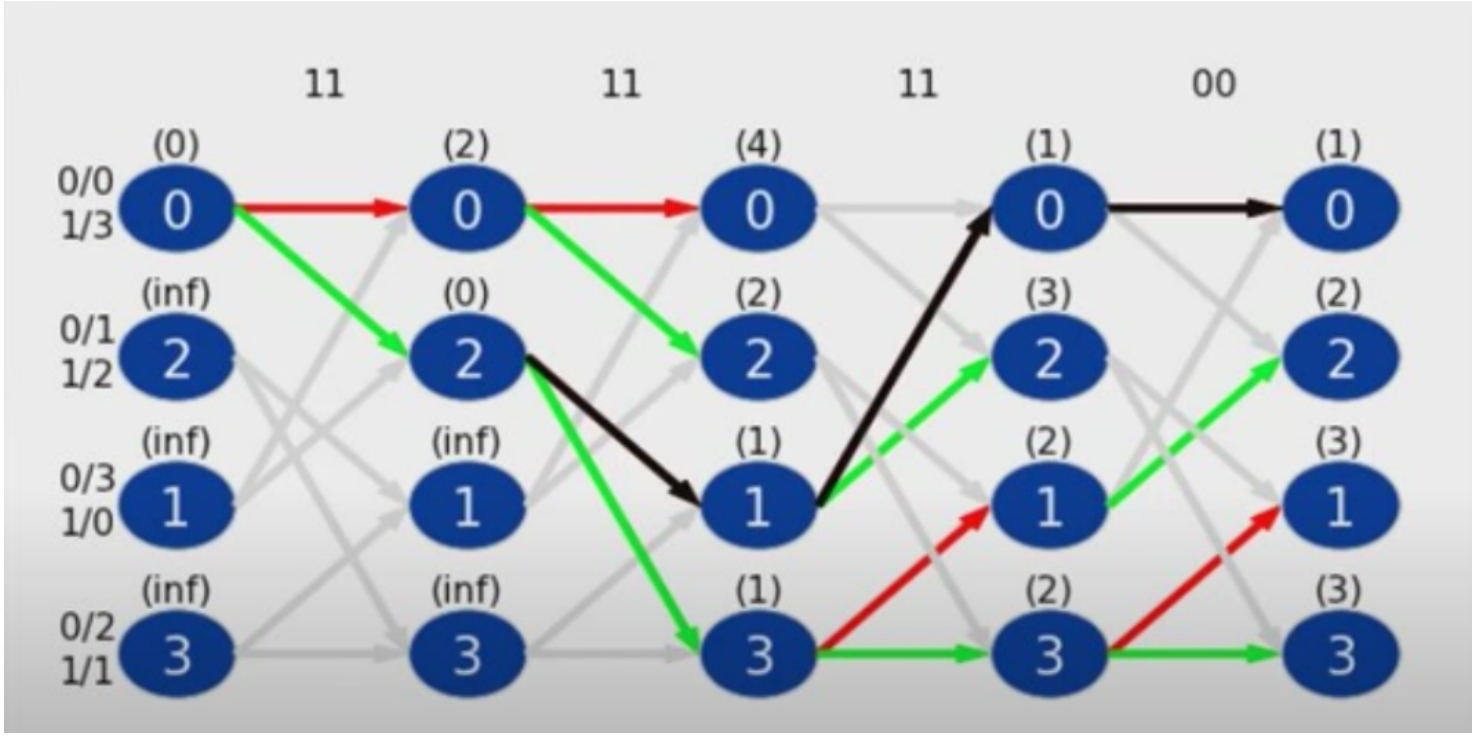

Figure 1: From link

Viterbi example

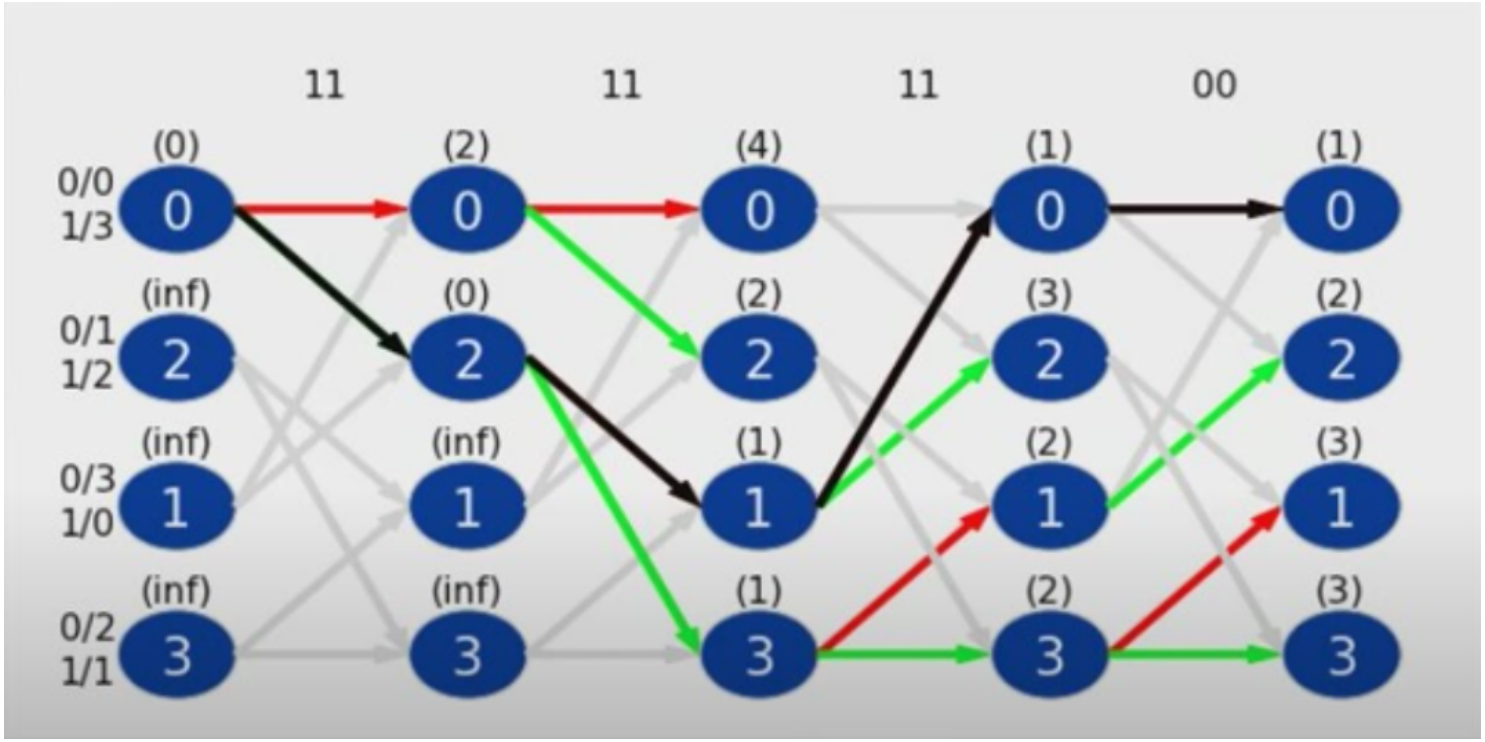

Figure 1: From link

Parameters inference

Similarly as for Kalman filtering, there is no exact inference, but the parameters can be estimated using EM.

- The E step is the same : getting the KL divergence, we have

$$q_k = p(.|x_{1:T}, \theta_{k-1}). \tag{1}$$

- For the M step, we write out the joint distribution. Then we just need to optimise the expectation. The only difference is that this time, we have a constrained problem :

$$\sum_k \pi_k = 1$$

$$\forall j \in [\![1, n]\!], \qquad \sum_k A_{jk} = 1.$$

HMM implementation challenges

- Numerical stability may be an issue with HMMs, because of taking the product of many small terms. To alleviate this issue, we use the rescaled values of α and β, along with log-probabilities.
- Initial estimates
  In general, uniform estimates for the transition matrix and the initial state probability vector are adequate (and sometimes adding some noise to break the symmetry). However for the estimation parameters, good estimates are helpful in the discrete case, and almost necessary for the continuous case.

HMM implementation challenges

- Insufficient training data
  Sometimes, the training data is insufficient to give good estimates of parameters. Although it is possible to reduce the size of the model, there are often physical reasons underneath. Another possibility is to interpolate parameters by fitting a smaller model (some parameters are tied to the initial model), where the data would be sufficient. Then,

$$\theta_{\text{interp}} = \varepsilon\theta + (1 - \varepsilon)\theta'.$$

  The optimal value of ε can be estimated using the forward-backward algorithm.

Data

data description

data are excerpts from the wall street journal

observations are words so that sentences form time series, the observation model is multinomial

we want to extract a latent space that characterizes the grammatical structure of sentences, the dataset is already labeled (for each observed word we are given the latent state which is the grammatical category the word belongs to)

more about the meaning of the latent space is here: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

we have the following variables ¶

'train_X' = a list of lists, each list is a sentence with numbers representing words (observations)

'word_indexer' and 'vocab_lookup go from words to numbers, length 20001

'label_indexer' and'label_lookup' go from category label to numbers, length 44

We will fit a HMM where each observed word is attributed to a category ("current state").

Example

```
ii = 7
print("example training sentence: \n\n", reconstruct_sequence(train_X[ii], vocab_lookup))
print("example's categories: \n\n", reconstruct_sequence(train_z[7], label_lookup))
```

```
example training sentence:

 ['a', 'record', 'date', 'has', "n't", 'been', 'set', '.']
example's categories:

 ['DT', 'NN', 'NN', 'VBZ', 'RB', 'VBN', 'VBN', '.']
```

```
print(len(train_X), ' training sentences')
print(len(test_X), 'testing sentences')
```

```
39815  training sentences
1700 testing sentences
```

Hidden Markov Model (HMM)

Hidden Markov Model

observed states $X = \{x_1, x_2, \ldots, x_N\}$ and $x_n = h \in \{h = 1, \ldots, H\}$

observed variable given the latent could be for instance gaussian distributed where the latent state defines which gaussian they are from - in the case of our dataset above the observed variables are multinomial

latent states $Z = \{z_1, z_2, \ldots, z_N\}$ and $z_n \in \{1, \ldots, K\}$, so a discrete multinomial variable

Transition probability matrix: $A \in R^{KxK}$ where $A_{i,j} = p(z_n = j | z_{n-1} = i)$ and rows sum to 1

Emission probability matrix: $C \in R^{KxH}$ where $C_{i,h} = C_i(x_n = h) = p(x_n = h | z_n = i)$ and rows sum to 1

Initial State Probability: $\pi \in R^K$ where we will set the initial state to one specified category so that $\pi_i = p(z_1 = i)$

special cases of HMM models through constraint on transition probability matrix A

- slow changes and fast changes
- directional, eg left-to-right HMM
- assume latent states have a natural order and make large jumps unlikely

any entry of A that is set to 0 in the initialization will stay 0 during the EM updates

Likelihood

Likelihood (the forward algorithm)

given parameters $\theta = \{A, C, \pi\}$ and observation sequence X find the likelihood $p(X|\theta)$

compute the probability of being in a state j after seeing the first n observations, denote that probability as $\alpha_n(j)$

$$\alpha_n(j) = P(x_1, \ldots, x_n, z_n = j) = P(x_n|z_n = j) \sum_{i=1}^{K} \alpha_{n-1} P(z_n = j|z_{n-1} = i) =$$
$$C_j(x_n) \sum_{i=1}^{K} \alpha_{n-1}(i) A_{ij}$$

with initial condition

$$\alpha_1(i) = P(x_1|z_1 = i)P(z_1 = i) = \pi_i C_i(x_1)$$

so that

$$P(X|\theta) = \sum_{i=1}^{K} \alpha_N(i)$$

implementation caveat: introduce scaling to avoid numerical problems due to small probability values (see lecture or Bishop)

there is the equivalent for the other direction, called the Backward Algorithm, that uses

$$\beta_n(j) = p(x_{n+1}, \ldots x_N|z_n = j)$$

Viterbi

Viterbi Algorithm (max sum algorithm)

Aim: inference - estimate the latent state sequence, assume parameters are given

There are exponentially many possible chains of latent spaces in a sequence. However, let's assume we are given the probabilities up to n we can infer the probabilities for the next time step $n+1$. If we work our way forward recursively in this way, we can dramatically decrease the computational cost. Essentially, we only consider the path with the highest probability at every time point for each state.

$v_n(j)$ is the probability that we are in state j after seeing the first n observations and passing through the most probable latent state sequence

- initialization:

$$v_1(j) = \pi_j C_j(x_1) \tag{1}$$

for every time point n and every possible latent state $j = 1, \ldots, K$

- update:

$$v_n(j) = C_j(x_n) max_{i=1}^{K} v_{n-1}(i) A_{ij} \tag{2}$$

- store best state

$$b_n(j) = argmax_{i=1}^{K} v_{n-1}(i) A_{ij} C_j(x_n) = argmax_{i=1}^{K} v_n(j) \tag{3}$$

Once this is done for the whole observation sequence, backtrack from the last time point (N) to find the estimate for the latent:

$$z_n = b_n(z_{n+1})$$

Where the process is initialized at the end with:

$$z_N = argmax_{i=1}^{K} n_{N-1}(i)$$

for implementation, make sure you use the log form instead (take the logarithm of the right hand side of equation (2))

Learning (EM)

parameter learning

in our dataset the latent states Z are given (E step of EM already solved), therefore we can update the parameters simply as follows:

$$\hat{A}_{ij} = \frac{num\ state\ transitions\ from\ i\ to\ j}{num\ state\ transitions\ from\ i}$$

$$\hat{C}_{ih} = \frac{num\ of\ times\ state\ i\ emits\ h}{num\ state\ i}$$

$$\hat{\pi}_i = \frac{num\ of\ chains\ start\ with\ i}{total\ num\ of\ chains}$$

Part 1 of lab (Viterbi)

```
def decode_single_chain(self, x):
    """
    Auxiliary method that uses Viterbi on single chain
    @param X: array-like with dimension [ # of length]
    @return z: array-like with dimension [# of length]
    """
    # init holders
    z = []
    V = np.zeros( (len(x), self.num_unique_states) )
    best_states = np.zeros( (len(x), self.num_unique_states) )

    ##########################################
    # TODO: implement the Viterbi algorithm #
    ##########################################

    return np.zeros(len(x))
```

Part 2 of lab (sampling)

```python
def sample(self, n_step, initial_state, seed=0):
    """
    Method that given initial state and produces n_step states and observations
    @param n_step: integer
    @param initial_state: an integer indicating the state
    """
    states = []
    observations = []
    current_state = initial_state

    np.random.seed(seed)

    ################################
    # TODO: sample from the model #
    ################################

    return states, observations
```