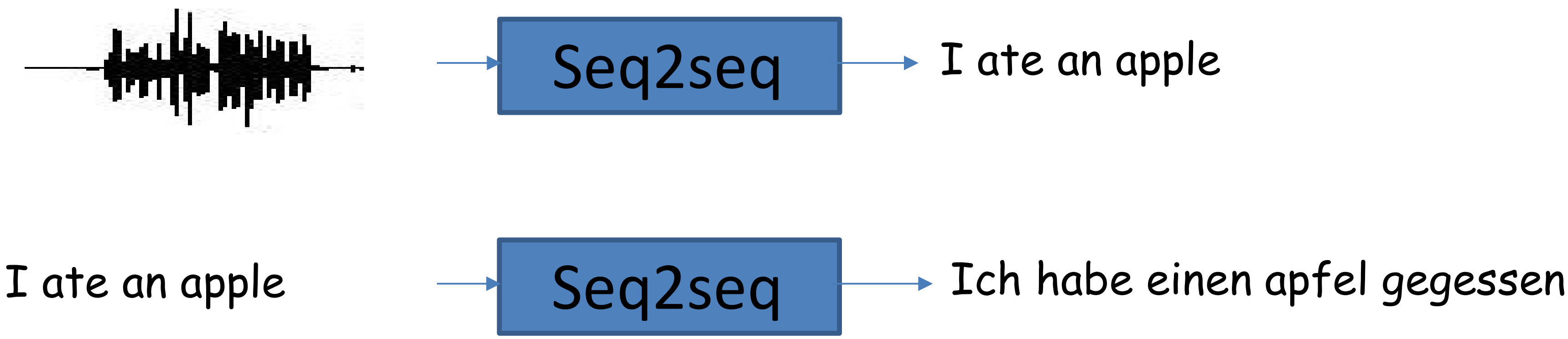
Deep Learning
Sequence to Sequence models: Connectionist Temporal Classification

Sequence-to-sequence modelling

- Problem:
  - A sequence $X_1 \dots X_N$ goes in
  - A different sequence $Y_1 \dots Y_M$ comes out
- E.g.
  - Speech recognition: Speech goes in, a word sequence comes out
    - Alternately output may be phoneme or character sequence
  - Machine translation: Word sequence goes in, word sequence comes out
  - Dialog : User statement goes in, system response comes out
  - Question answering : Question comes in, answer goes out
- In general $N \neq M$
  - No synchrony between X and Y.

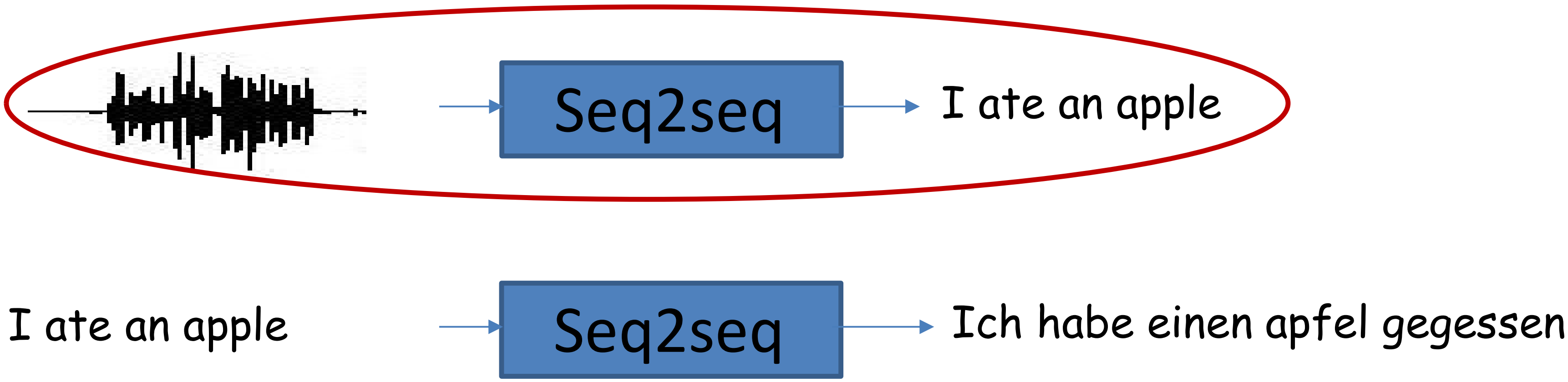
Sequence to sequence



- Sequence goes in, sequence comes out
- No notion of "time synchrony" between input and output
  - May even not even maintain order of symbols
    - E.g. "I ate an apple" → "Ich habe einen apfel gegessen"

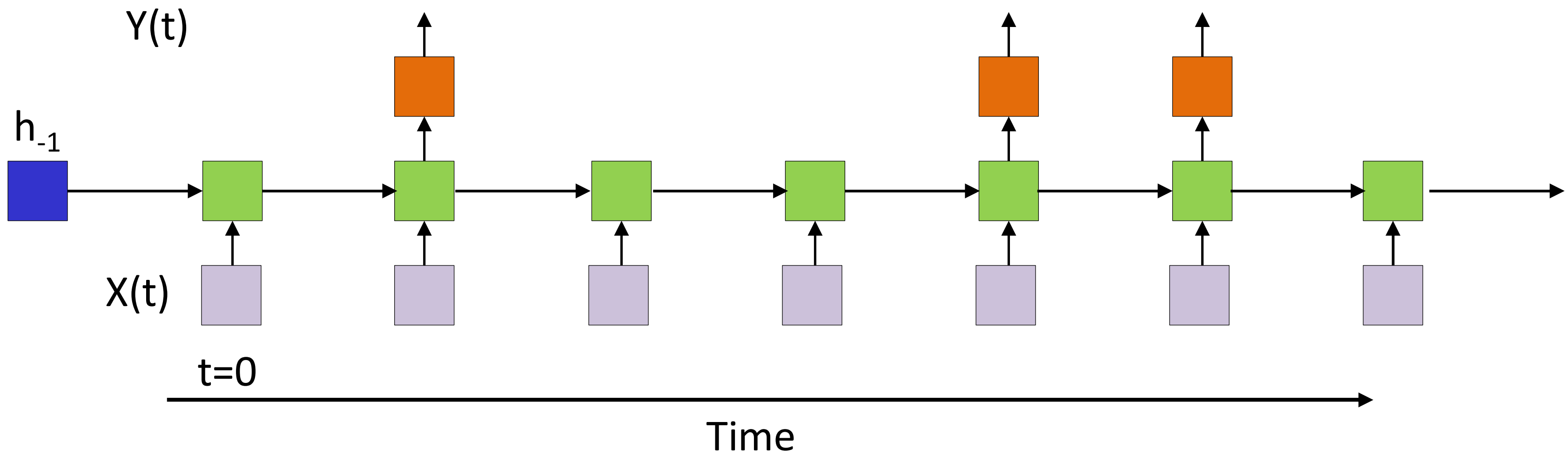
  - Or even seem related to the input
    - E.g. "My screen is blank" → "Please check if your computer is plugged in."

Sequence to sequence



- Sequence goes in, sequence comes out
- No notion of "time synchrony" between input and output
  - May even not even maintain order of symbols
    - E.g. "I ate an apple" → "Ich habe einen apfel gegessen"
  - Or even seem related to the input
    - E.g. "My screen is blank" → "Can you check if your computer is plugged in?"

Case 1: Order-aligned but not time synchronous



- The input and output sequences happen in the same order
  - Although they may not be *time synchronous*, they can be "aligned" against one another
  - E.g. Speech recognition
    - The input speech can be aligned to the phoneme sequence output

Problems

- How do we perform *inference* on such a model
  - How to output time-asynchronous sequences

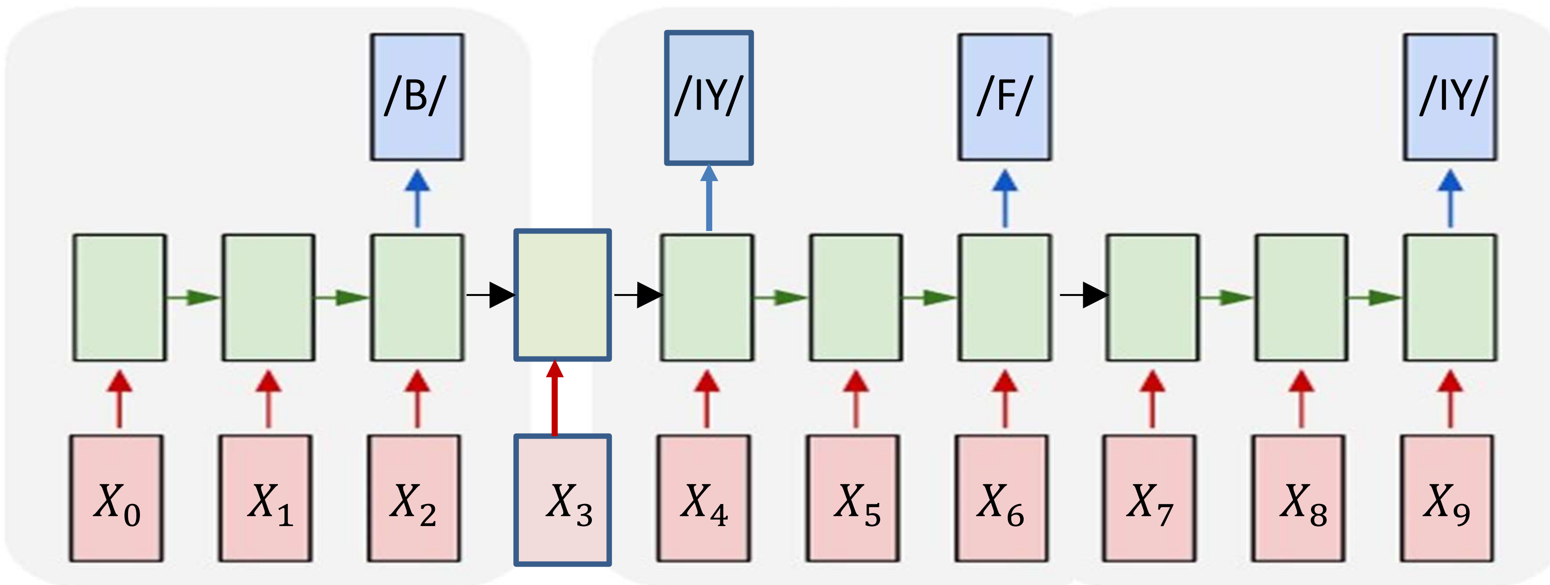
- How do we *train* such models

Problems

- How do we perform *inference* on such a model
  - How to output time-asynchronous sequences

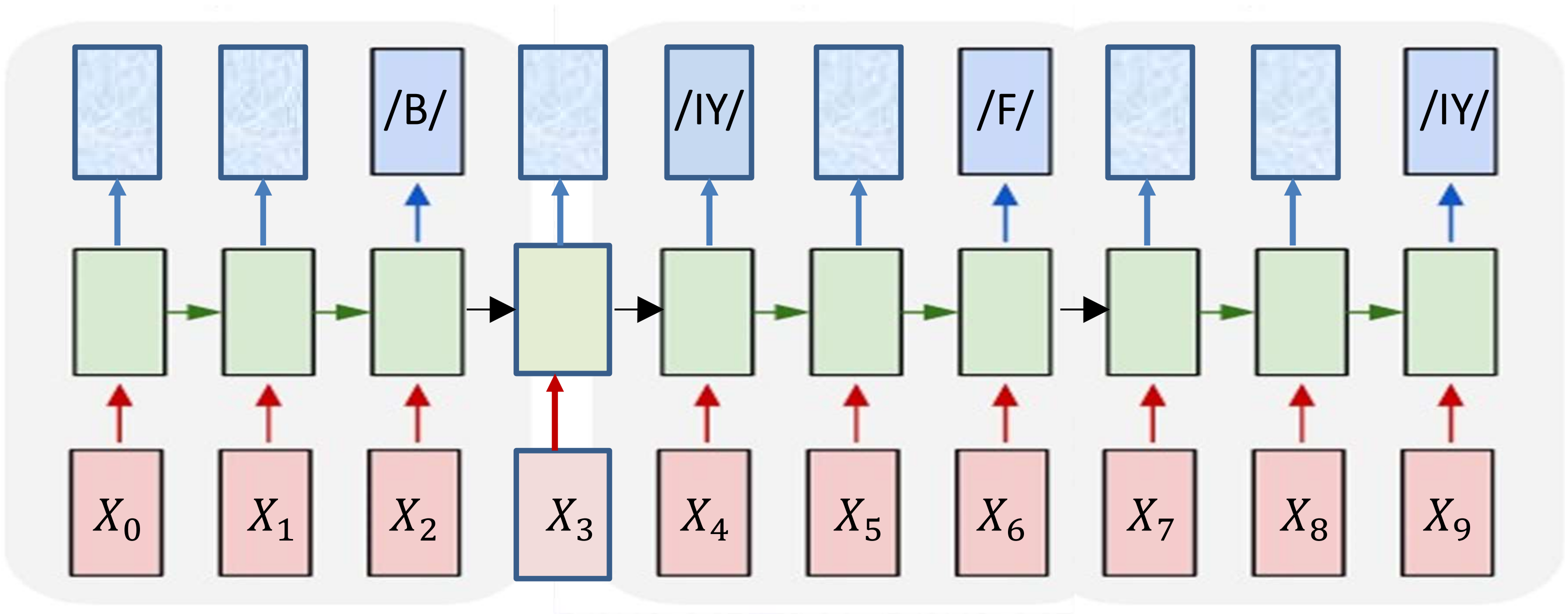
- How do we *train* such models

The inference problem



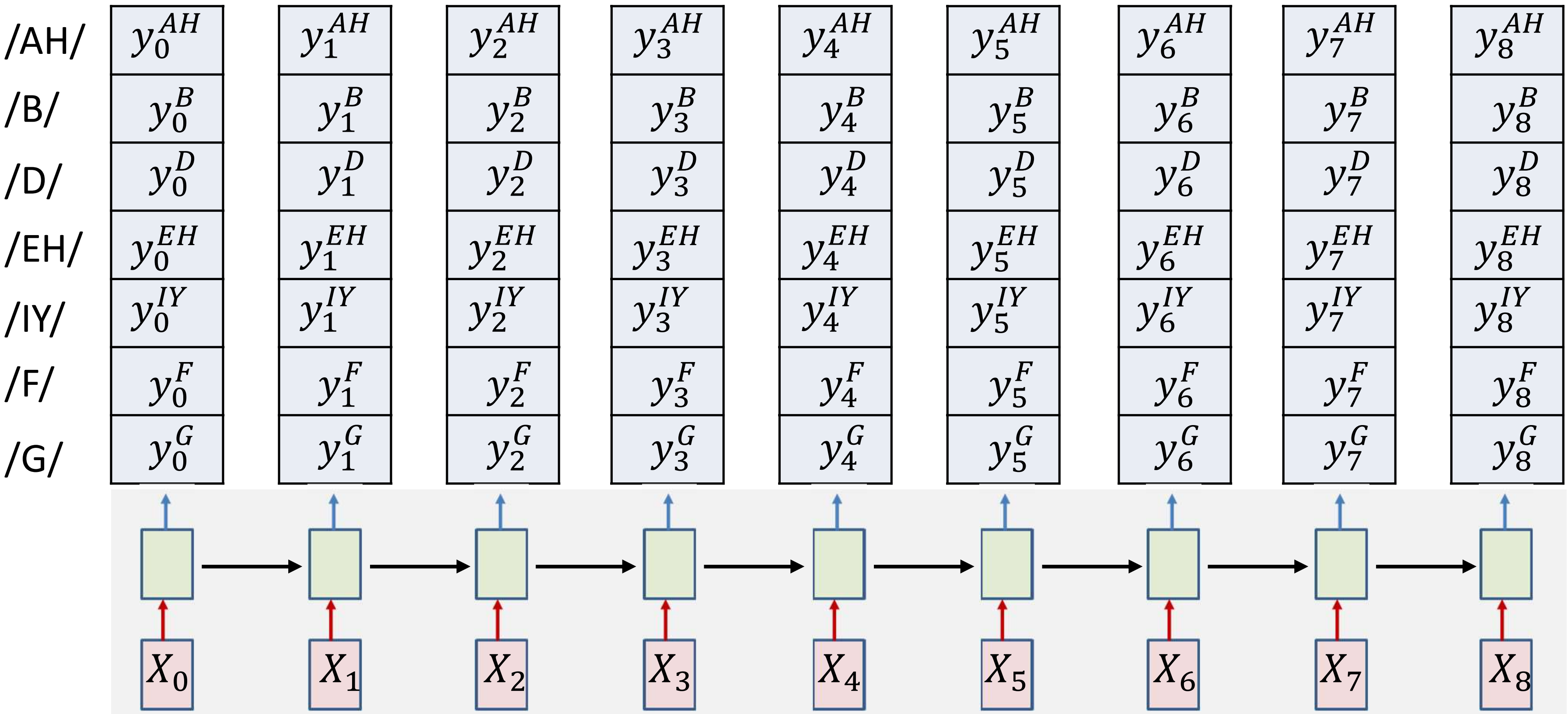
- Objective: Given a sequence of inputs, asynchronously output a sequence of symbols
  - "Decoding"

Recap: Inference



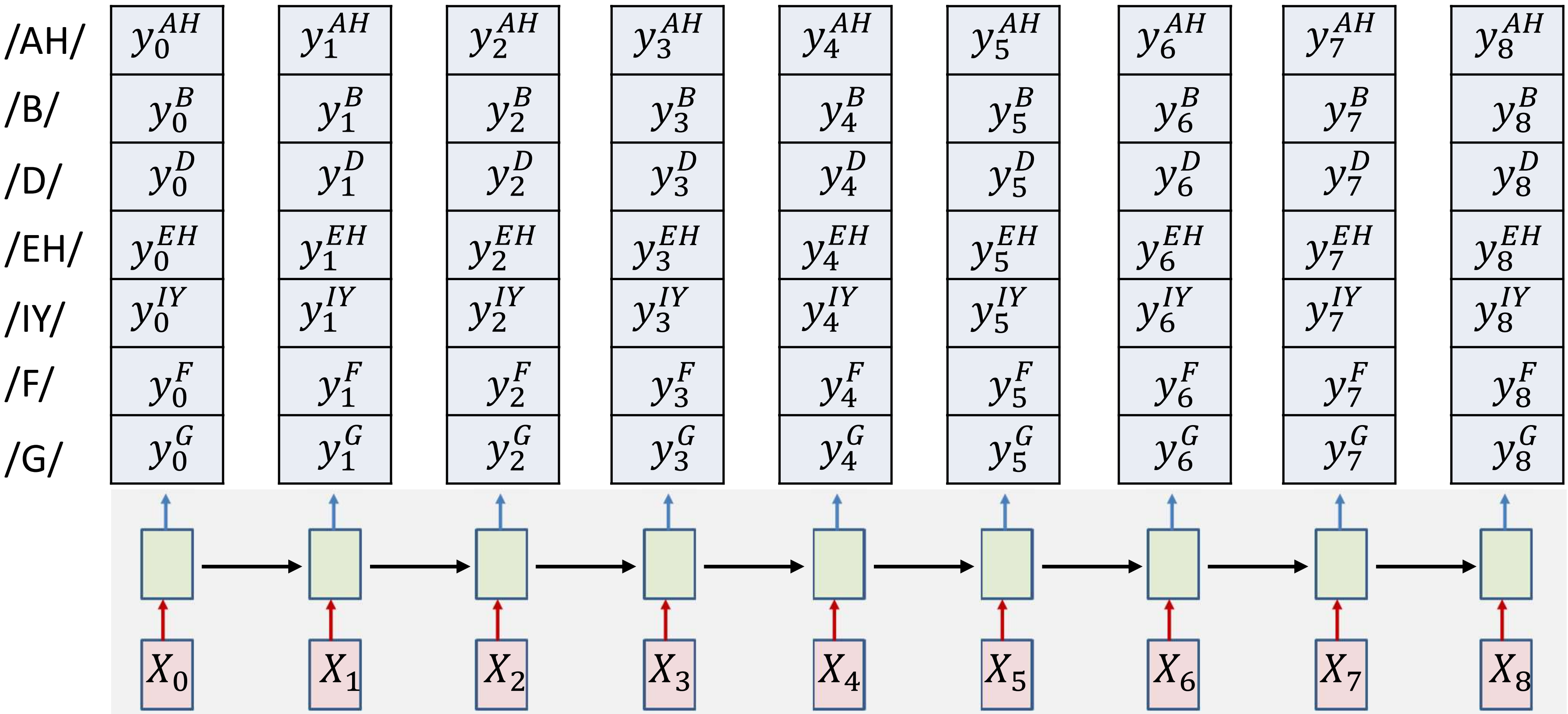
- How do we know *when* to output symbols
  - In fact, the network produces outputs at *every* time
  - *Which* of these are the *real* outputs?

The actual output of the network



- At each time the network outputs a probability for *each* output symbol given all inputs until that time
  - E.g. $y_4^D = prob(s_4 = D | X_0 \ldots X_4)$

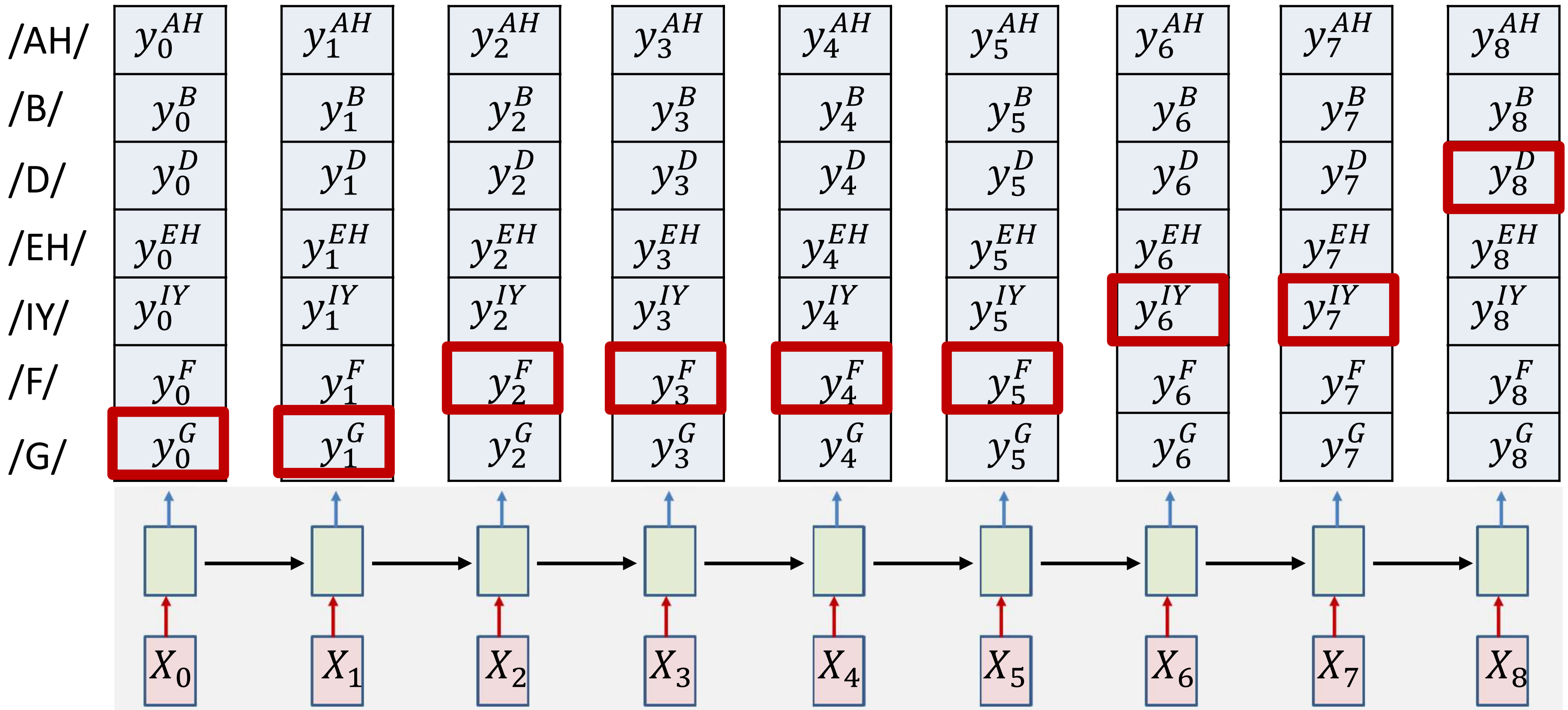
Overall objective



- Find most likely symbol sequence given inputs

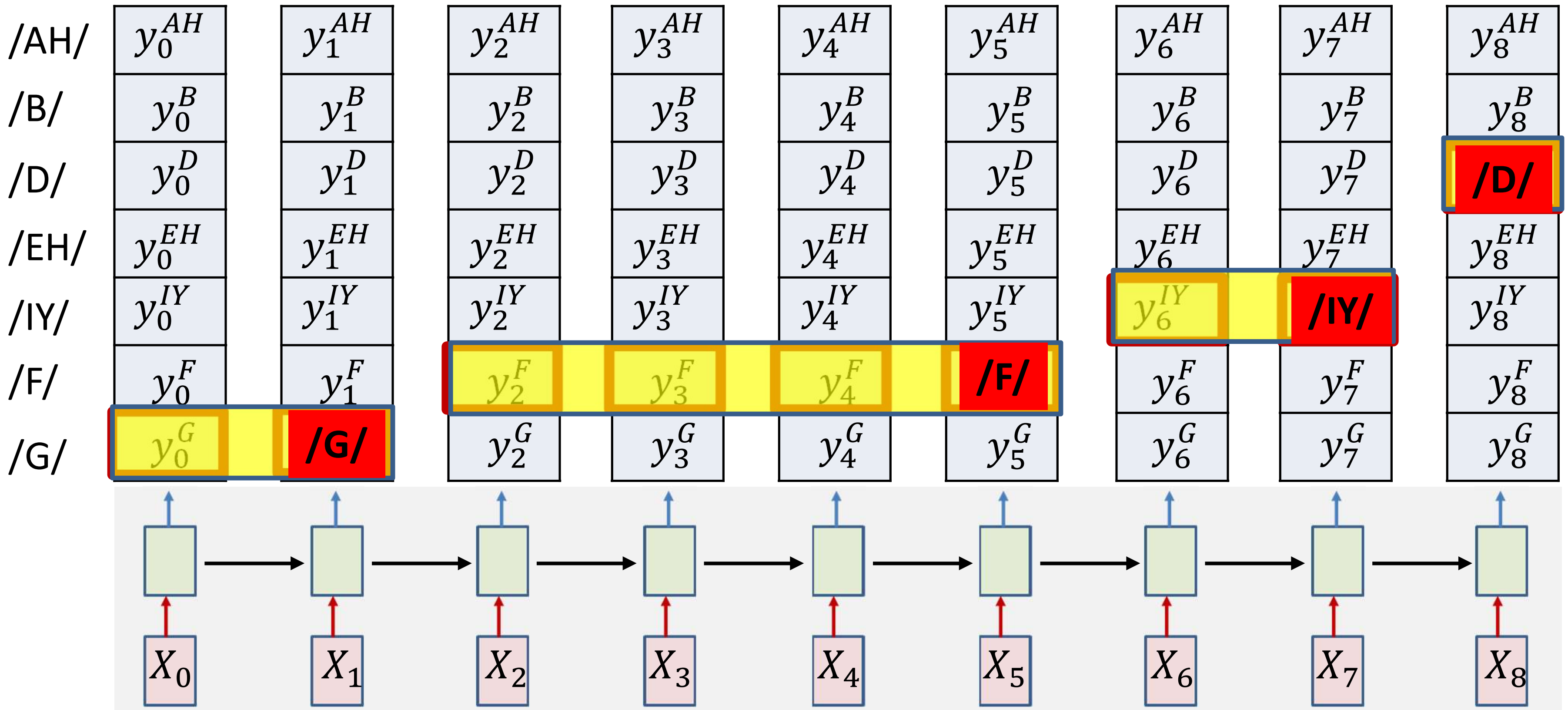
$$S_0 \dots S_{K-1} = \underset{S'_0 \dots S'_{K-1}}{\operatorname{argmax}} \; prob(S'_0 \dots S'_{K-1} | X_0 \dots X_{N-1})$$

Finding the best output



- Option 1: Simply select the most probable symbol at each time

Finding the best output



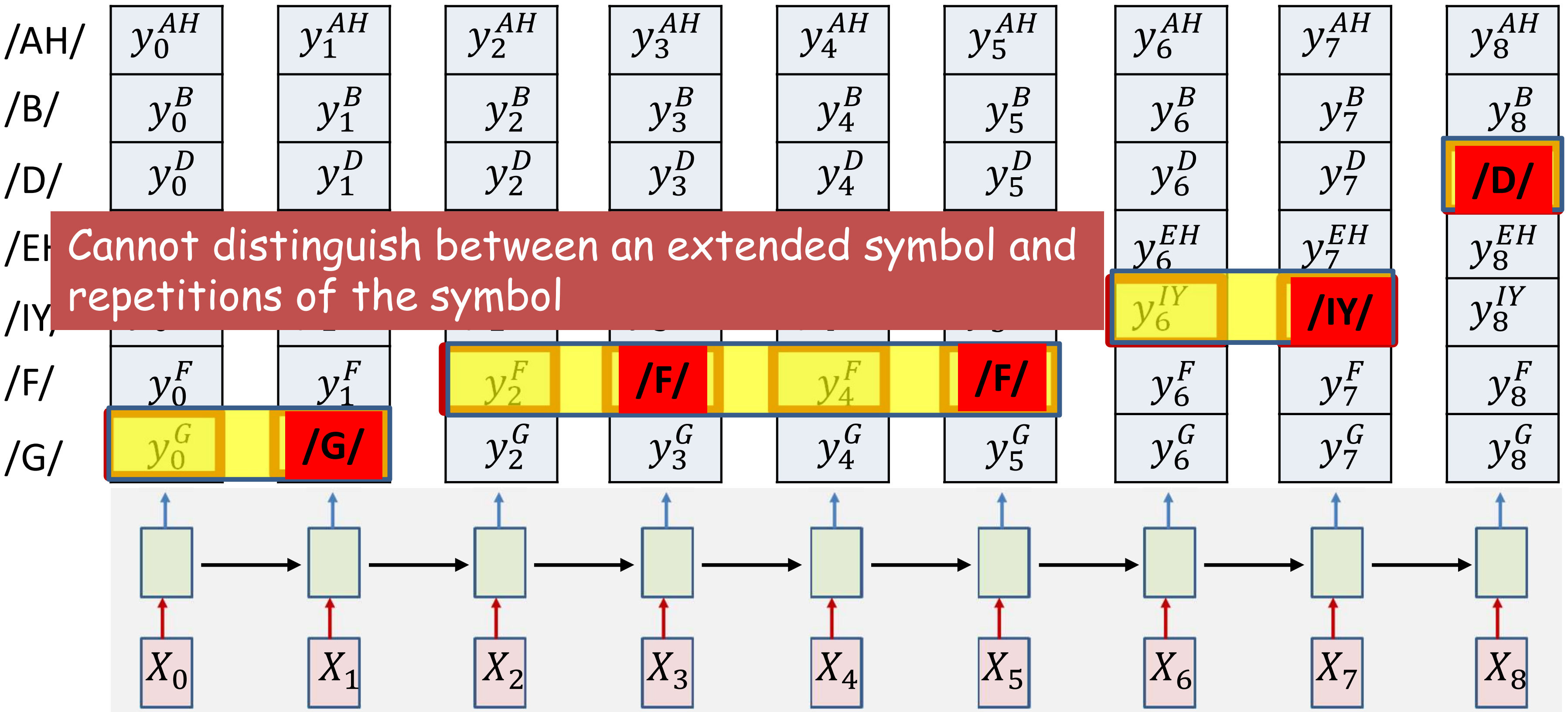
- Option 1: Simply select the most probable symbol at each time
  - *Merge* adjacent repeated symbols, and place the actual emission of the symbol in the final instant

Simple pseudocode

- Assuming $y(t, i), t = 1 \ldots T, i = 1 \ldots N$ is already computed using the underlying RNN

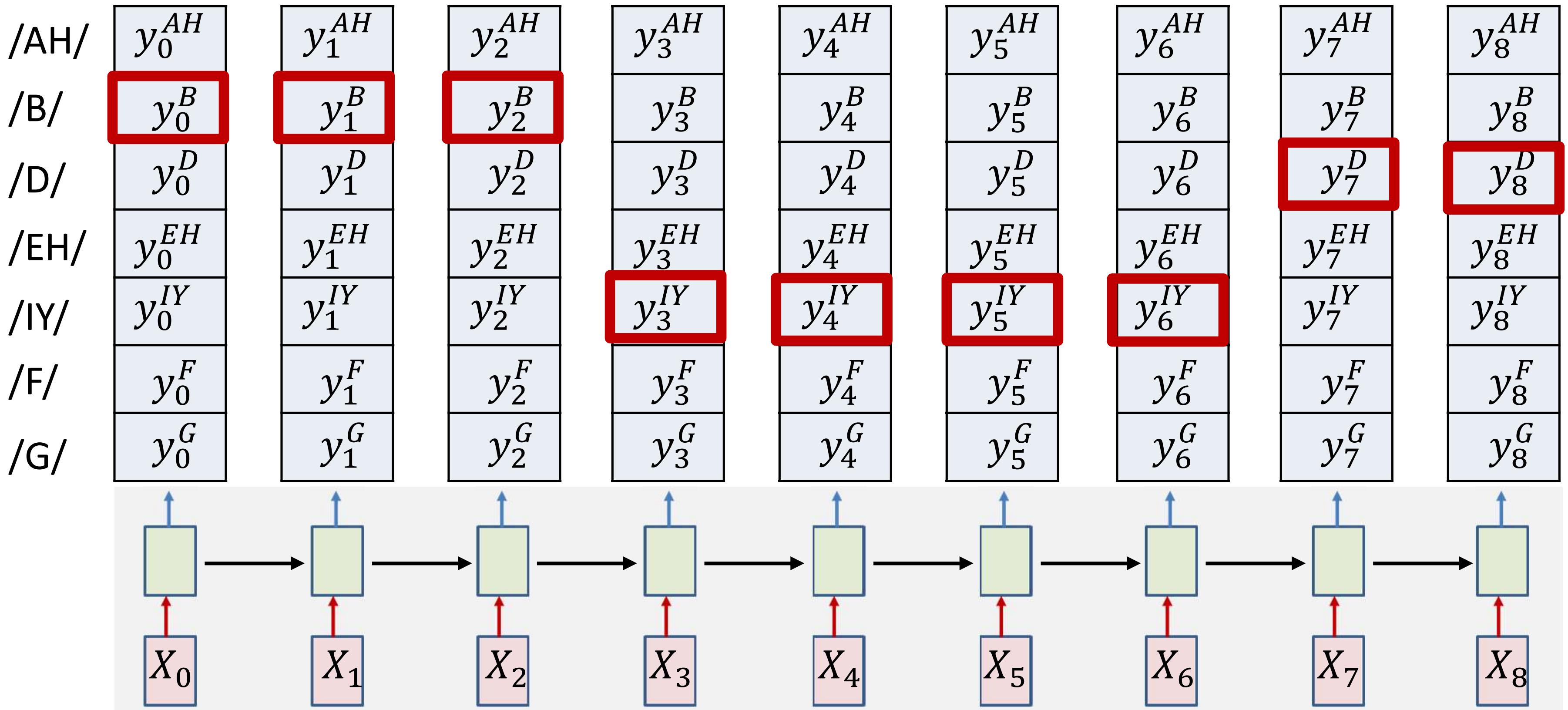
```
n = 1
best(1)= argmax_i(y(1,i))
for t = 1:T
    best(t)= argmax_i(y(t,i))
    if (best(t) != best(t-1))
        out(n) = best(t-1)
        time(n) = t-1
        n = n+1
```

The actual output of the network



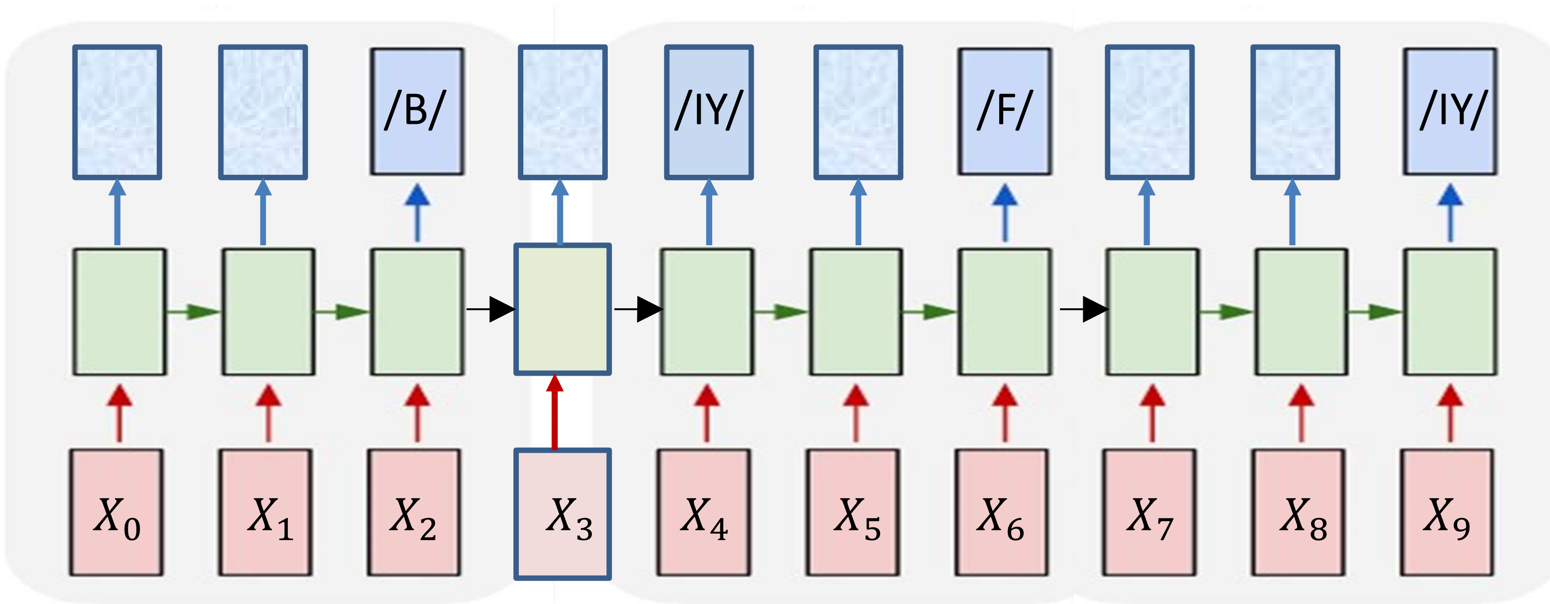
- Option 1: Simply select the most probable symbol at each time
  - *Merge* adjacent repeated symbols, and place the actual emission of the symbol in the final instant

Greedy Decoding: Recap



- This is in fact a *suboptimal* decode that actually finds the most likely *time-synchronous* output sequence
  - Which is not necessarily the most likely *order-synchronous* sequence
  - We will return to this topic later

The *sequence-to-sequence* problem

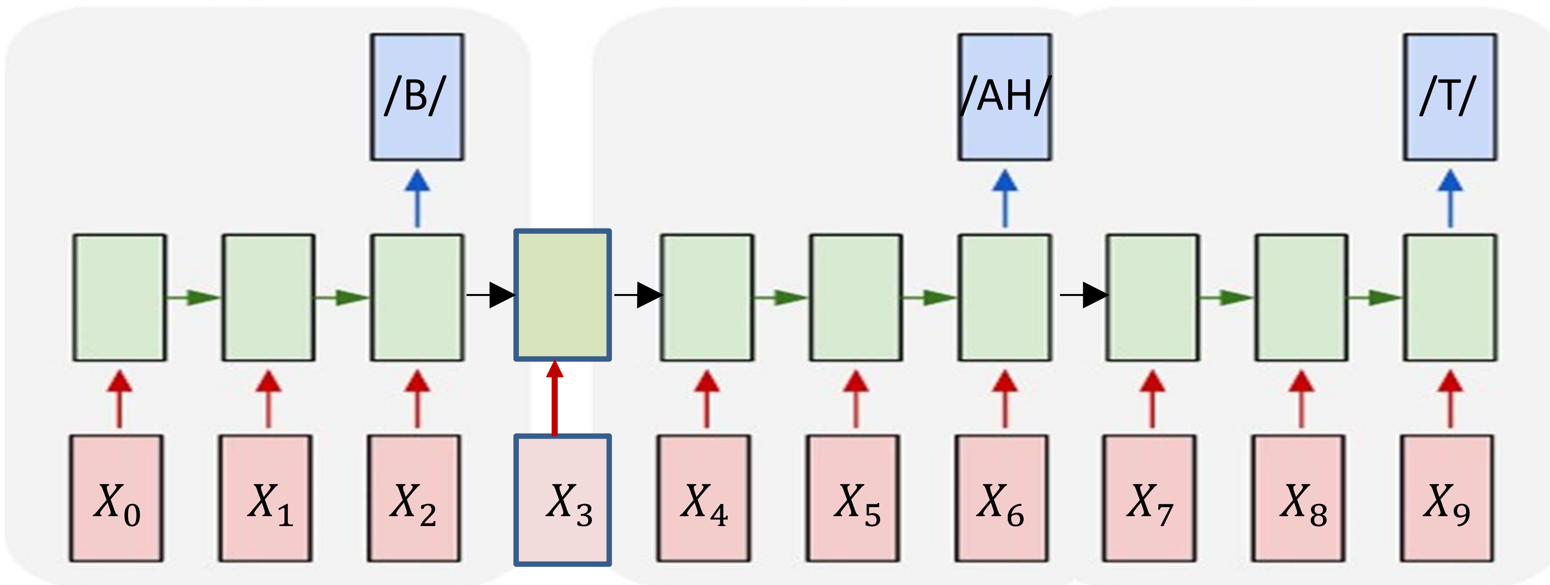


- How do we know *when* to output symbols
  - In fact, the network produces outputs at *every* time
  - *Which* of these are the *real* outputs

Addressed

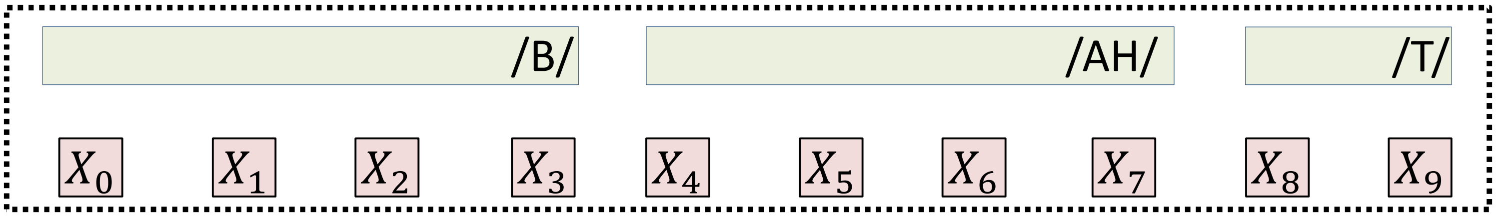
- How do we *train* these models?

Recap: Training with alignment



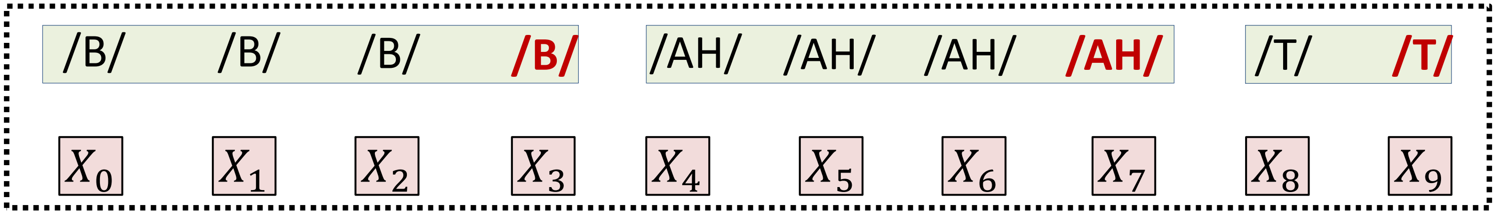
- Training data: input sequence + output sequence
  - Output sequence length <= input sequence length
- Given the *alignment* of the output to the input
  - The phoneme /B/ ends at X_2, /AH/ at X_6, /T/ at X_9

Recap: Characterizing an alignment



- Given only the order-synchronous sequence and its time stamps
  - $S_0(T_0), S_1(T_1), \ldots, S_{K-1}(T_{K-1})$
  - E.g. $S_0 = /B/\ (3)$, $S_1 = /B/\ (7)$, $S_2 = /T/\ (9)$,

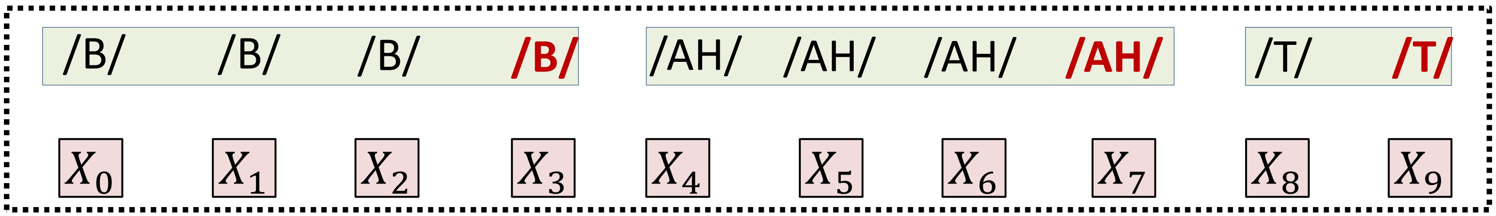
Recap: Characterizing an alignment



- Given only the order-synchronous sequence and its time stamps
  - $S_0(T_0), S_1(T_1), \dots, S_{K-1}(T_{K-1})$
  - E.g. $S_0 = /B/\ (3),\ \ S_1 = /B/\ (7),\ \ S_2 = /T/\ (9),$

- Repeat symbols to convert it to a time-synchronous sequence
  - $s_0, s_1, \dots, s_{N-1} = S_0, S_0, \dots, (T_0\ times), S_1, S_1, \dots, (T_1\ times), \dots, S_{K-1}$
  - E.g. $s_0, s_1, \dots, s_9 = /B//B//B//B//AH//AH//AH//AH//AH//T//T/$

Recap: Characterizing an alignment

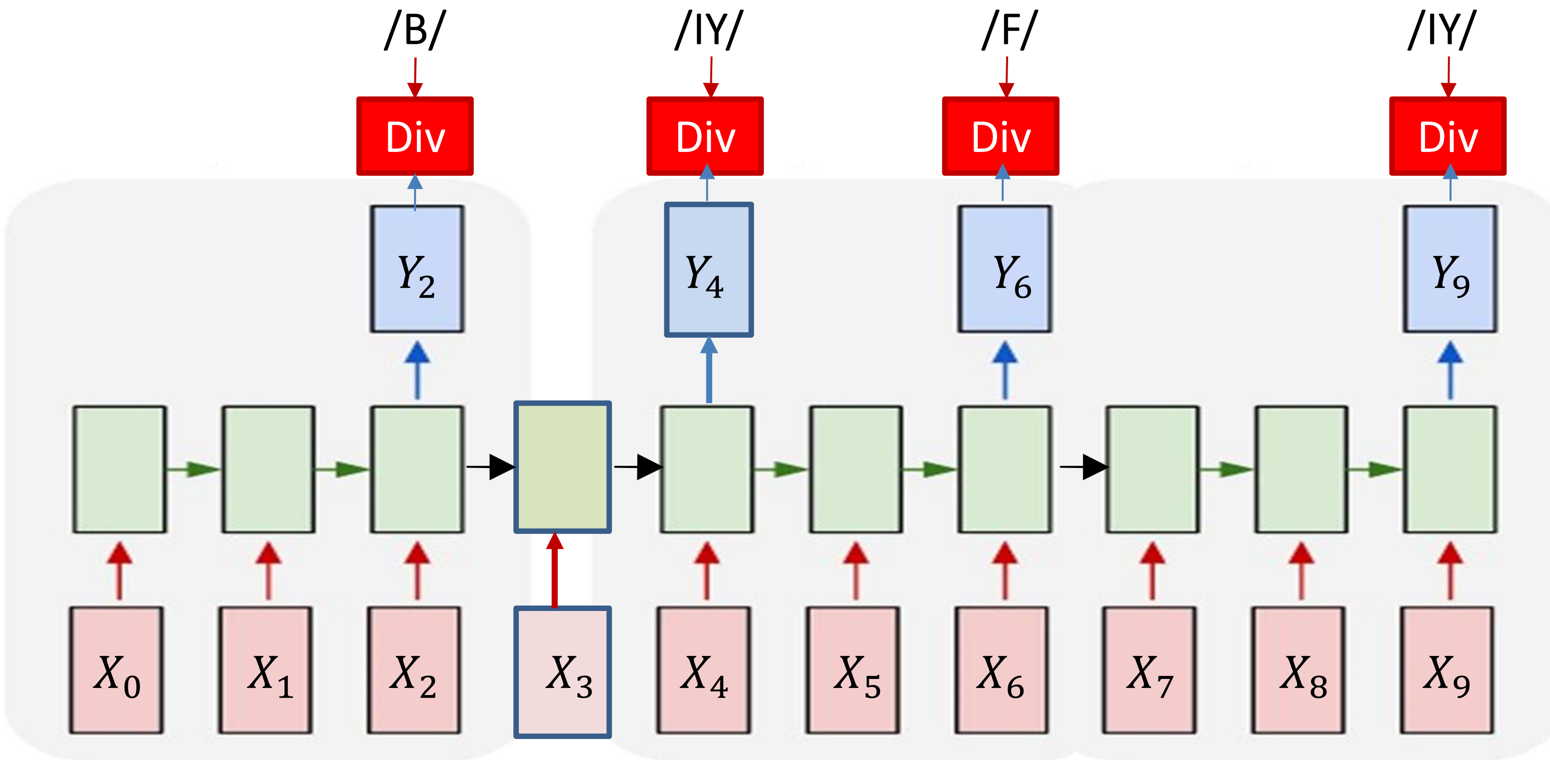


- Given only the order-synchronous sequence and its time stamps
  - $S_0(T_0), S_1(T_1), \ldots, S_{K-1}(T_{K-1})$
  - E.g. $S_0 = /B/\ (3),\ \ S_1 = /B/\ (7),\ \ S_2 = /T/\ (9),$

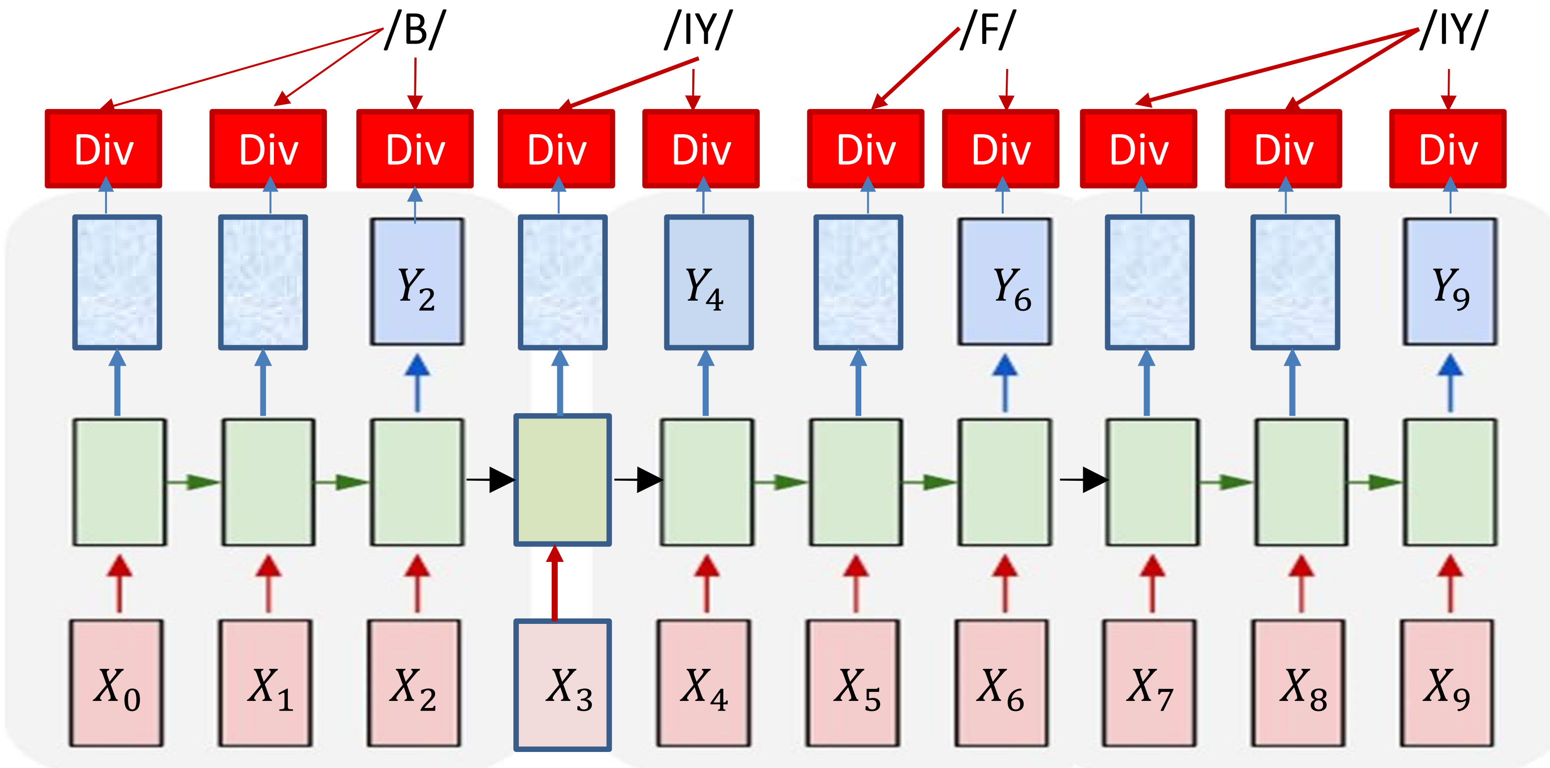
- Repeat symbols to convert it to a time-synchronous sequence
  - $s_0 = S_0, s_1 = S_0, \ldots, s_{T_0} = S_0, s_{T_0+1} = S_1, \ldots, s_{T_1} = S_1, s_{T_1+1} = S_2, \ldots, s_{N-1} = S_{K-1}$
  - E.g. $s_0, s_1, \ldots, s_9 = /B//B//B//B//AH//AH//AH//AH//AH//T//T/$

- For our purpose an alignment of $S_0 \ldots S_{K-1}$ to an input of length N has the form
  - $\boldsymbol{s_0, s_1, \ldots, s_{N-1}} = \boldsymbol{S_0, S_0, \ldots, S_0, S_1, S_1, \ldots, S_1, S_2, \ldots, S_{K-1}}$ **(of length N)**
- Any sequence of this kind of length N that contracts (by eliminating repetitions) to $S_0 \ldots S_{K-1}$ is a candidate alignment of $S_0 \ldots S_{K-1}$

Recap: Training with alignment

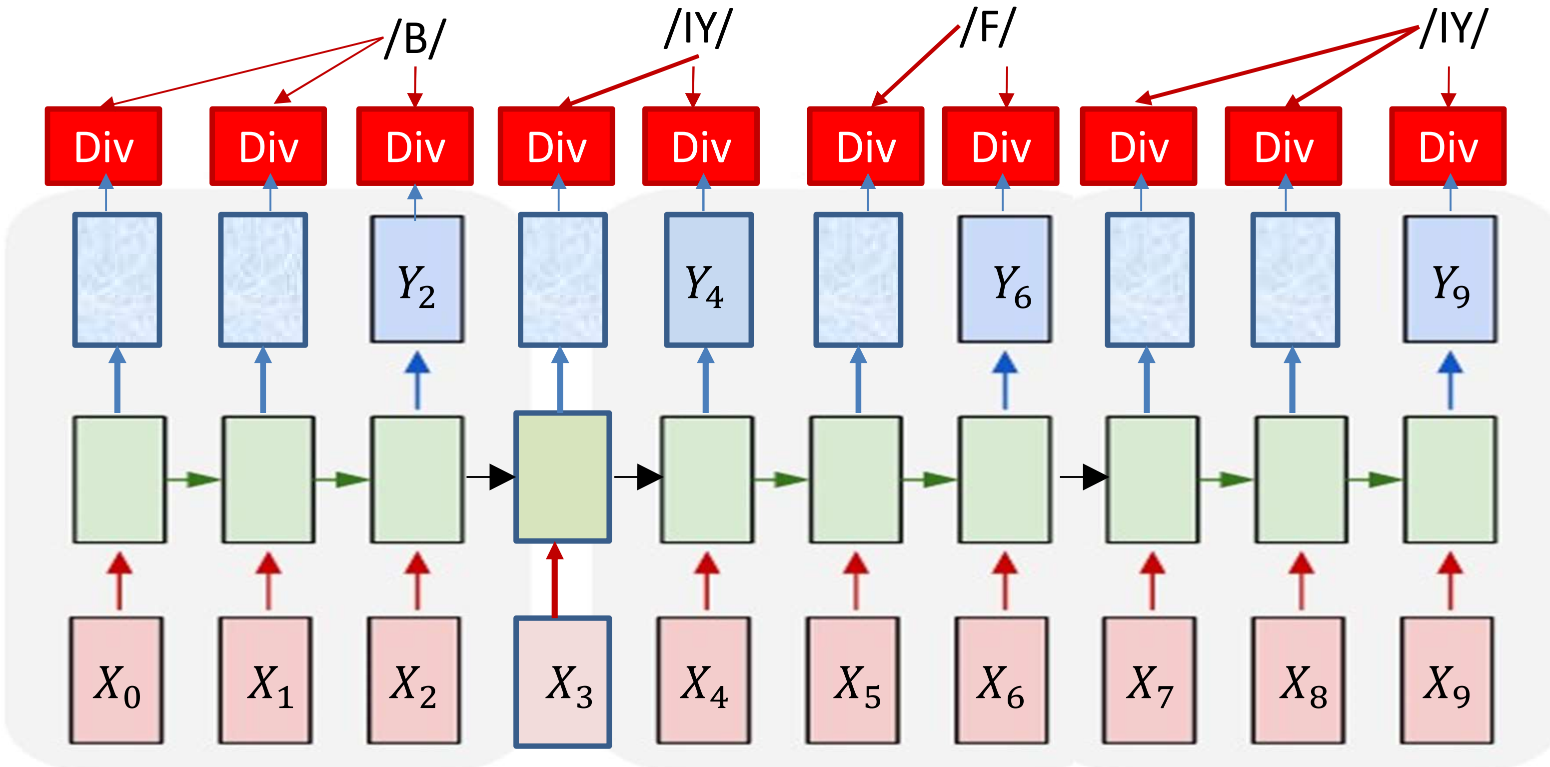


- Given the order-aligned output sequence with timing



- Given the order aligned output sequence with timing
  - Convert it to a time-synchronous alignment by repeating symbols
- Compute the divergence from the time-aligned sequence

$$DIV = \sum_t KL(Y_t, symbol_t) = -\sum_t \log Y(t, symbol_t)$$



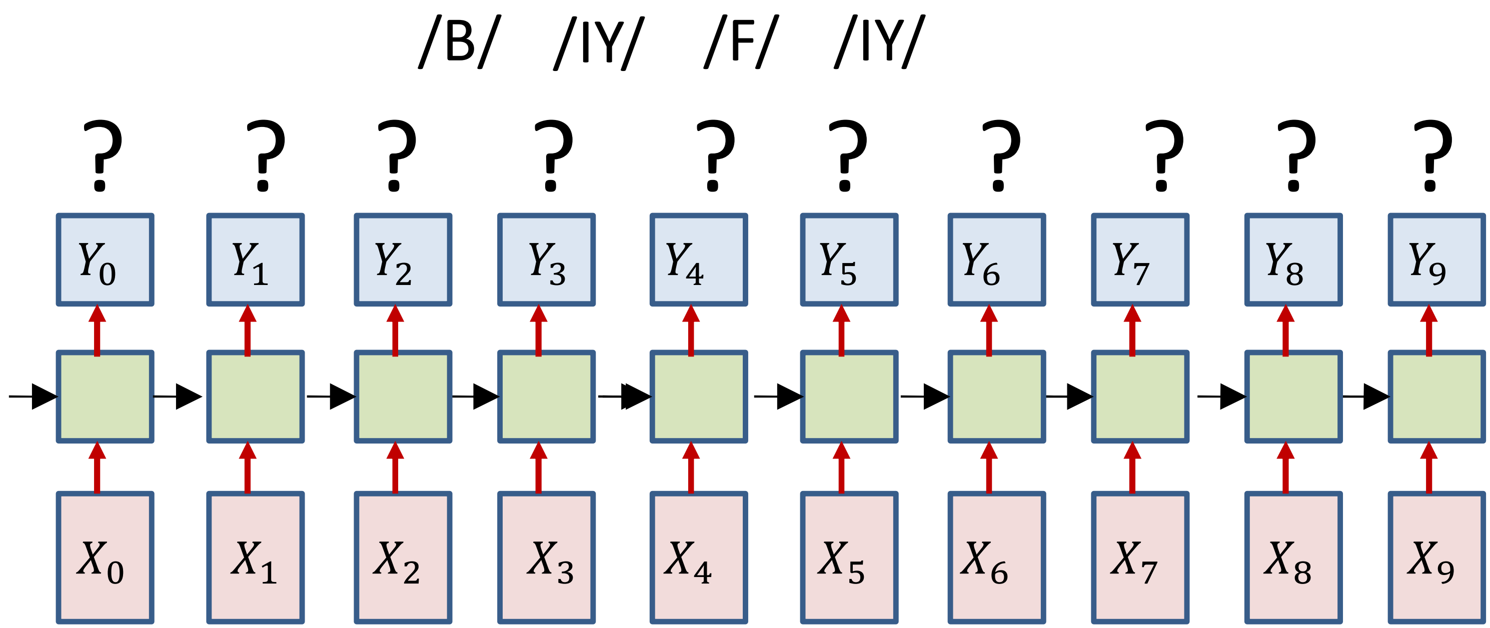
$$DIV = \sum_t KL(Y_t, symbol_t) = -\sum_t \log Y(t, symbol_t)$$

- The gradient w.r.t the t-th output vector Y_t

$$\nabla_{Y_t} DIV = \begin{bmatrix} 0 & 0 & \dots & \frac{-1}{Y(t, symbol_t)} & 0 & \dots & 0 \end{bmatrix}$$

  – Zeros except at the component corresponding to the target aligned to that time

Problem: Alignment not provided



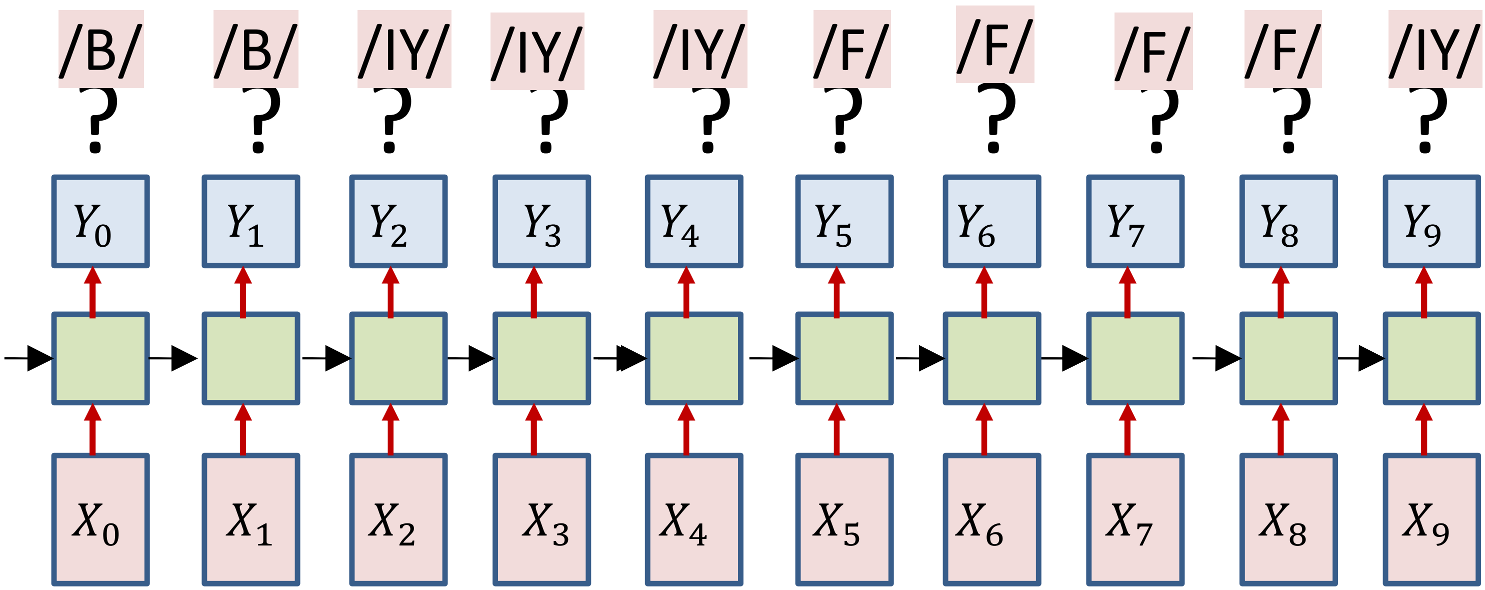
- Only the sequence of output symbols is provided for the training data
  - But no indication of which one occurs where
- How do we compute the divergence?
  - And how do we compute its gradient w.r.t. Y_t

Recap: Training *without* alignment

- We know how to train if the alignment is provided
- Problem: Alignment is *not* provided

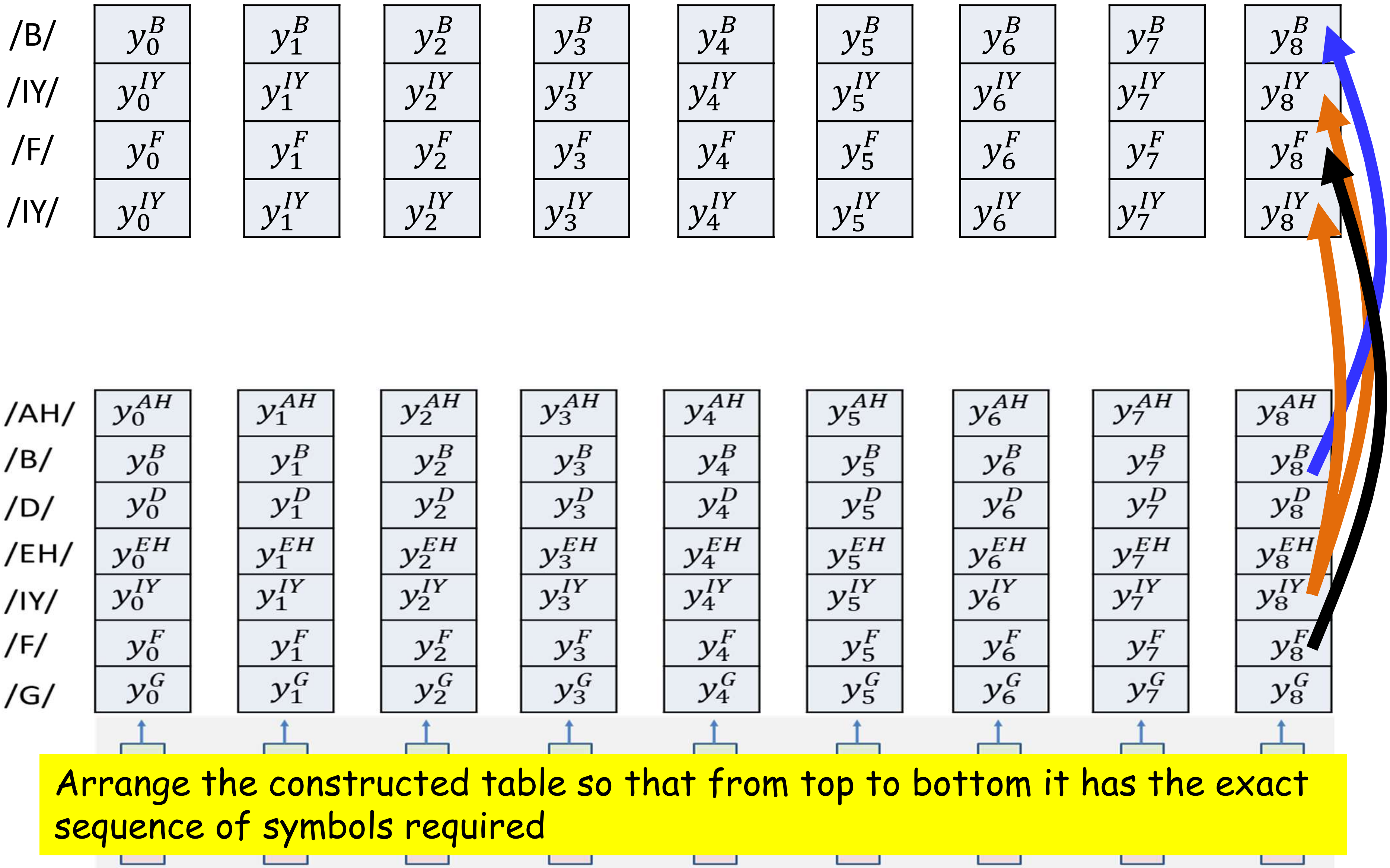
- Solution:
  1. *Guess* the alignment
  2. Consider *all possible* alignments

Solution 1: *Guess the alignment*



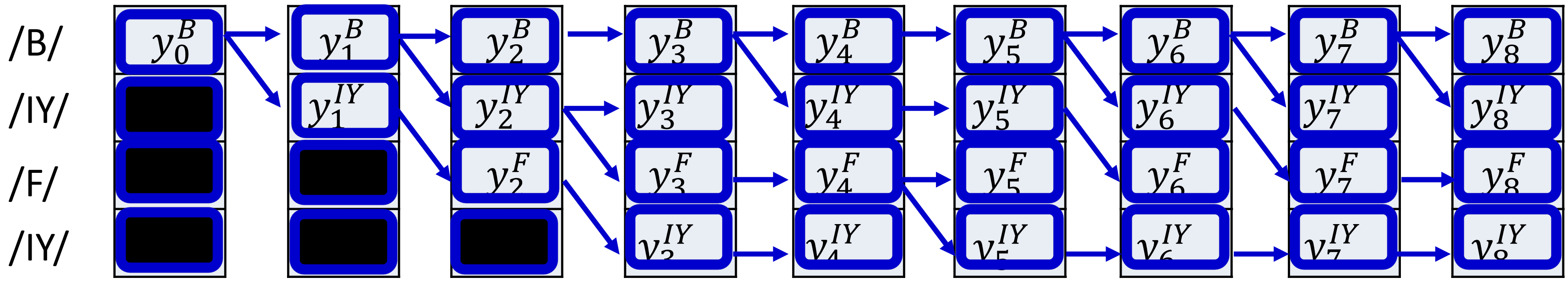
- Initialize: Assign an initial alignment
  - Either randomly, based on some heuristic, or any other rationale
- Iterate:
  - Train the network using the current alignment
  - *Reestimate* the alignment for each training instance
    - Using the Viterbi algorithm

Recap: Estimating the alignment: Step 1



Arrange the constructed table so that from top to bottom it has the exact sequence of symbols required

Recap: Viterbi algorithm



- Initialization:

$$BP(0,i) = null, \; i = 0 \ldots K-1$$

$$Bscr(0,0) = y_0^{S(0)}, \; Bscr(0,i) = -\infty, \; i = 1 \ldots K-1$$

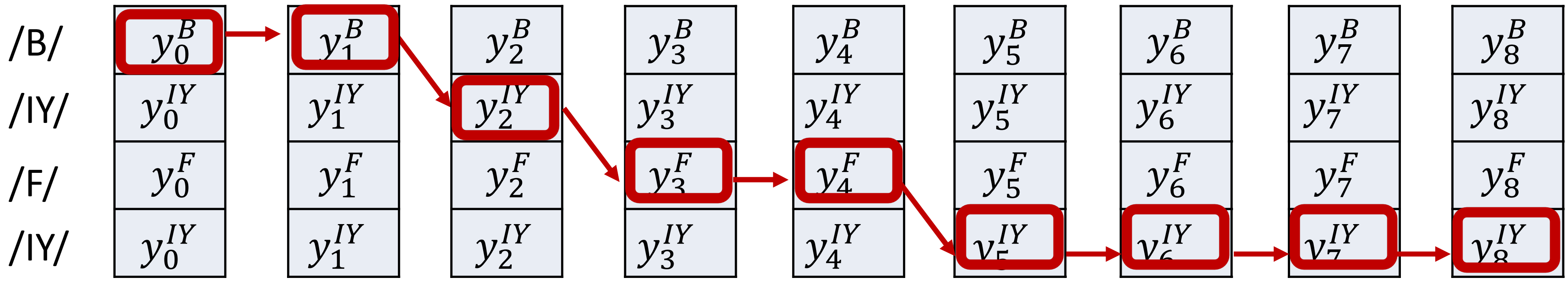
- for $t = 1 \ldots T-1$

$$BP(t,0) = 0; \; Bscr(t,0) = Bscr(t-1,0) \times y_t^{S(0)}$$

for $l = 1 \ldots K-1$

  - $BP(t,l) = \begin{pmatrix} l-1 : \; if\ (Bscr(t-1,l-1) > Bscr(t-1,l)) \; l-1; \\ l : else \end{pmatrix}$
  - $Bscr(t,l) = Bscr(BP(t,l)) \times y_t^{S(l)}$

Recap: Viterbi algorithm



- $s(T-1) = S(K-1)$
- for $t = T-1 \; downto \; 1$

  $s(t-1) = BP(s(t))$

/B/ /B/ /IY/ /F/ /F/ /IY/ /IY/ /IY/ /IY/

VITERBI

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#T = length of input

#First create output table
For i = 1:N
    s(1:T,i) = y(1:T, S(i))

#Now run the Viterbi algorithm
# First, at t = 1
BP(1,1) = -1
Bscr(1,1) = s(1,1)
Bscr(1,2:N) = -infty
for t = 2:T
    BP(t,1) = 1;
    Bscr(t,1) = Bscr(t-1,1)*s(t,1)
    for i = 1:min(t,N)
          BP(t,i) = Bscr(t-1,i) > Bscr(t-1,i-1) ? i : i-1
          Bscr(t,i) = Bscr(t-1,BP(t,i))*s(t,i)

# Backtrace
AlignedSymbol(T) = N
for t = T downto 2
    AlignedSymbol(t-1) = BP(t,AlignedSymbol(t))
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

VITERBI

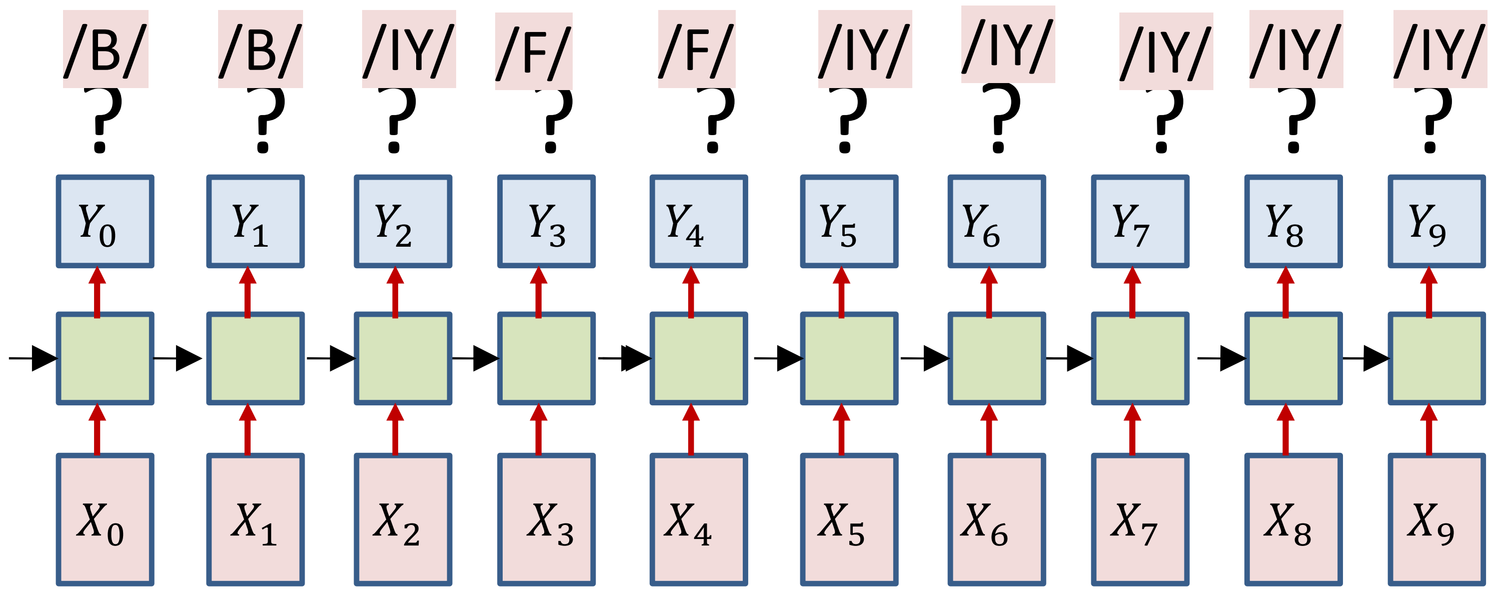
```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#T = length of input
```

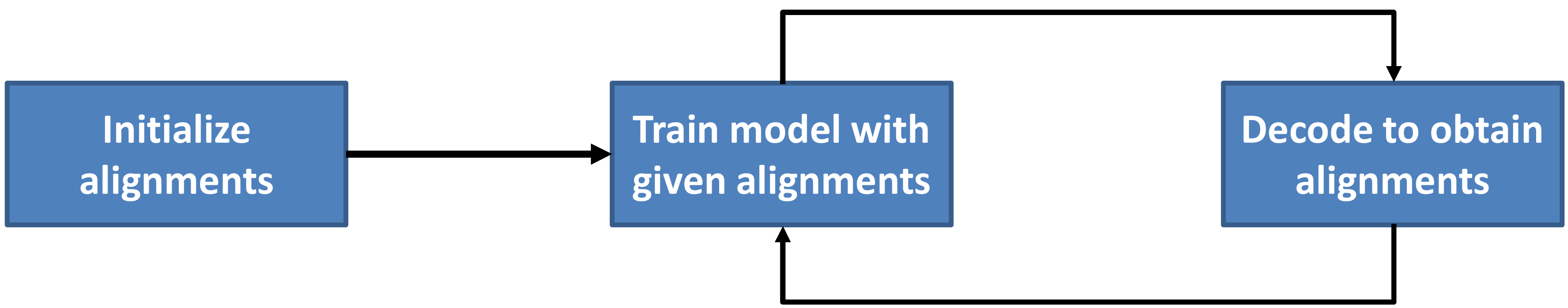
Without explicit construction of output table

```
# First, at t = 1
BP(1,1) = -1
Bscr(1,1) = y(1,S(1))
Bscr(1,2:N) = -infty
for t = 2:T
    BP(t,1) = 1;
    Bscr(t,1) = Bscr(t-1,1)*y(t,S(1))
    for i = 2:min(t,N)
        BP(t,i) = Bscr(t-1,i) > Bscr(t-1,i-1) ? i : i-1
        Bscr(t,i) = Bscr(t-1,BP(t,i))*y(t,S(i))

# Backtrace
AlignedSymbol(T) = N
for t = T downto 2
    AlignedSymbol(t-1) = BP(t,AlignedSymbol(t))
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

Recap: Iterative Estimate and Training





The "decode" and "train" steps may be combined into a single "decode, find alignment compute derivatives" step for SGD and mini-batch updates

Iterative update: Problem

- Approach heavily dependent on initial alignment

- Prone to poor local optima

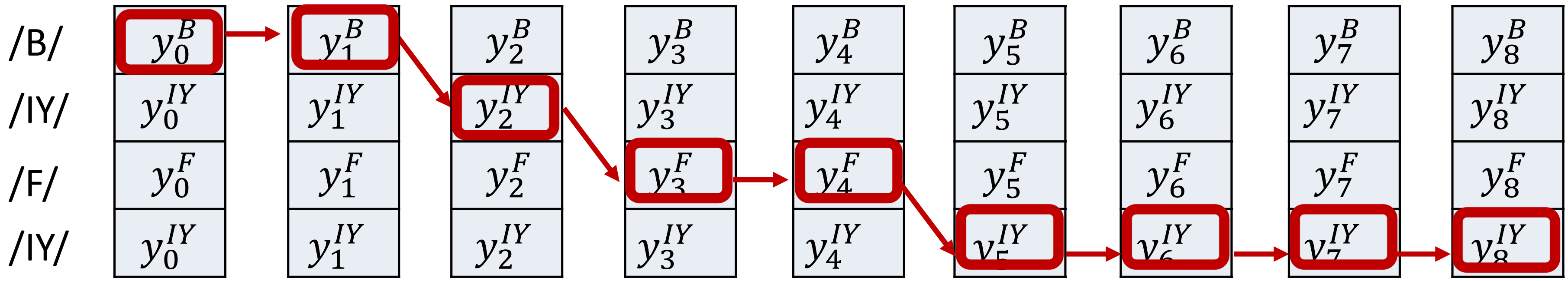
- Alternate solution: Do not commit to an alignment during any pass..

Recap: Training *without* alignment

- We know how to train if the alignment is provided
- Problem: Alignment is *not* provided

- Solution:
  1. *Guess* the alignment
  2. Consider *all possible* alignments

The reason for suboptimality

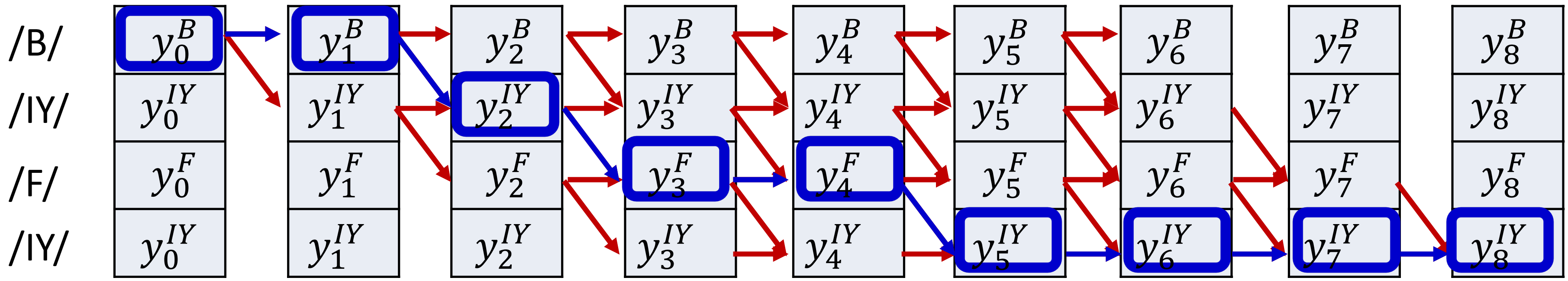


- We *commit* to the single "best" estimated alignment
  - The *most likely* alignment

$$DIV = -\sum_t \log Y\left(t, symbol_t^{bestpath}\right)$$

  - This can be way off, particularly in early iterations, or if the model is poorly initialized

The reason for suboptimality



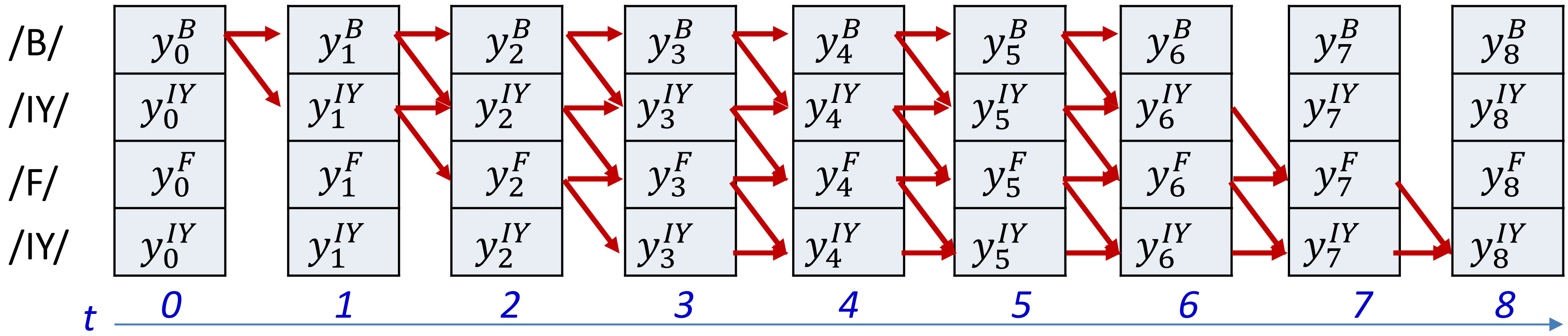
- We *commit* to the single "best" estimated alignment
  - The *most likely* alignment

$$DIV = -\sum_t \log Y\left(t, symbol_t^{bestpath}\right)$$

  - This can be way off, particularly in early iterations, or if the model is poorly initialized

- **Alternate view:** there is a probability distribution over alignments of the target Symbol sequence (to the input)
  - *Selecting a single alignment is the same as drawing a single sample from it*
  - Selecting the most likely alignment is the same as deterministically always drawing the most probable value from the distribution

Averaging over *all* alignments



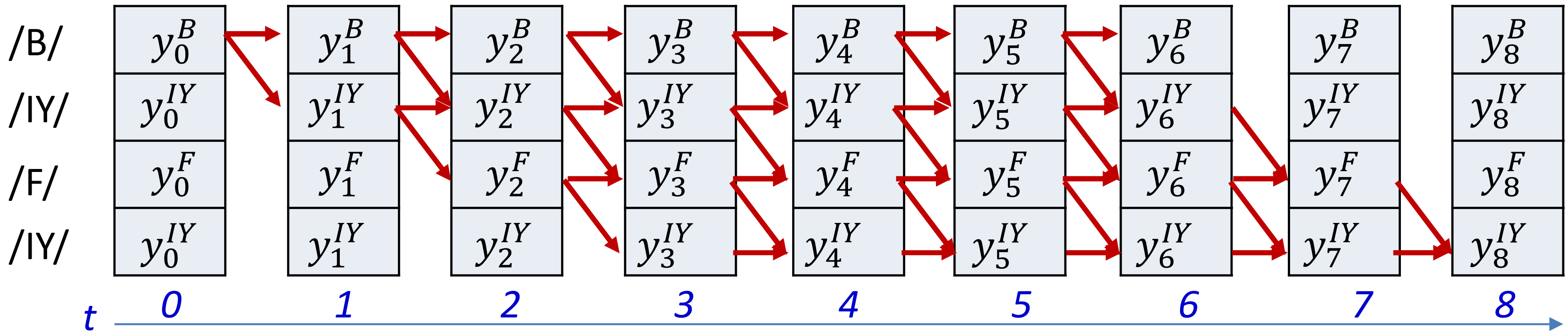
- Instead of only selecting the most likely alignment, use the statistical expectation over *all* possible alignments

$$DIV = E\left[-\sum_t \log Y(t, s_t)\right]$$

  – Use the *entire distribution of alignments*

  – This will mitigate the issue of suboptimal selection of alignment

The expectation over *all* alignments



$$DIV = E\left[-\sum_t \log Y(t, s_t)\right]$$

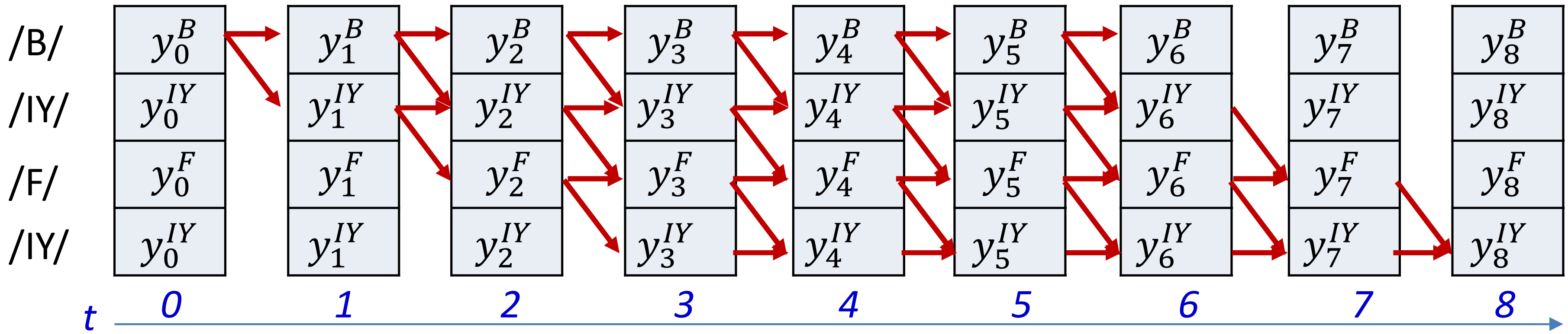
- Using the linearity of expectation

$$DIV = -\sum_t E[\log Y(t, s_t)]$$

  - This reduces to finding the expected divergence *at each input*

$$DIV = -\sum_t \sum_{S \in S_1 \dots S_K} P(s_t = S|\mathbf{S}, \mathbf{X}) \log Y(t, s_t = S)$$

The expectation over *all* alignments

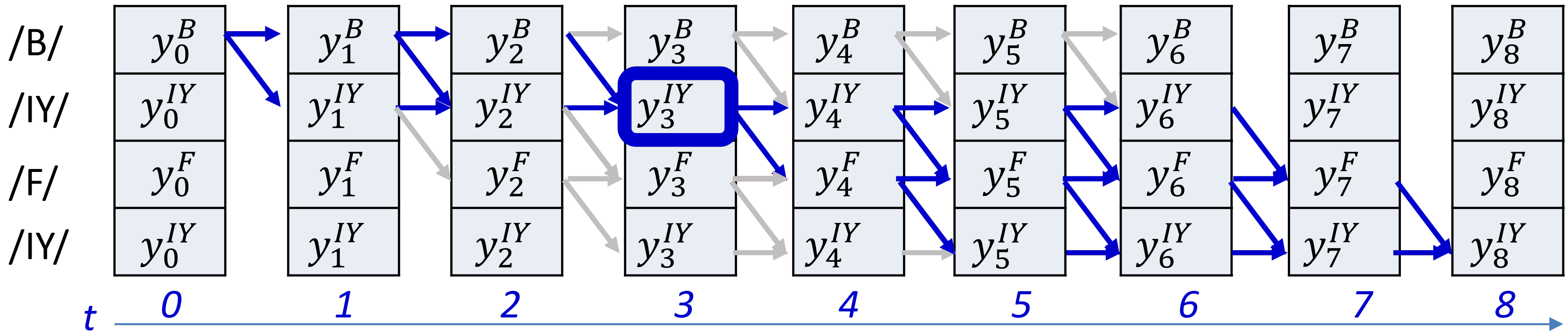


The probability of aligning the specific symbol s at time t, *given* that unaligned sequence $\mathbf{S} = S_0 \ldots S_{K-1}$ *and given* the input sequence $\mathbf{X} = X_0 \ldots X_{N-1}$
We need to be able to compute this

- $$DIV = -\sum_t E[\log Y(t, s_t)]$$
  - This reduces to finding the expected divergence *at each input*

$$DIV = -\sum_t \sum_{S \in S_1 \ldots S_K} P(s_t = S | \mathbf{S}, \mathbf{X}) \log Y(t, s_t = S)$$

A posteriori probabilities of symbols

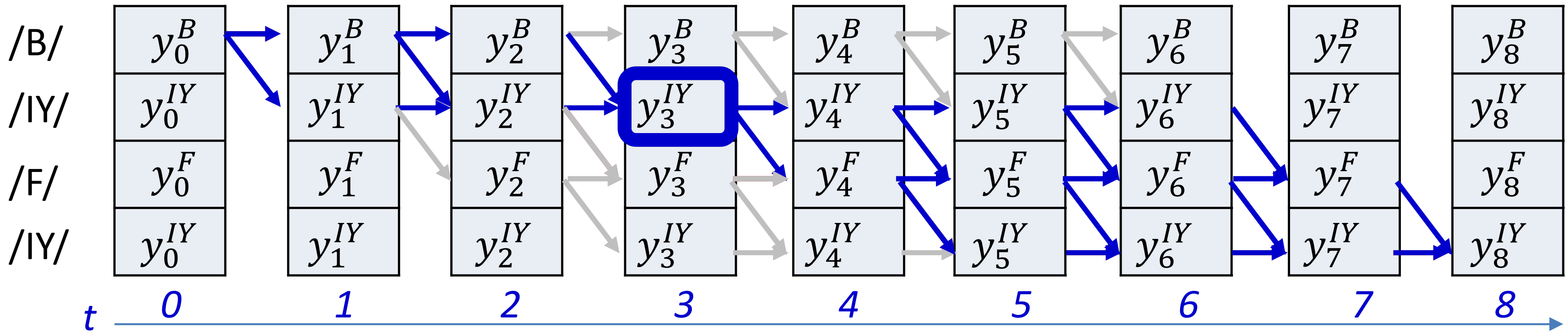


$$P(s_t = S_r|\mathbf{S}, \mathbf{X}) \propto P(s_t = S_r, \mathbf{S}|\mathbf{X})$$

- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ is the total probability of all valid paths *in the graph for target sequence* $\mathbf{S}$ that go through the symbol S_r (the r^{th} symbol in the sequence $S_0 \ldots S_{K-1}$) at time t

- We will compute this using the "forward-backward" algorithm

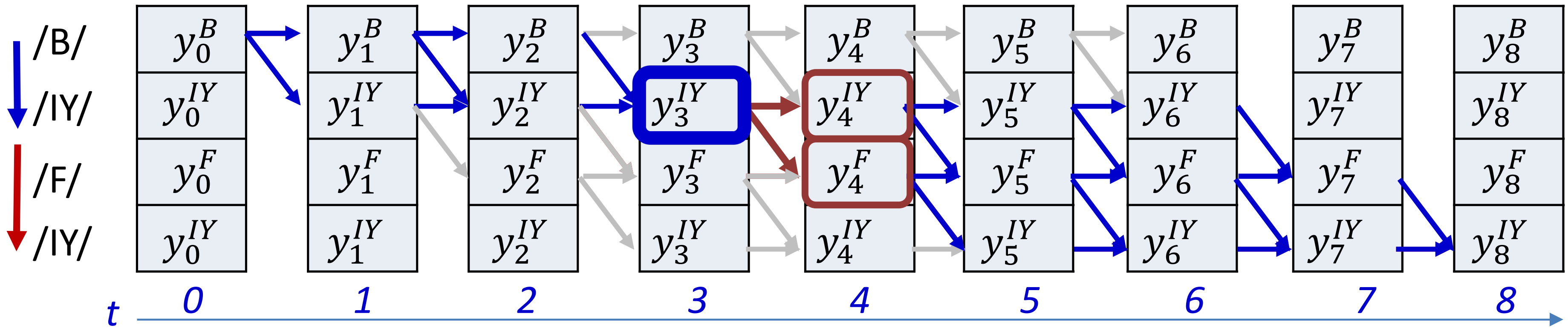
A posteriori probabilities of symbols



- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ can be decomposed as

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_0, \ldots, S_{K-1}, s_t = S_r|\mathbf{X})$$

A posteriori probabilities of symbols



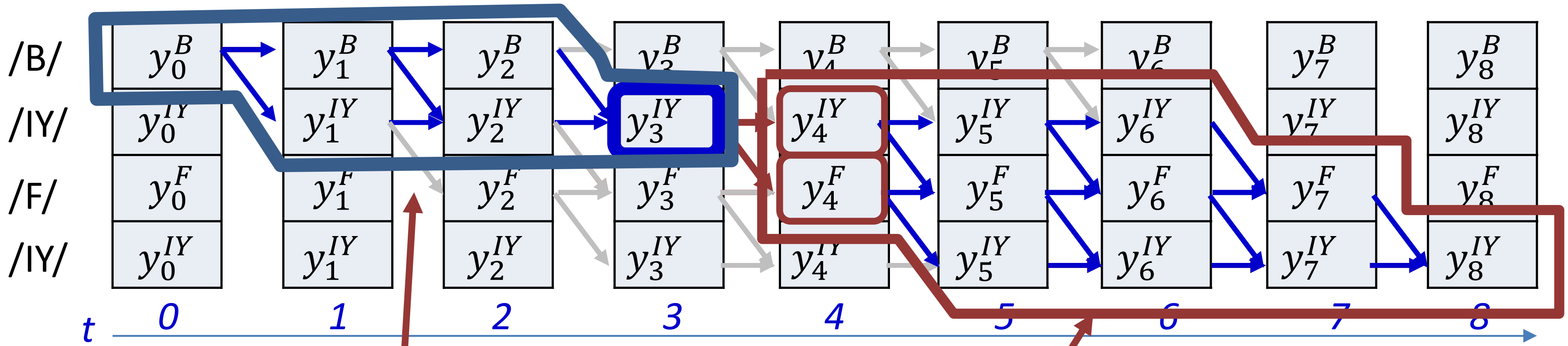
- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ can be decomposed as

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_0, \ldots, S_r, \ldots, S_{K-1}, s_t = S_r|\mathbf{X})$$

$$= P(S_0 \ldots S_r, s_t = S_r, s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1}, |\mathbf{X})$$

- Where $succ(S_r)$ is a symbol that can follow S_r in a sequence
  - Here it is either S_r or S_{r+1} (red blocks in figure)
  - The equation literally says that after the blue block, either of the two red arrows may be followed

A posteriori probabilities of symbols



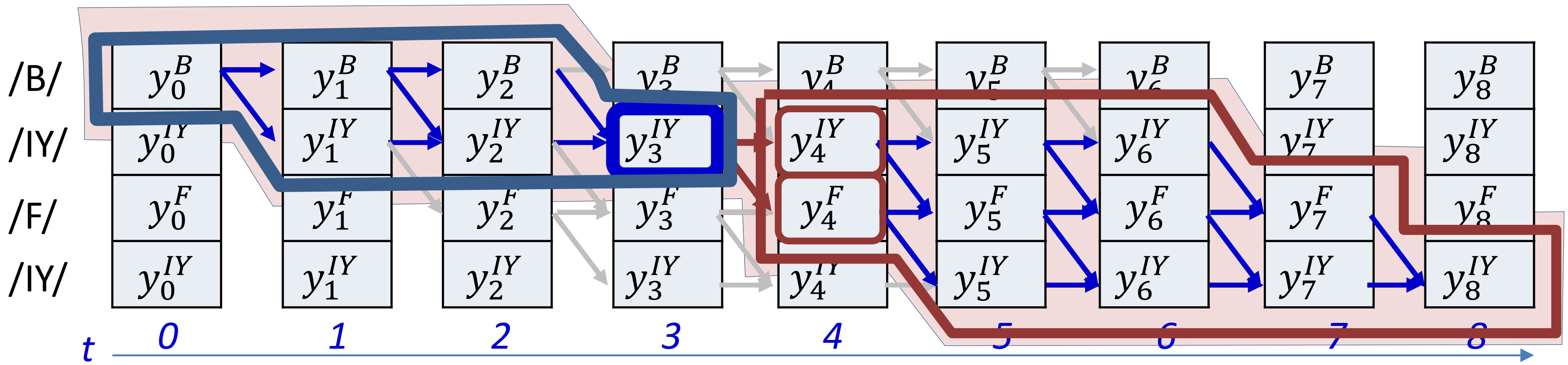
- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ can be decomposed as

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_1, \ldots, S_r, \ldots, S_K, s_t = S_r|\mathbf{X})$$

$$= P(S_0 \ldots S_r, s_t = S_r, s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1}, |\mathbf{X})$$

- Where $succ(S_r)$ is a symbol that can follow S_r in a sequence
  - Here it is either S_r or S_{r+1} (red blocks in figure)
  - The equation literally says that after the blue block, either of the two red arrows may be followed

A posteriori probabilities of symbols



- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ can be decomposed as

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_0, \dots, S_r, \dots, S_{K-1}, s_t = S_r|\mathbf{X})$$

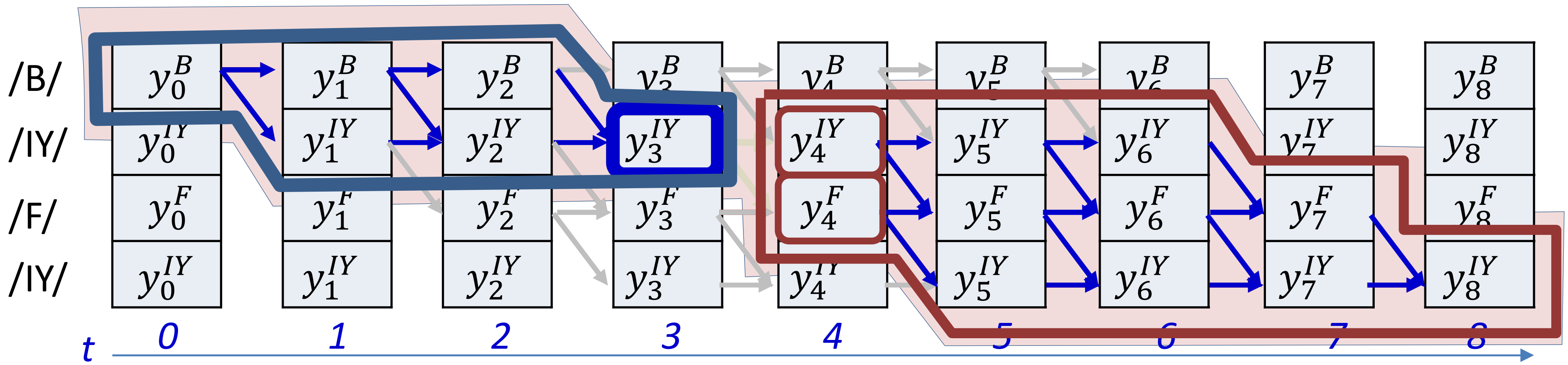
$$= P(S_0 \dots S_r, s_t = S_r, s_{t+1} \in succ(S_r), succ(S_r), \dots, S_{K-1}|\mathbf{X})$$

- Using Bayes Rule

$$= P(S_0 \dots S_r, s_t = S_r\ |\mathbf{X})P(s_{t+1} \in succ(S_r), succ(S_r), \dots, S_{K-1}\ |S_0 \dots S_r, s_t = S_r\mathbf{X})$$

- The probability of the subgraph in the blue outline, times the conditional probability of the red-encircled subgraph, given the blue subgraph

A posteriori probabilities of symbols



- $P(s_t = S_r, \mathbf{S}|\mathbf{X})$ can be decomposed as

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_0, \dots, S_r, \dots, S_{K-1}, s_t = S_r|\mathbf{X})$$

$$= P(S_0 \dots S_r, s_t = S_r, s_{t+1} \in succ(S_r), succ(S_r), \dots, S_{K-1}|\mathbf{X})$$

- Using Bayes Rule

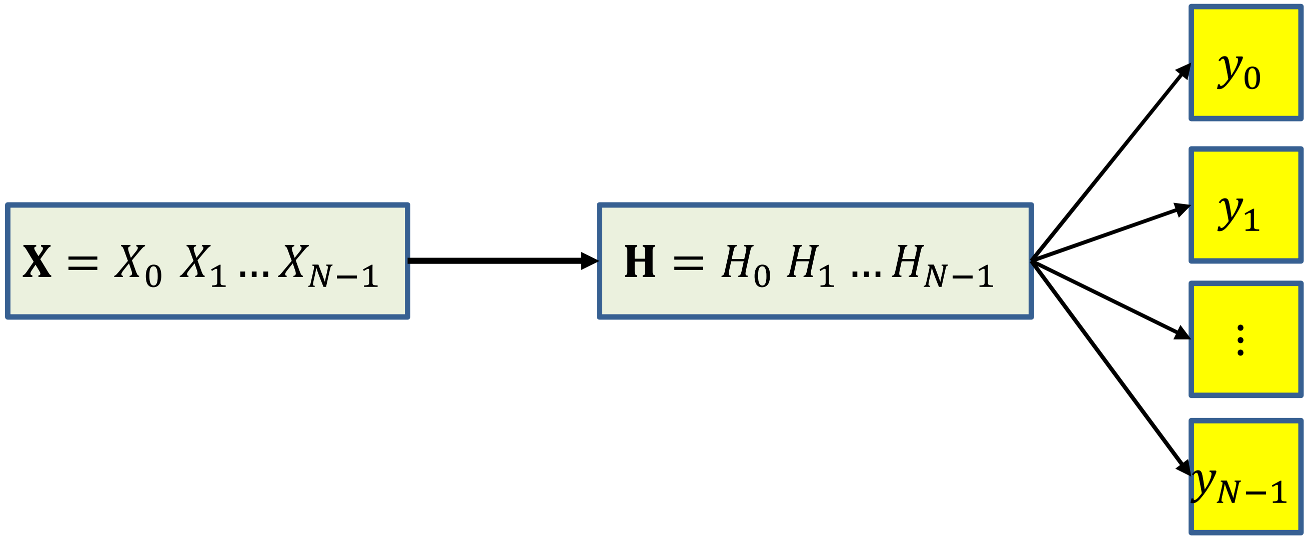
$$= P(S_0 \dots S_r, s_t = S_r\,|\mathbf{X})P(s_{t+1} \in succ(S_r), succ(S_r), \dots, S_{K-1}\,|S_0 \dots S_r, s_t = S_r \mathbf{X})$$

- For a recurrent network without feedback from the output we can make the conditional independence assumption:

$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = P(S_0 \dots S_r, s_t = S_r\,|\mathbf{X})P(s_{t+1} \in succ(S_r), succ(S_r), \dots, S_{K-1}\,|\mathbf{X})$$

Assuming past output symbols do not directly feed back into the net

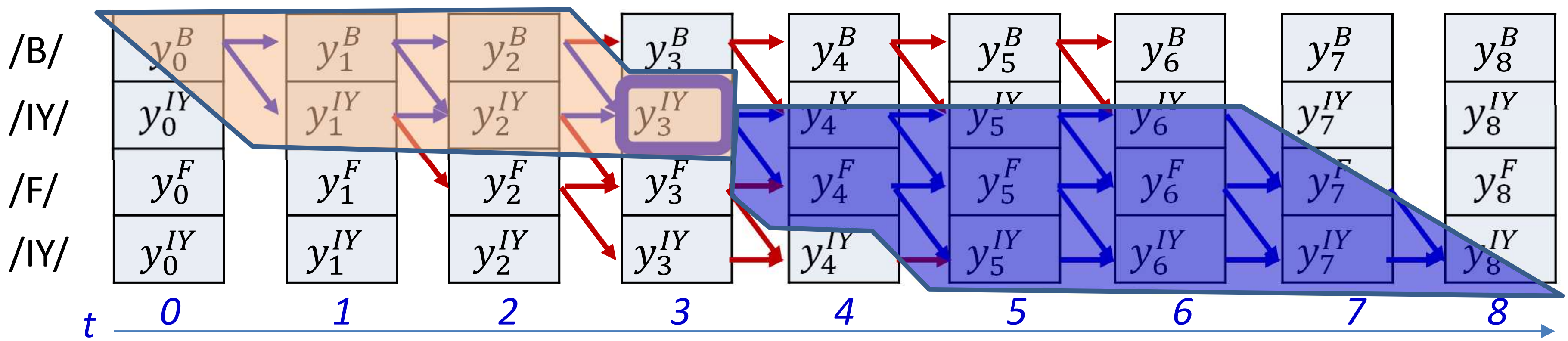
Conditional independence



- ***Dependency graph:*** Input sequence $\mathbf{X} = X_0\ X_1 \dots X_{N-1}$ governs hidden variables $\mathbf{H} = H_0\ H_1 \dots H_{N-1}$
- Hidden variables govern output predictions $y_0, y_1, \dots y_{N-1}$ individually
- $y_0, y_1, \dots y_{N-1}$ are conditionally independent given $\mathbf{H}$

- Since $\mathbf{H}$ is deterministically derived from $\mathbf{X}$, $y_0, y_1, \dots y_{N-1}$ are also conditionally independent given $\mathbf{X}$
  - This wouldn't be true if the relation between $\mathbf{X}$ and $\mathbf{H}$ were not deterministic or if $\mathbf{X}$ is unknown, or if the ys at any time went back into the net as inputs

A posteriori symbol probability

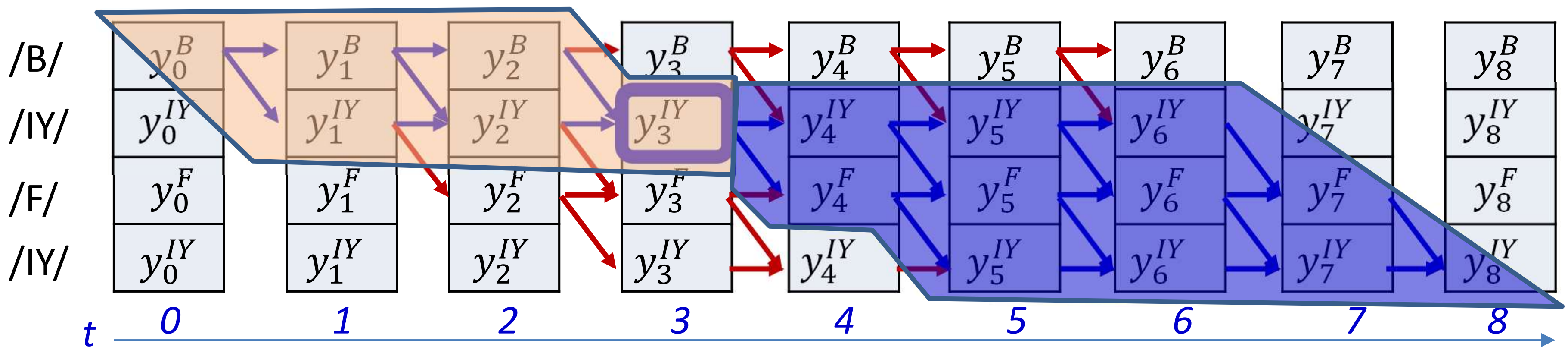


$P(s_t = S_r, \mathbf{S}|\mathbf{X})$

$= P(S_0 \ldots S_r, s_t = S_r\ |\mathbf{X})P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1}\ |\mathbf{X})$

- We will call the first term the *forward probability* $\alpha(t, r)$
- We will call the second term the *backward* probability $\beta(t, r)$

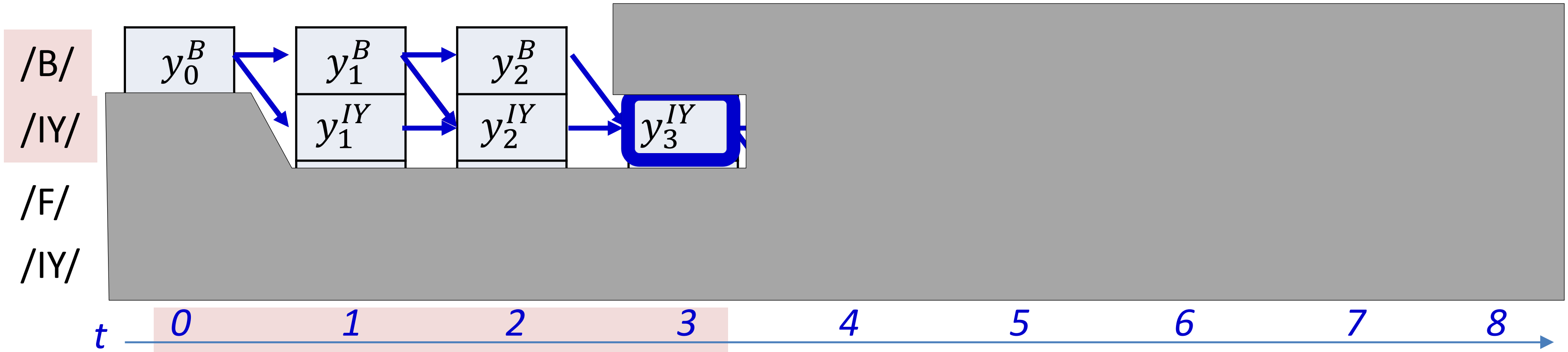
A posteriori symbol probability



$$P(s_t = S_r, \mathbf{S}|\mathbf{X})$$
$$= P(S_0 \ldots S_r, s_t = S_r \,|\mathbf{X}) P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1} \,|\mathbf{X})$$

- We will call the first term the *forward probability* $\alpha(t,r)$
- We will call the second term the *backward* probability $\beta(t,r)$

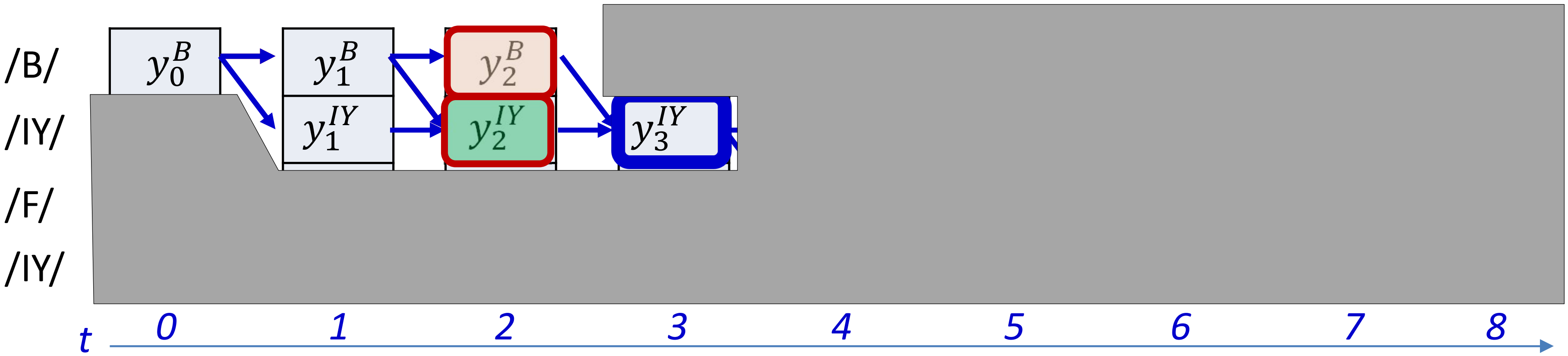
Computing $\alpha(t,r)$: Forward algorithm

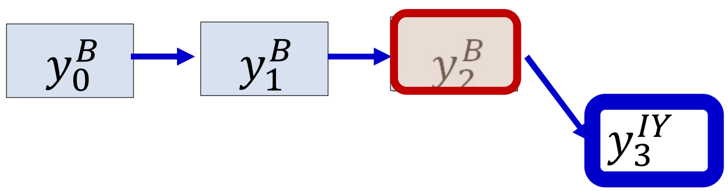


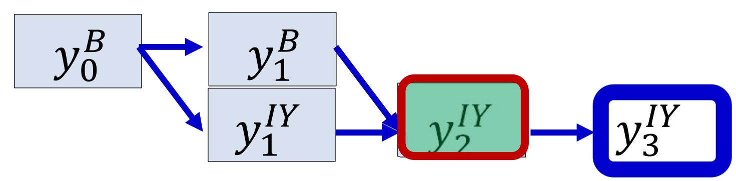
$$\alpha(t,r) = P(S_0..S_r, s_t = S_r|\mathbf{X})$$

- The $\alpha(t,r)$ is the total probability of the subgraph shown
  - The total probability of all paths leading to the alignment of S_r to time t

Computing $\alpha(t, r)$: Forward algorithm







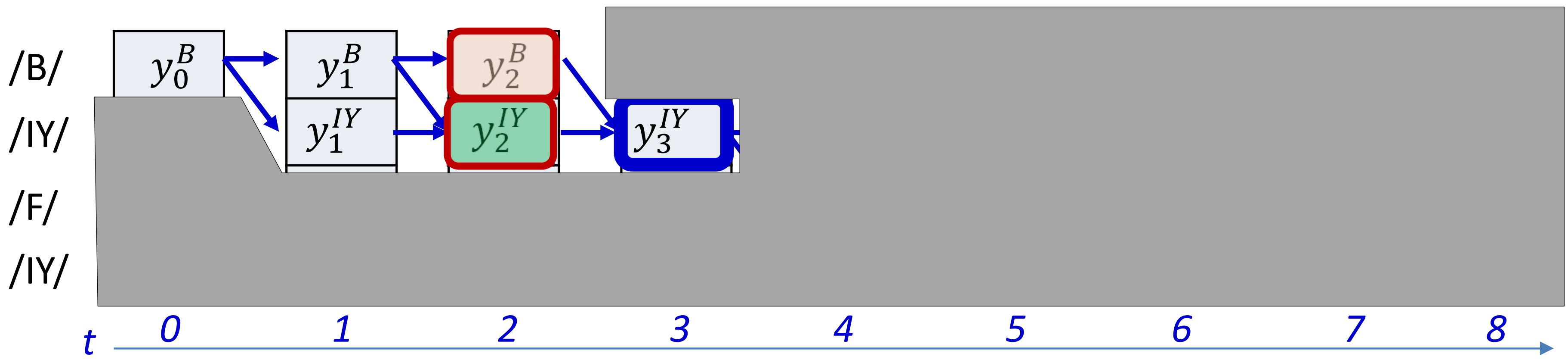
$$\alpha(3, IY) = P(S_0..S_r, s_t = S_r|\mathbf{X})$$

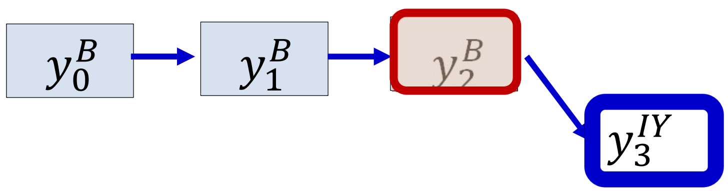
$$\alpha(3, IY) = P(subgraph\ ending\ at\ (2,B))y_3^{IY} + P(subgraph\ ending\ at\ (2,IY))y_3^{IY}$$

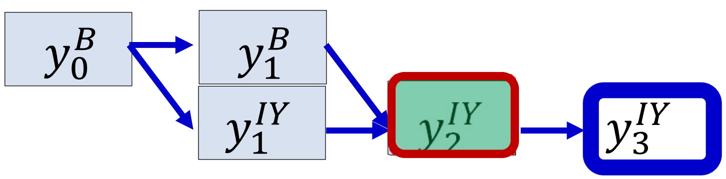
$$\alpha(t,r) = \sum_{q:S_q \in pred(S_r)} P(subgraph\ ending\ at\ (\mathrm{t}{-}1,q))Y_t^{S(r)}$$

- Where $pred(S_r)$ is any symbol that is permitted to come before an S_r and may include S_r
- q is its row index, and can take values r and $r-1$ in this example

Computing $\alpha(t,r)$: Forward algorithm







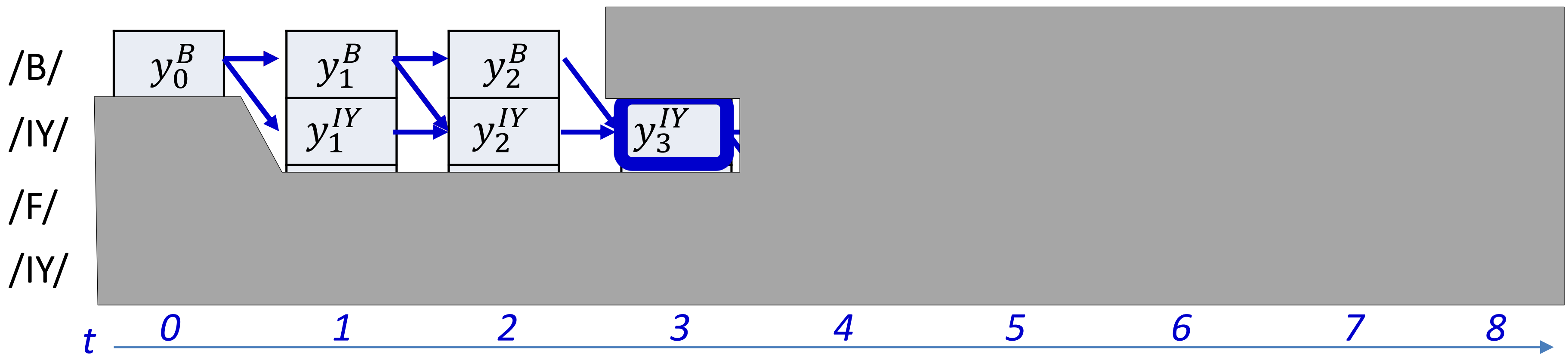
$$\alpha(t,r) = P(S_0..S_r, s_t = S_r|\mathbf{X})$$

$$\alpha(3,IY) = \alpha(2,B)y_3^{IY} + \alpha(2,IY)y_3^{IY}$$

$$\alpha(t,r) = \sum_{q:S_q \in pred(S_r)} \alpha(t-1,q)\, Y_t^{S(r)}$$

- Where $pred(S_r)$ is any symbol that is permitted to come before an S_r and may include S_r
- q is its row index, and can take values r and $r-1$ in this example

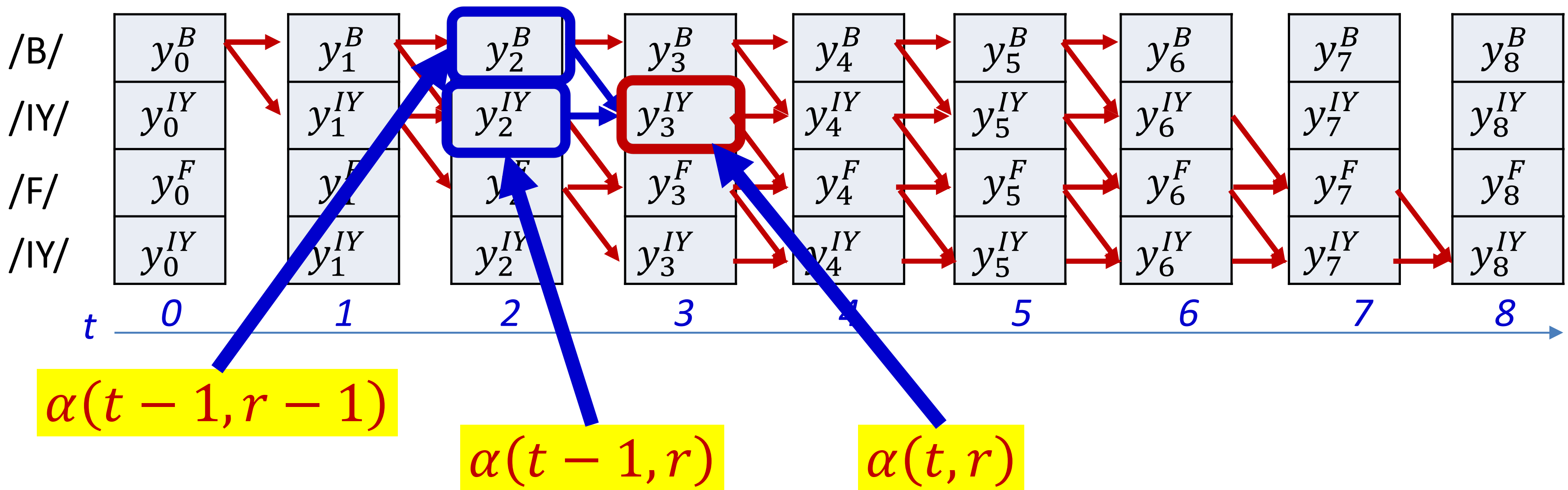
Forward algorithm



$$\alpha(t,r) = \sum_{q:S_q \in pred(S_r)} \alpha(t-1,q) y_t^{S_r}$$

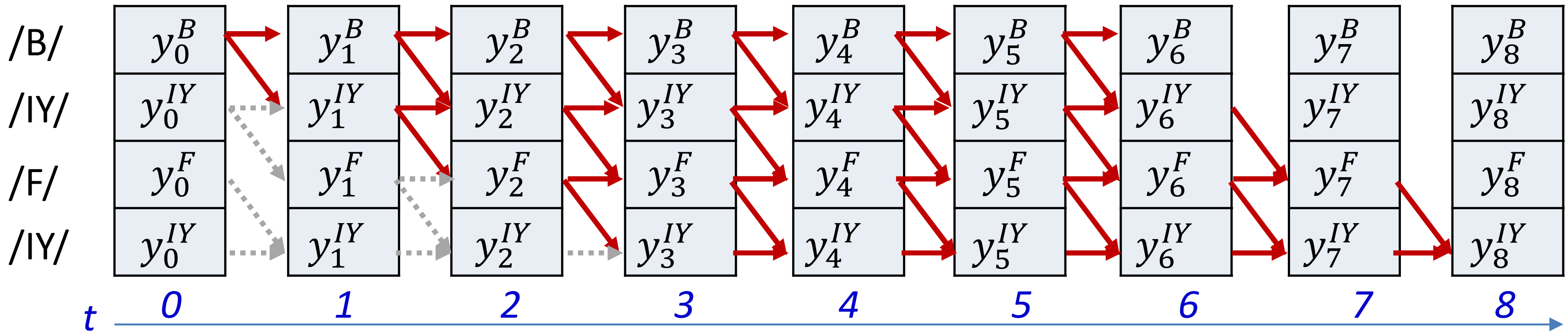
- The $\alpha(t,r)$ is the total probability of the subgraph shown

Forward algorithm



$$\alpha(t,r) = \big(\alpha(t-1,r) + \alpha(t-1,r-1)\big)y_t^{S(r)}$$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0, \; r > 0$$

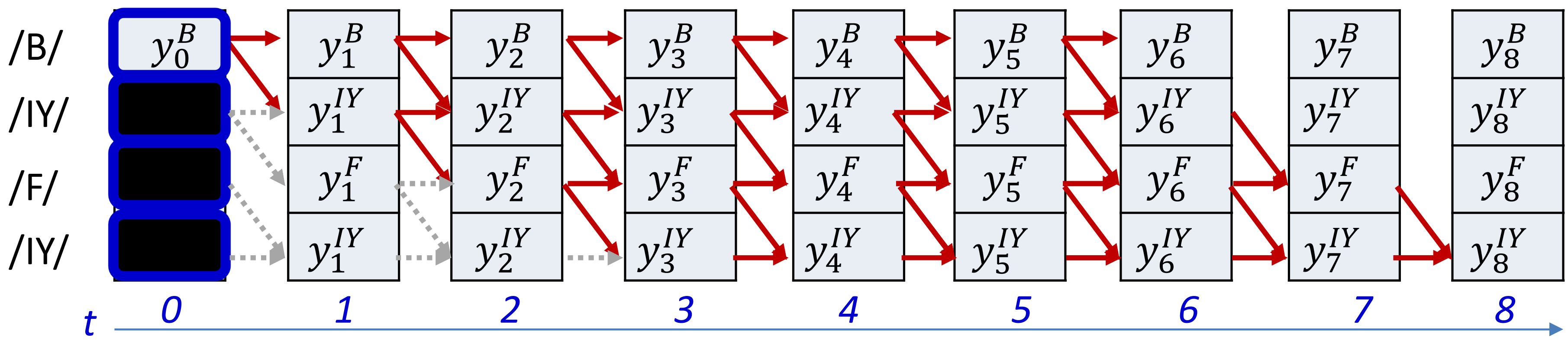
- for $t = 1 \ldots T-1$

$$\alpha(t,0) = \alpha(t-1,0) y_t^{S(0)}$$

for $l = 1 \ldots K-1$

$$\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1)) y_t^{S(l)}$$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0,\ r > 0$$

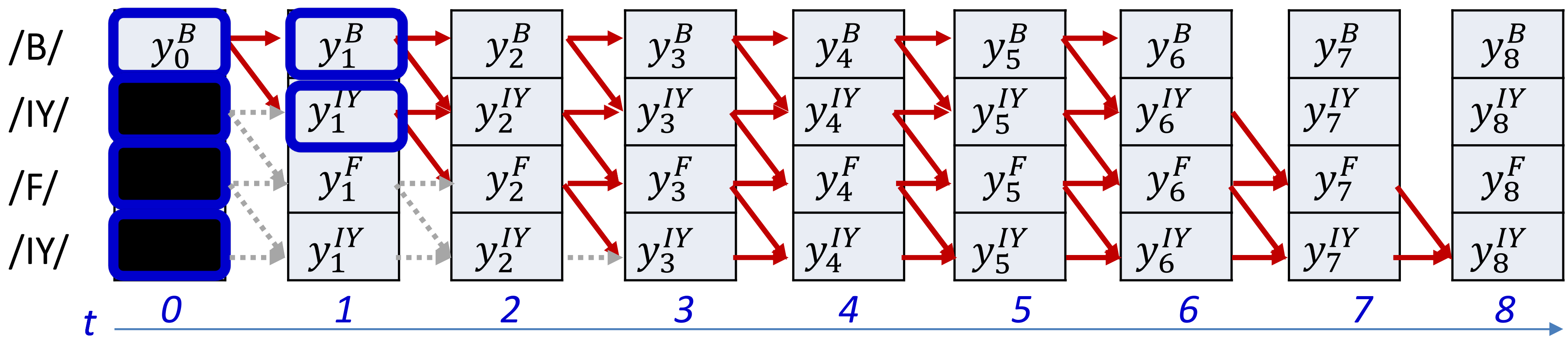
- for $t = 1 \ldots T-1$

$$\alpha(t,0) = \alpha(t-1,0) y_t^{S(0)}$$

for $l = 1 \ldots K-1$

- $\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1)) y_t^{S(l)}$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0, \ r > 0$$

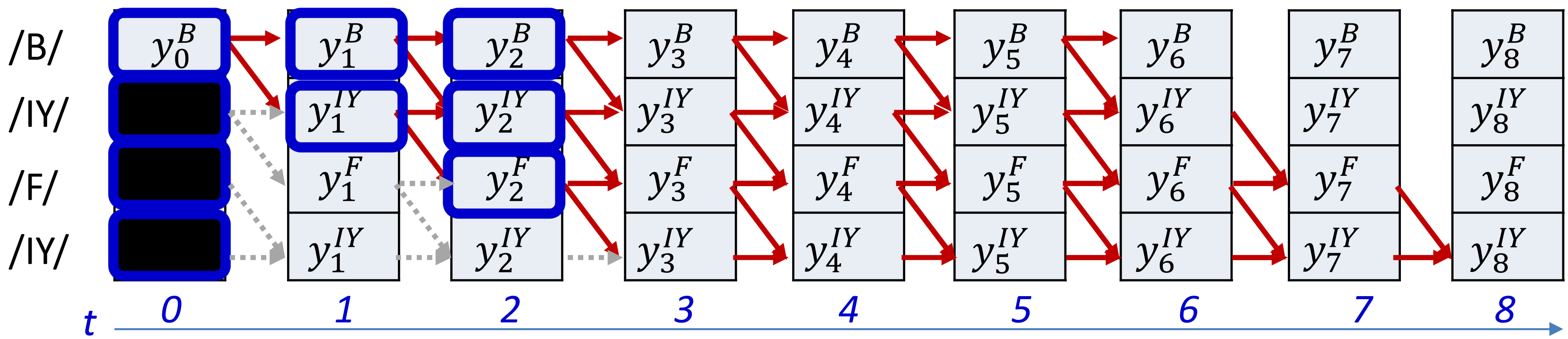
- for $t = 1 \ldots T-1$

$$\alpha(t,0) = \alpha(t-1,0)y_t^{S(0)}$$

for $l = 1 \ldots K-1$

  - $\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1))y_t^{S(l)}$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0,\ r > 0$$

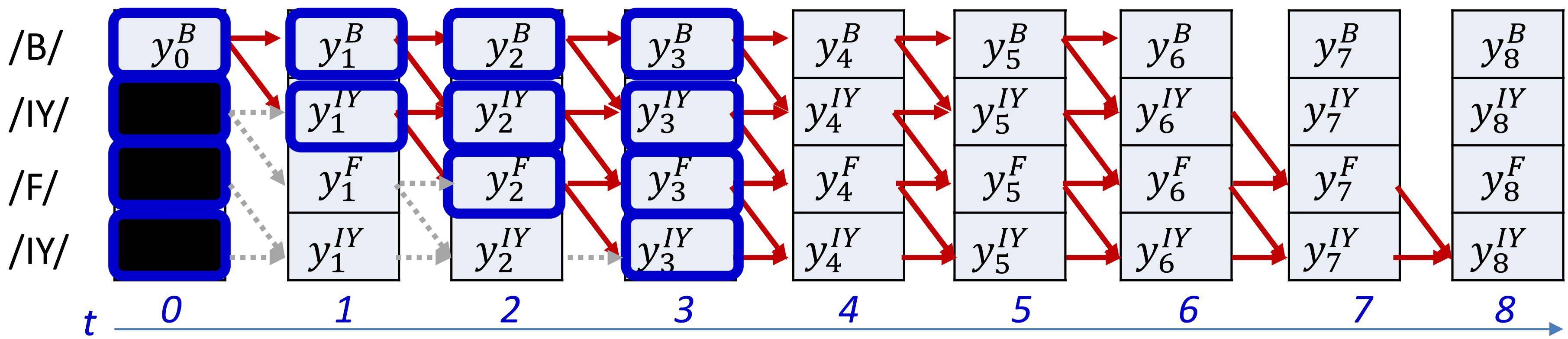
- for $t\ = \ 1 \dots T-1$

$$\alpha(t,0)\ = \alpha(t-1,0)y_t^{S(0)}$$

for $l\ = 1 \dots K-1$

- $\alpha(t,l)\ = (\alpha(t-1,l) + \alpha(t-1,l-1))y_t^{S(l)}$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0, \; r > 0$$

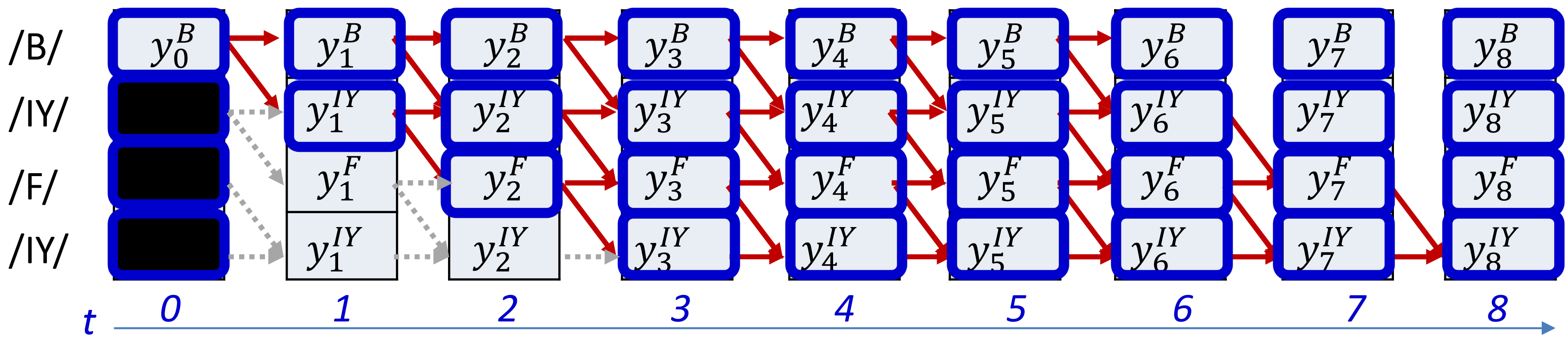
- for $t = 1 \ldots T-1$

$$\alpha(t,0) = \alpha(t-1,0) y_t^{S(0)}$$

for $l = 1 \ldots K-1$

  - $\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1)) y_t^{S(l)}$

Forward algorithm



- Initialization:

$$\alpha(0,0) = y_0^{S(0)}, \quad \alpha(0,r) = 0, \; r > 0$$

- for $t = 1 \dots T-1$

$$\alpha(t,0) = \alpha(t-1,0) y_t^{S(0)}$$

for $l = 1 \dots K-1$

- $\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1)) y_t^{S(l)}$

In practice..

- The recursion

  $$\alpha(t,l) = (\alpha(t-1,l) + \alpha(t-1,l-1))y_t^{S(l)}$$

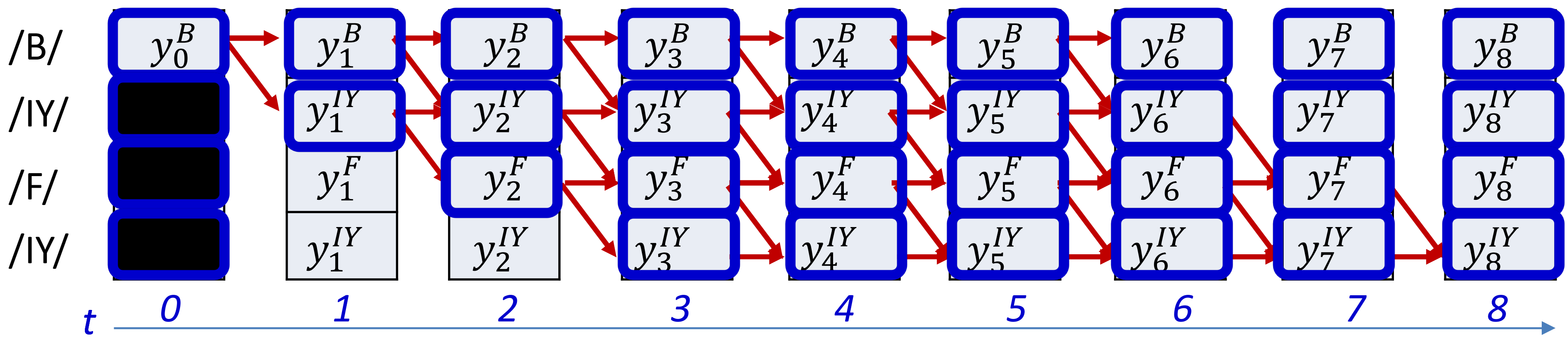
  will generally underflow

- Instead we can do it in the *log* domain

  $$\log \alpha(t,l) = \log(e^{\log \alpha(t-1,l)} + e^{\log \alpha(t-1,l-1)}) + \log y_t^{S(l)}$$

  – This can be computed entirely without underflow

Forward algorithm: Alternate statement



- The algorithm can also be stated as follows which separates the graph probability from the observation probability. This is needed to compute derivatives
- Initialization:

  $\hat{\alpha}(0,0) = 1, \;\; \hat{\alpha}(0,r) = 0, \;\; r > 0$

  $\alpha(0,r) = \hat{\alpha}(0,r)y_0^{S(r)}, \quad 0 \le r \le K-1$

- for $t \; = \; 1 \dots T-1$

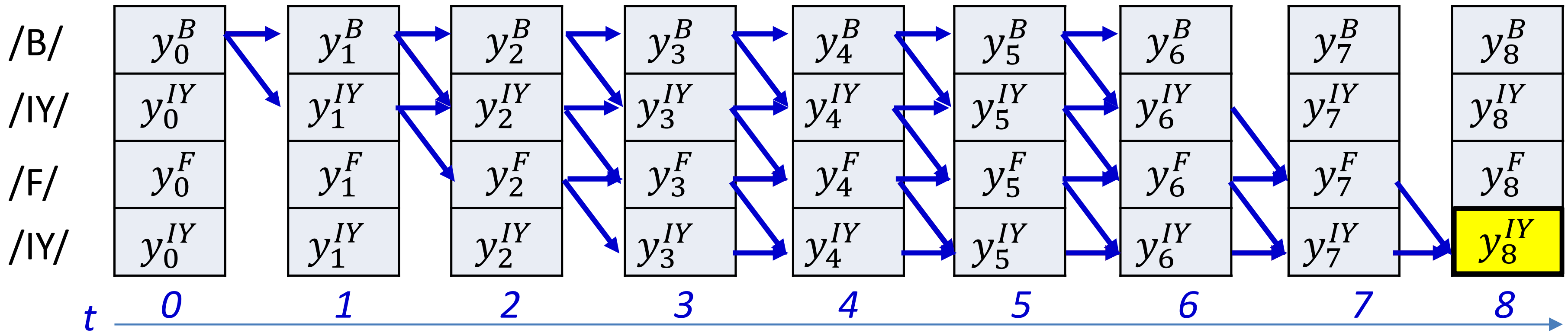
  $\hat{\alpha}(t,0) \; = \alpha(t-1,0)$

  for $l \;= 1 \dots K-1$

  - $\hat{\alpha}(t,l) \; = \alpha(t-1,l) + \alpha(t-1,l-1)$

  $\alpha(t,r) = \hat{\alpha}(t,r)y_t^{S(r)}, \quad 0 \le r \le K-1$

The final forward probability $\alpha(t, r)$



$$\alpha(T-1, K-1) = P(S_0 .. S_{K-1} | \mathbf{X})$$

- The probability of the entire symbol sequence is the alpha at the bottom right node

SIMPLE FORWARD ALGORITHM

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the output of the network for the ith symbol at time t
#T = length of input

#First create output table
For i = 1:N
    s(1:T,i) = y(1:T, S(i))

#The forward recursion
# First, at t = 1
alpha(1,1) = s(1,1)
alpha(1,2:N) = 0
for t = 2:T
    alpha(t,1) = alpha(t-1,1)*s(t,1)
    for i = 2:N
        alpha(t,i) = alpha(t-1,i-1) + alpha(t-1,i)
        alpha(t,i) *= s(t,i)
```

Can actually be done without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

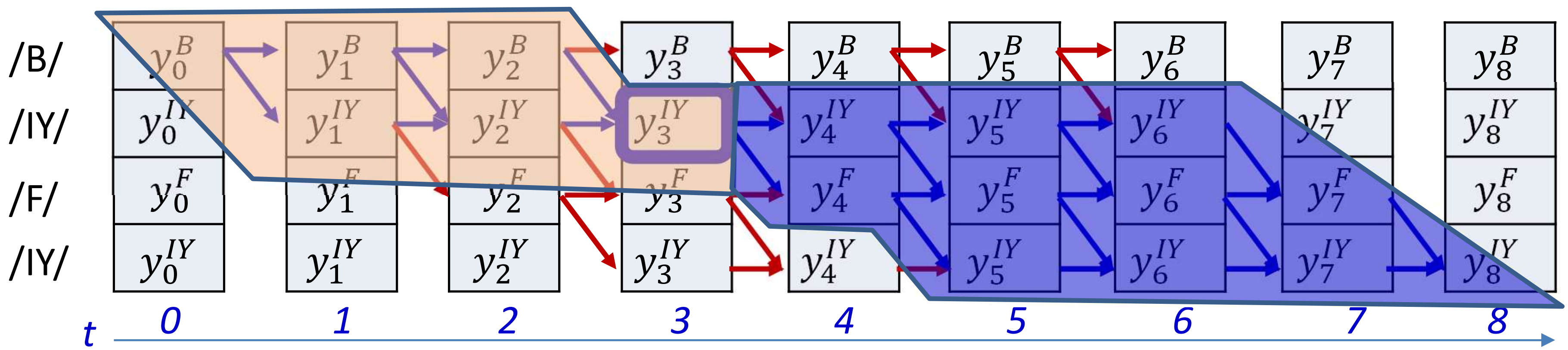
SIMPLE FORWARD ALGORITHM

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the network output for the ith symbol at time t
#T = length of input

#The forward recursion
# First, at t = 1
alpha(1,1) = y(1,S(1))
alpha(1,2:N) = 0
for t = 2:T
    alpha(t,1) = alpha(t-1,1)*y(t,S(1))
    for i = 2:N
        alpha(t,i) = alpha(t-1,i-1) + alpha(t-1,i)
        alpha(t,i) *= y(t,S(i))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

A posteriori symbol probability



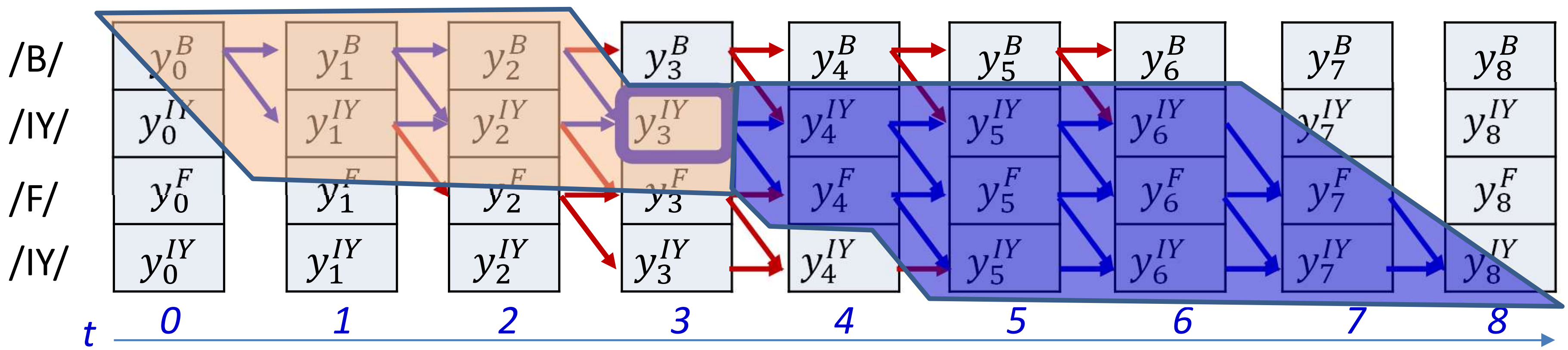
$P(s_t = S_r, \mathbf{S}|\mathbf{X})$

$= P(S_0 \ldots S_r, s_t = S_r\ |\mathbf{X}) P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1}\ |\mathbf{X})$

- We will call the first term the *forward probability* $\alpha(t, r)$
- We will call the second term the *backward* probability $\beta(t, r)$

We have seen how to compute this

A posteriori symbol probability

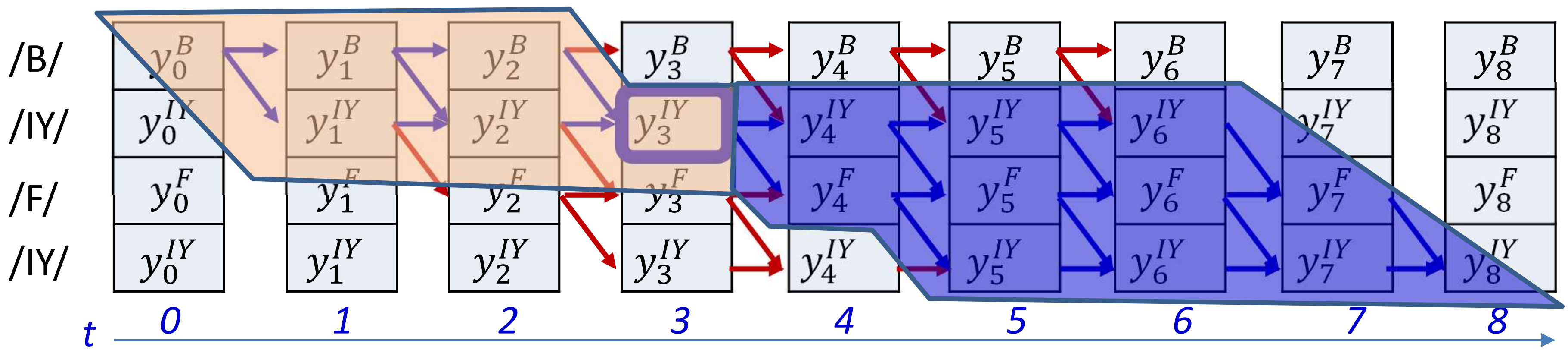


$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r) P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1} |\mathbf{X})$$

- We will call the first term the *forward probability* $\alpha(t,r)$
- We will call the second term the *backward* probability $\beta(t,r)$

We have seen how to compute this

A posteriori symbol probability

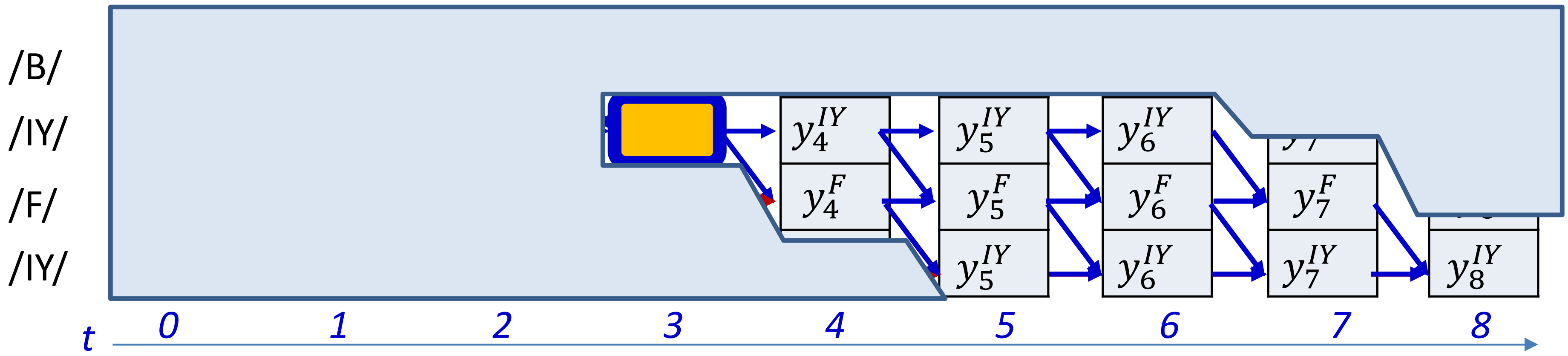


$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r) P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1} |\mathbf{X})$$

- We will call the first term the *forward probability* $\alpha(t,r)$
- We will call the second term the *backward* probability $\beta(t,r)$

Lets look at this

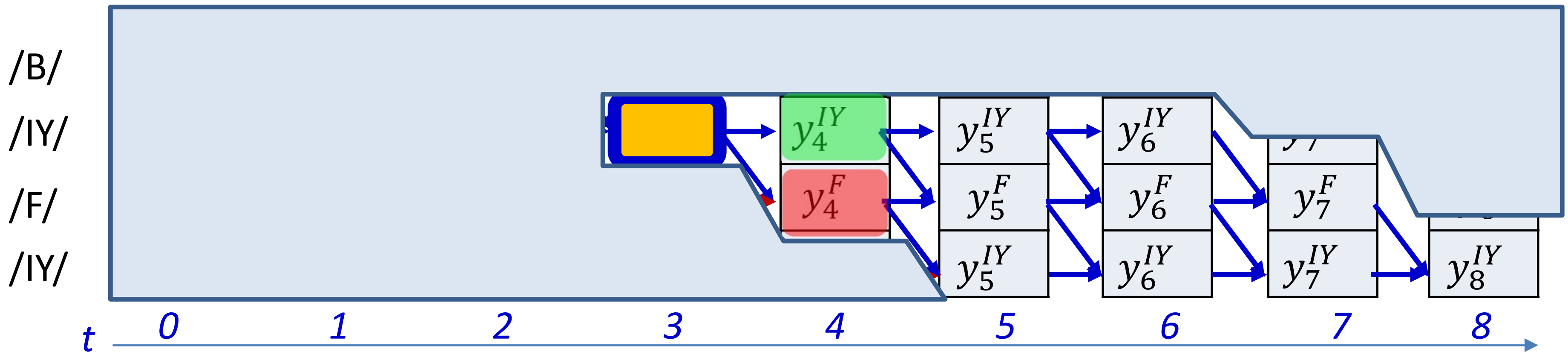
Bacward probability

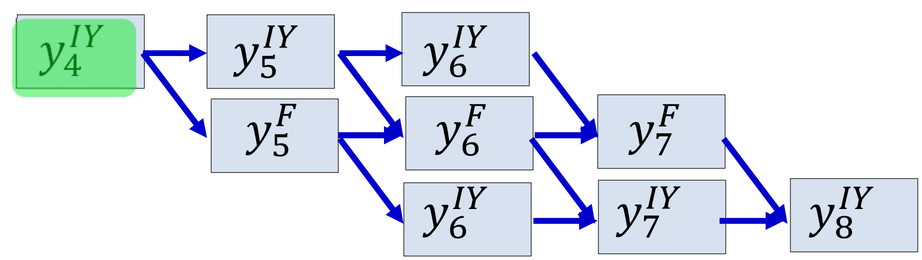


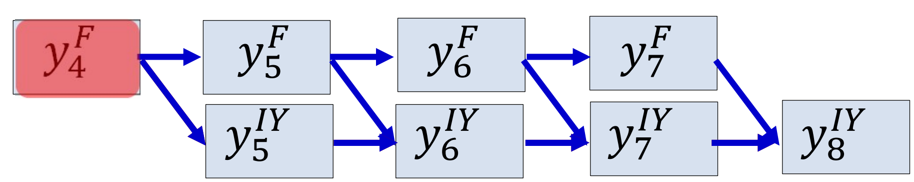
$$\beta(t,r) = P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1} \,|\mathbf{X})$$

- $\beta(t,r)$ is the probability of the exposed subgraph, not including the orange shaded box

Backward probability

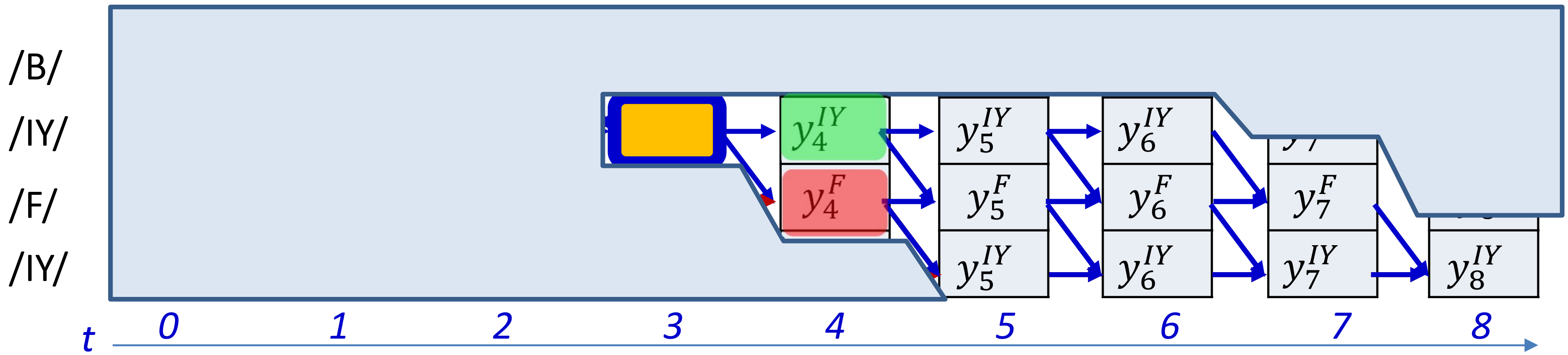


$\beta(3,1) = P\Big($ 

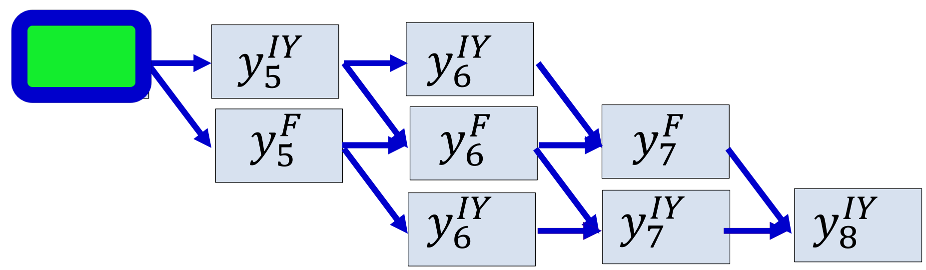
$\Big) + P\Big($ 

$\Big)$

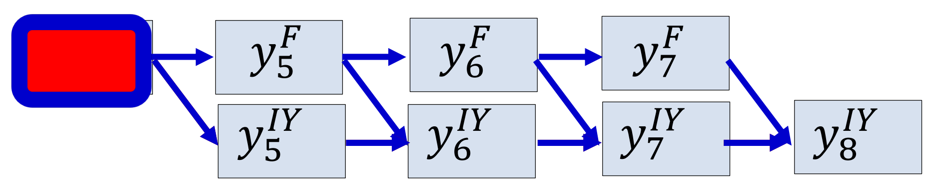
Backward probability



$$\beta(3,1) = y_4^{IY} P\Big($$

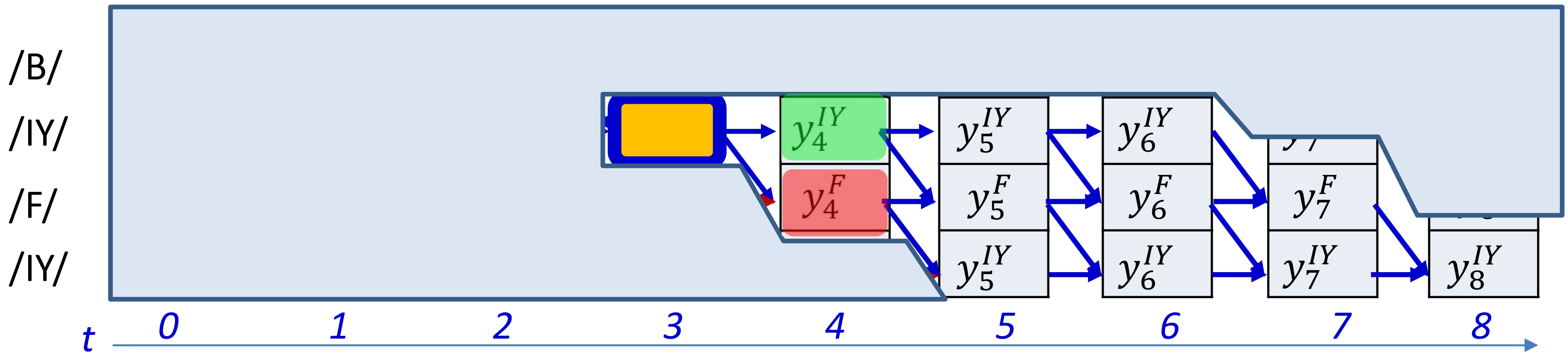


$$\Big) + y_4^F P\Big($$



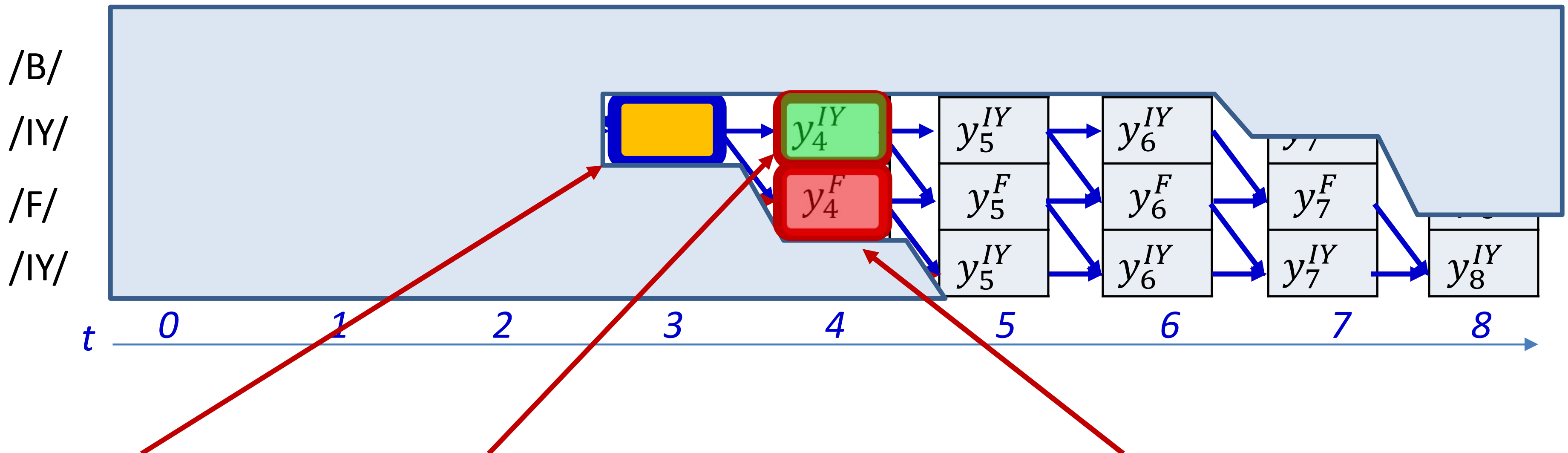
$$\Big)$$

Backward probability



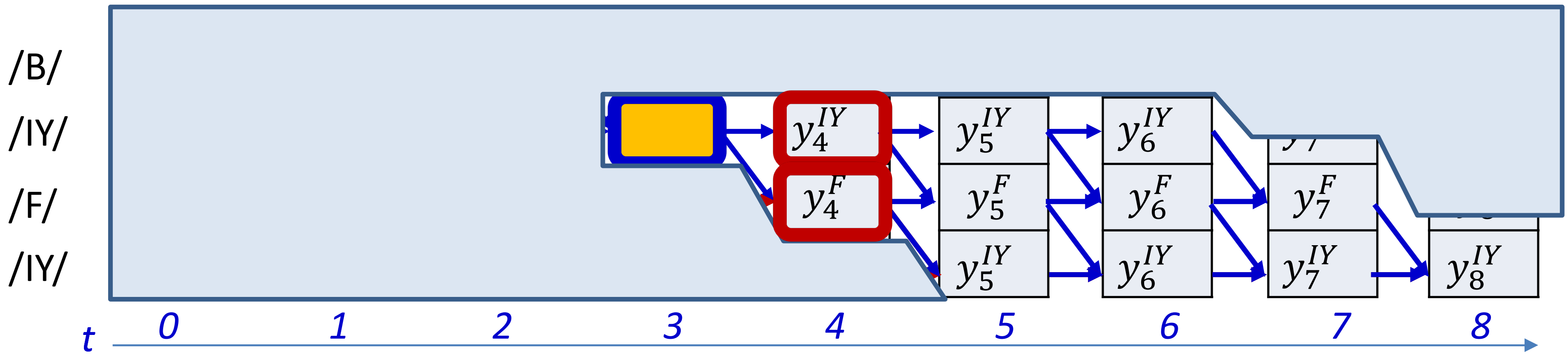
$$\beta(3,1) = y_4^{IY}\beta(4,1) + y_4^{F}\beta(4,2)$$

Backward algorithm



$$\beta(t,r) = y_{t+1}^{S(r)}\beta(t+1,r) \; + \; y_{t+1}^{S(r+1)}\beta(t+1,r+1)$$

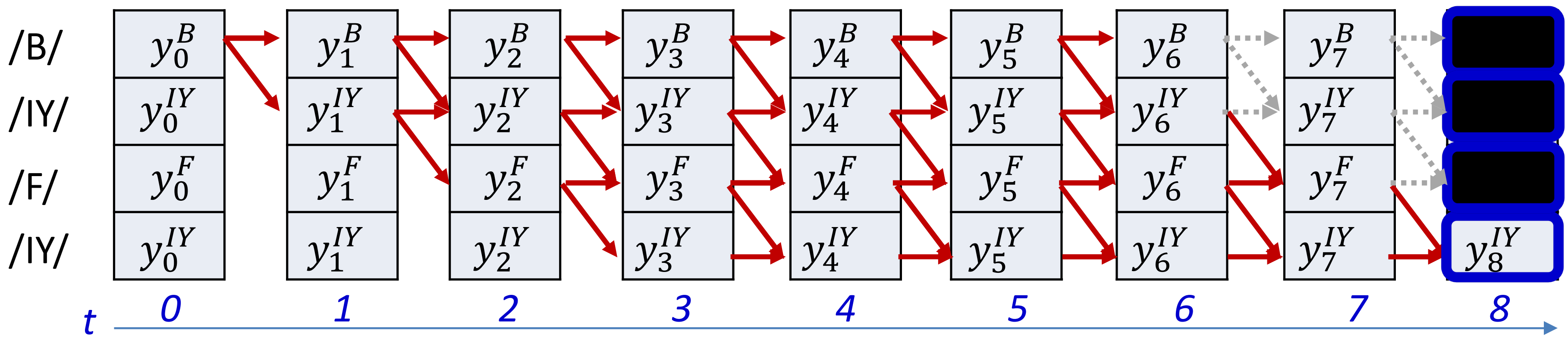
Backward algorithm



$$\beta(t,r) = \sum_{q:S_q \in succ(S_r)} \beta(t+1,q) y_{t+1}^{S_q}$$

- The $\beta(t,r)$ is the total probability of the subgraph shown
- The $\beta(t,r)$ terms at any time t are defined recursively in terms of the $\beta(t+1,q)$ terms at the next time

Backward algorithm



- Initialization:

$$\beta(T-1, K-1) = 1,\ \ \beta(T-1, r) = 0,\ \ r < K-1$$

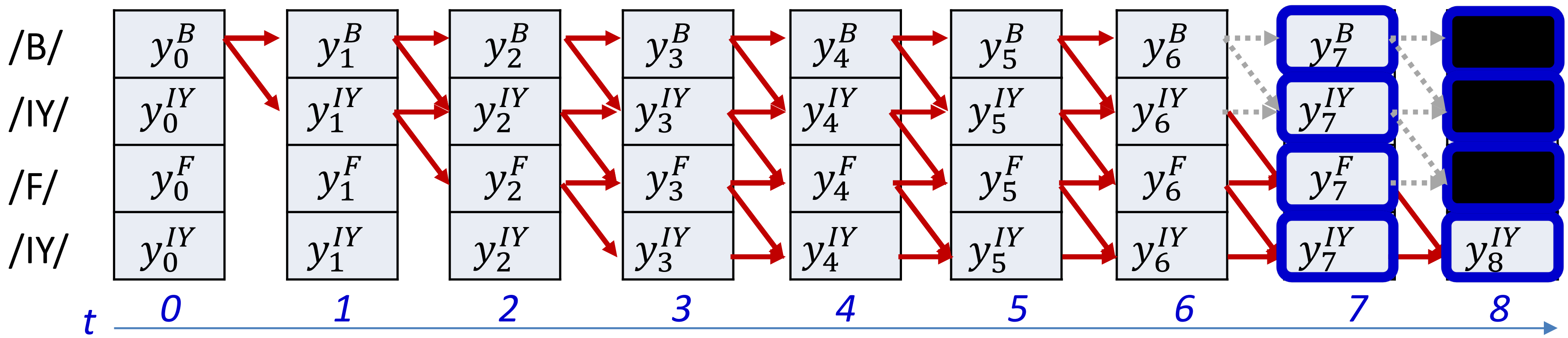
- for $t\ = T - 2\ downto\ 0$

$$\beta(t, K)\ = \beta(t+1, K) y_{t+1}^{S(K)}$$

for $r\ = K - 2 \ldots 0$

- $\beta(t, r) = y_{t+1}^{S(l)} \beta(t+1, r)\ +\ y_{t+1}^{S(r+1)} \beta(t+1, r+1)$

Backward algorithm



- Initialization:

$$\beta(T-1, K-1) = 1, \;\; \beta(T-1, r) = 0, \;\; r < K-1$$

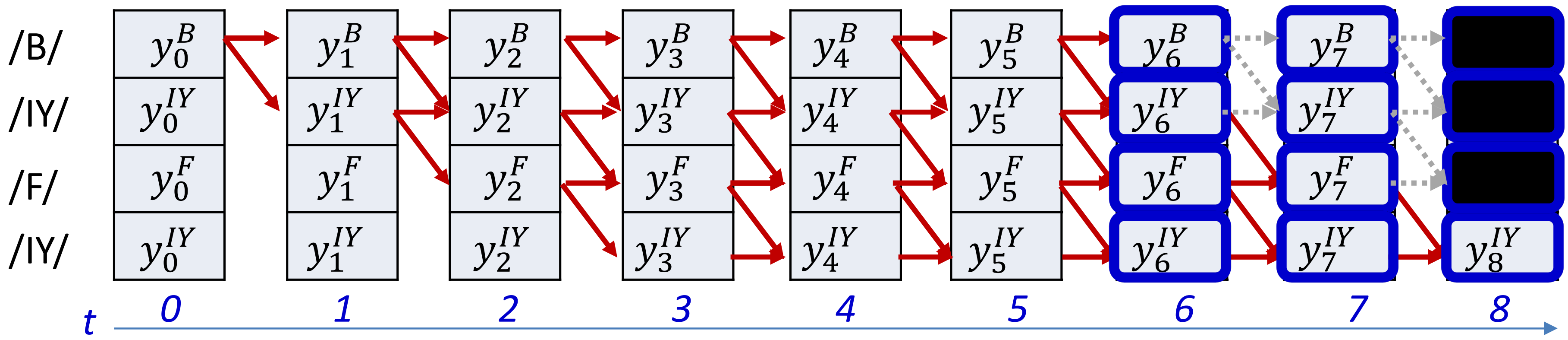
- for $t \;= T-2 \; downto \; 0$

$$\beta(t, K) \;= \beta(t+1, K) y_{t+1}^{S(K)}$$

for $r \;= K-2 \ldots 0$

- $\beta(t, r) = y_{t+1}^{S(l)} \beta(t+1, r) \; + \; y_{t+1}^{S(r+1)} \beta(t+1, r+1)$

Backward algorithm



- Initialization:

$$\beta(T-1, K-1) = 1,\ \ \beta(T-1, r) = 0,\ \ r < K-1$$

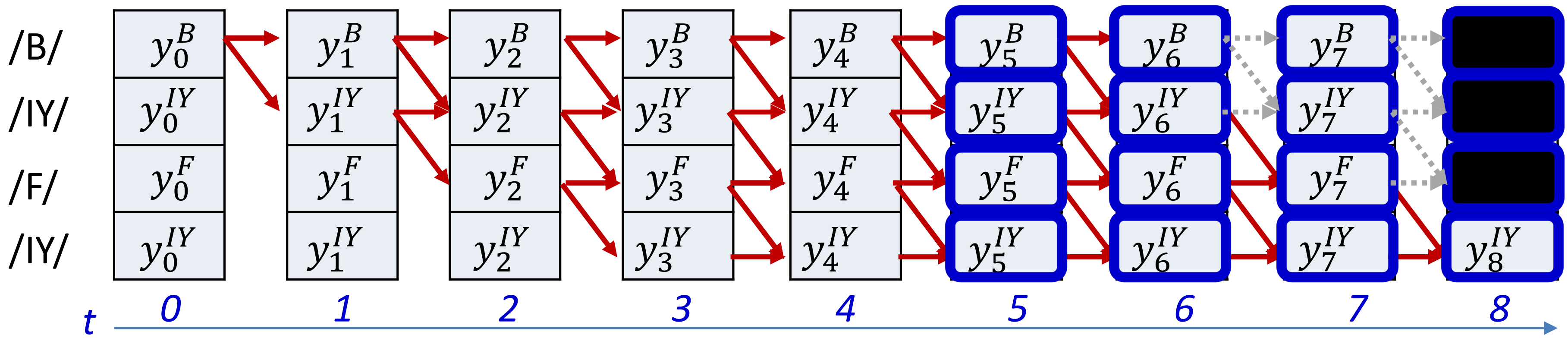
- for $t\ = T-2\ downto\ 0$

$$\beta(t, K)\ = \beta(t+1, K) y_{t+1}^{S(K)}$$

for $r\ = K-2 \ldots 0$

- $\beta(t, r) = y_{t+1}^{S(l)} \beta(t+1, r)\ +\ y_{t+1}^{S(r+1)} \beta(t+1, r+1)$

Backward algorithm



- Initialization:

$$\beta(T-1, K-1) = 1, \;\; \beta(T-1, r) = 0, \;\; r < K-1$$

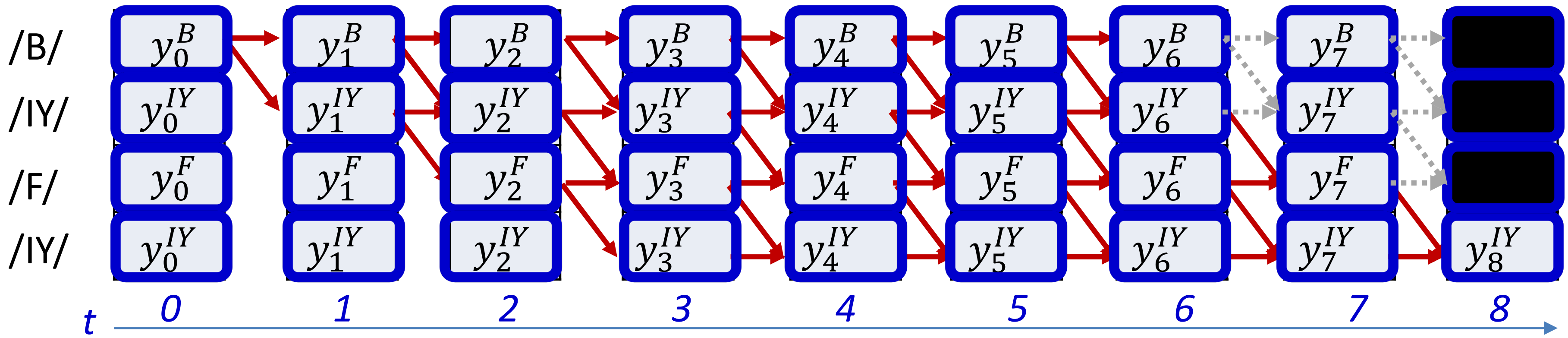
- for $t \; = T - 2 \; downto \; 0$

$$\beta(t, K) \; = \beta(t+1, K) y_{t+1}^{S(K)}$$

for $r \; = K - 2 \ldots 0$

- $\beta(t, r) = y_{t+1}^{S(l)} \beta(t+1, r) \; + \; y_{t+1}^{S(r+1)} \beta(t+1, r+1)$

Backward algorithm



- Initialization:

$$\beta(T-1, K-1) = 1,\ \beta(T-1, r) = 0,\ r < K-1$$

- for $t\ = T-2\ downto\ 0$

$$\beta(t, K)\ = \beta(t+1, K) y_{t+1}^{S(K)}$$

for $r\ = K-2 \ldots 0$

- $\beta(t, r) = y_{t+1}^{S(l)} \beta(t+1, r)\ +\ y_{t+1}^{S(r+1)} \beta(t+1, r+1)$

SIMPLE BACKWARD ALGORITHM

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the output of the network for the ith symbol at time t
#T = length of input

#First create output table
For i = 1:N
    s(1:T,i) = y(1:T, S(i))

#The backward recursion
# First, at t = T
beta(T,N) = 1
beta(T,1:N-1) = 0
for t = T-1 downto 1
    beta(t,N) = beta(t+1,N)*s(t+1,N)
    for i = N-1 downto 1
          beta(t,i) = beta(t+1,i)*s(t+1,i) + beta(t+1,i+1))*s(t+1,i+1)
```

Can actually be done without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

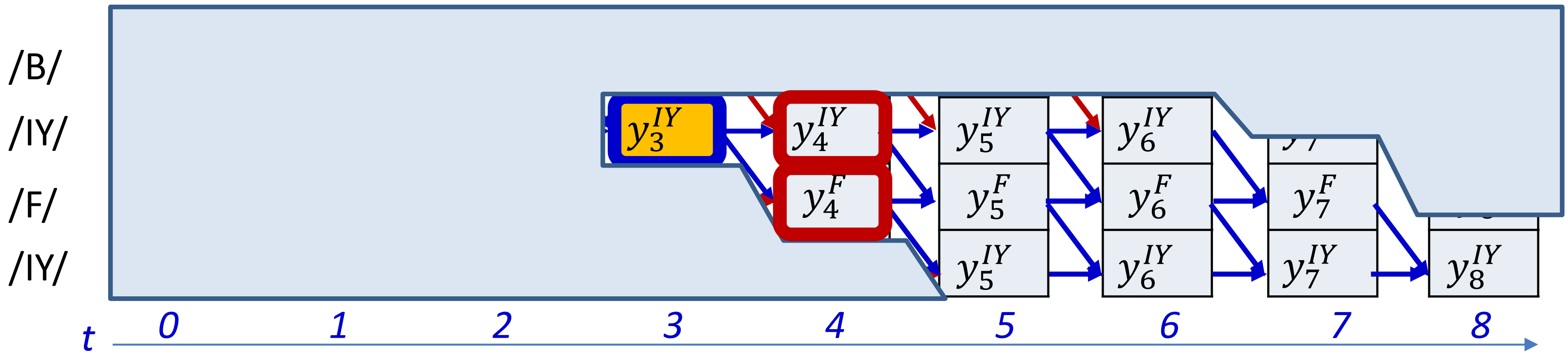
BACKWARD ALGORITHM

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the output of the network for the ith symbol at time t
#T = length of input

#The backward recursion
# First, at t = T
beta(T,N) = 1
beta(T,1:N-1) = 0
for t = T-1 downto 1
    beta(t,N) = beta(t+1,N)*y(t+1,S(N))
    for i = N-1 downto 1
          beta(t,i) = beta(t+1,i)*y(t+1,S(i)) + beta(t+1,i+1))*y(t+1,S(i+1))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

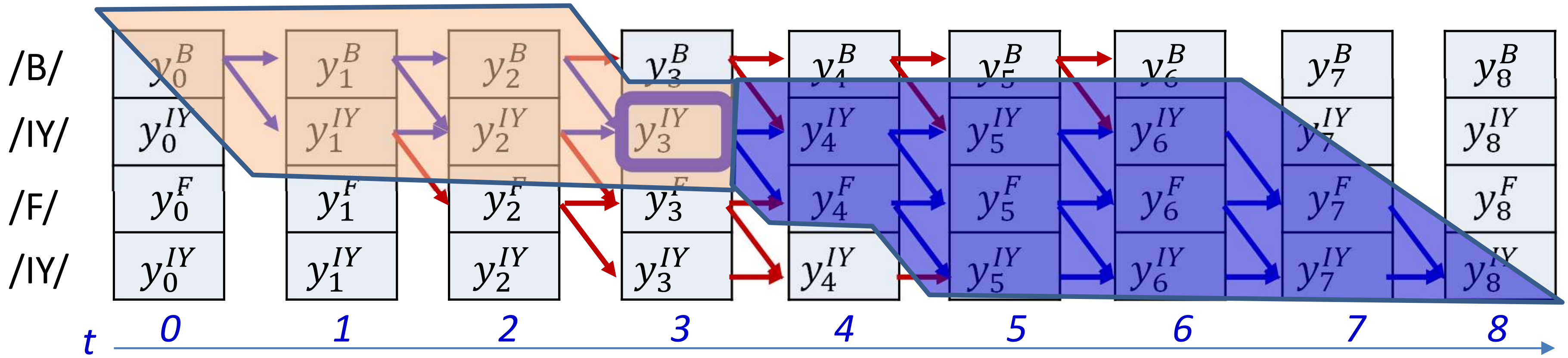
Alternate Backward algorithm



$$\hat{\beta}(t,r) = y_t^{S(r)}\left(\hat{\beta}(t+1,r) + \hat{\beta}(t+1,r+1)\right)$$

- Some implementations of the backward algorithm will use the above formula
- Note that here the probability of the observation at t is also factored into beta
- It will have to be unfactored later (we'll see how)

The joint probability

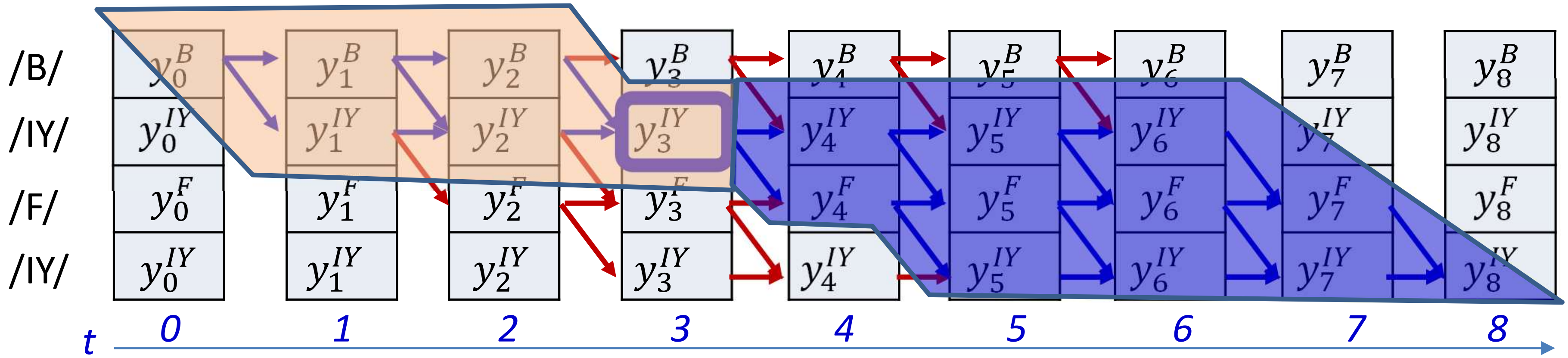


$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r)P(s_{t+1} \in succ(S_r), succ(S_r), \ldots, S_{K-1}\,|\mathbf{X})$$

- We will call the first term the *forward probability* $\alpha(t,r)$
- We will call the second term the *backward* probability $\beta(t,r)$

We now can compute this

The joint probability



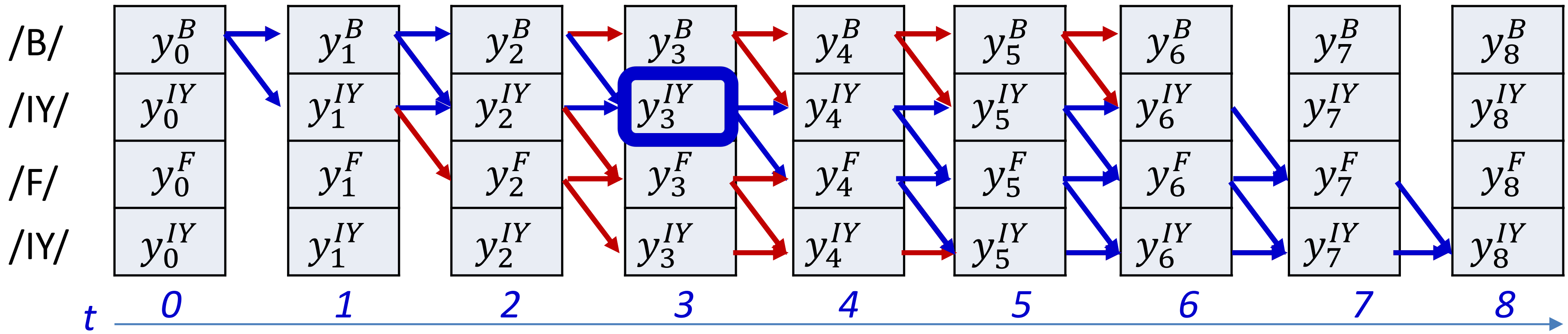
$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r)\beta(t,r)$$

- We will call the first term the *forward probability* $\alpha(t,r)$
- We will call the second term the *backward probability* $\beta(t,r)$

Forward algo

Backward algo

The posterior probability

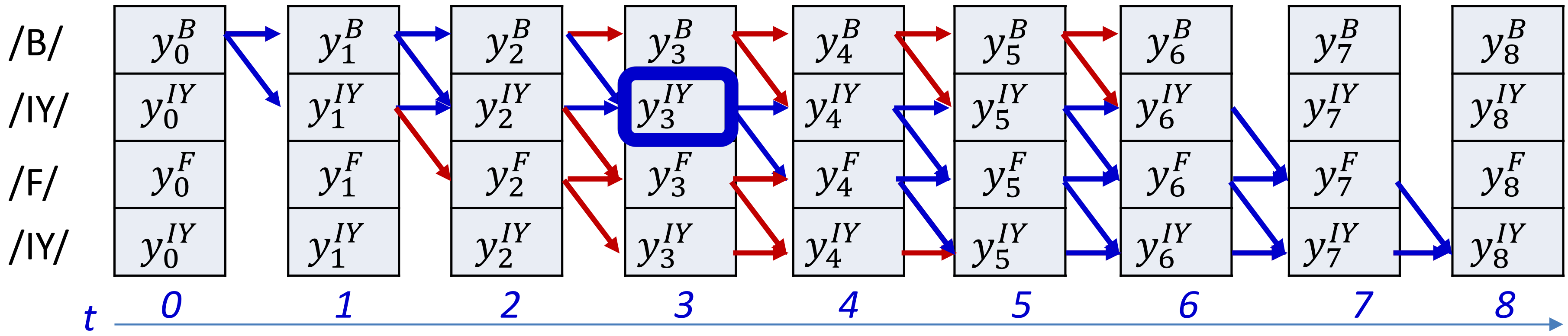


$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r)\beta(t,r)$$

- The *posterior* is given by

$$P(s_t = S_r|\mathbf{S}, \mathbf{X}) = \frac{P(s_t = S_r, \mathbf{S}|\mathbf{X})}{\sum_{S'_r} P(s_t = S'_r, \mathbf{S}|\mathbf{X})} = \frac{\alpha(t,r)\beta(t,r)}{\sum_{r'} \alpha(t,r')\beta(t,r')}$$

The posterior probability



- Let the posterior $P(s_t = S_r | \mathbf{S}, \mathbf{X})$ be represented by $\gamma(t, r)$

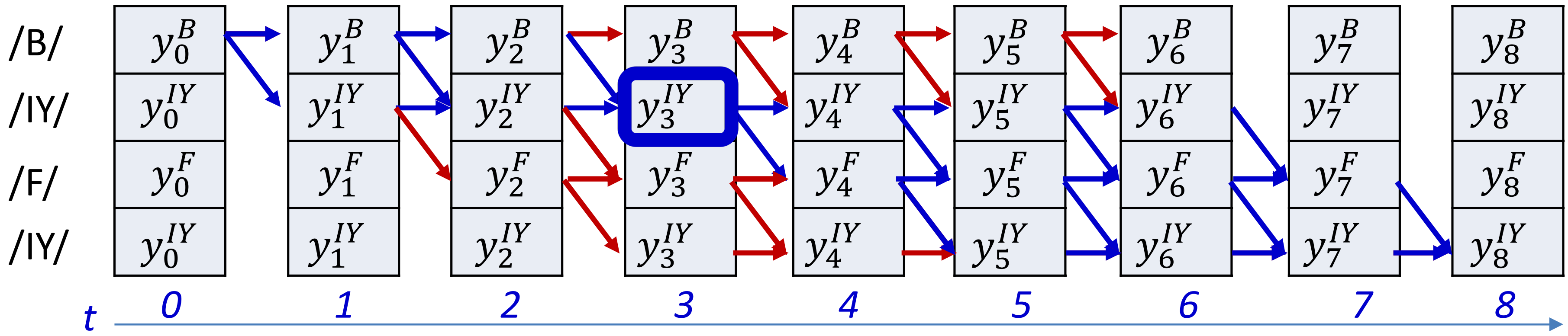
$$\gamma(t, r) = \frac{\alpha(t, r)\beta(t, r)}{\sum_{r'} \alpha(t, r')\beta(t, r')}$$

COMPUTING POSTERIORS

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the output of the network for the ith symbol at time t
#T = length of input

#Assuming the forward are completed first
alpha = forward(y, S)   # forward probabilities computed
beta  = backward(y, S)  # backward probabilities computed

#Now compute the posteriors
for t = 1:T
    sumgamma(t) = 0
    for i = 1:N
        gamma(t,i) = alpha(t,i) * beta(t,i)
        sumgamma(t) += gamma(t,i)
    end
    for i=1:N
        gamma(t,i) = gamma(t,i) / sumgamma(t)
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

The posterior probability



$$P(s_t = S_r, \mathbf{S}|\mathbf{X}) = \alpha(t,r)\beta(t,r)$$

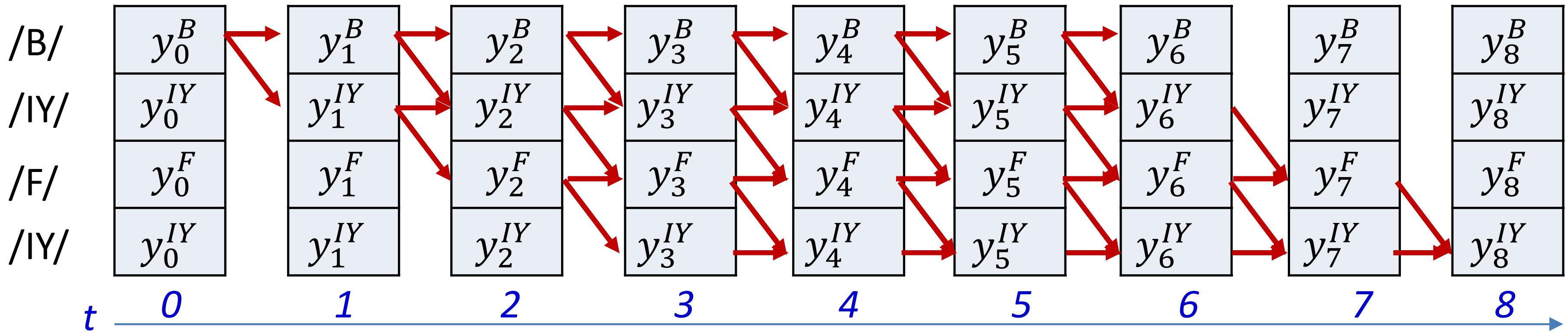
- The *posterior* is given by

$$\gamma(t,r) = \frac{\alpha(t,r)\beta(t,r)}{\sum_{r'} \alpha(t,r')\beta(t,r')}$$

- We can also write this using the modified beta formula as (you will see this in papers)

$$\gamma(t,r) = \frac{\frac{1}{y_t^{S(r)}}\alpha(t,r)\hat{\beta}(t,r)}{\sum_{r'} \frac{1}{y_t^{S(r)}}\alpha(t,r)\hat{\beta}(t,r)}$$

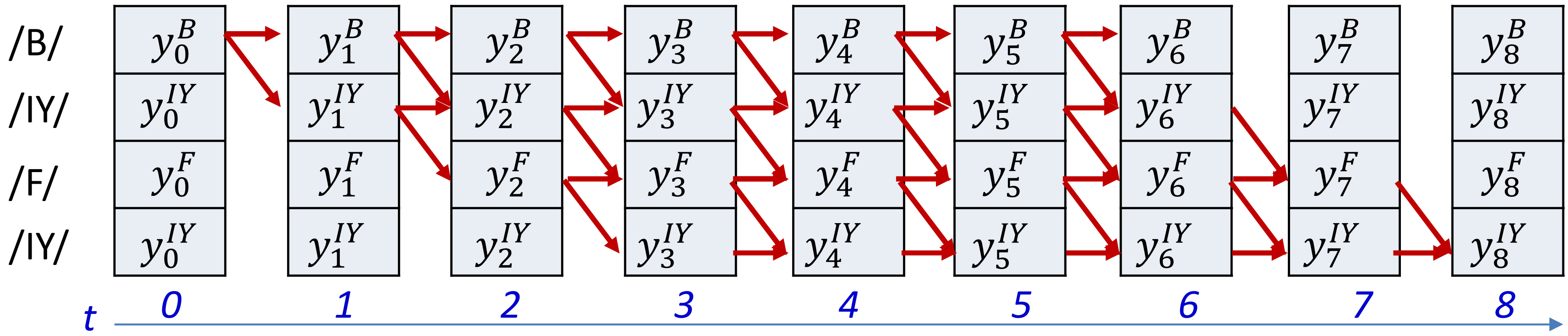
The expected divergence



$$DIV = -\sum_t \sum_{s \in S_0 \ldots S_{K-1}} P(s_t = s | \mathbf{S}, \mathbf{X}) \log Y(t, s_t = s)$$

$$DIV = -\sum_t \sum_r \gamma(t, r) \log y_t^{S(r)}$$

The expected divergence



$$DIV = -\sum_t \sum_{s \in S_0 \ldots S_{K-1}} P(s_t = s | \mathbf{S}, \mathbf{X}) \log Y(t, s_t = s)$$

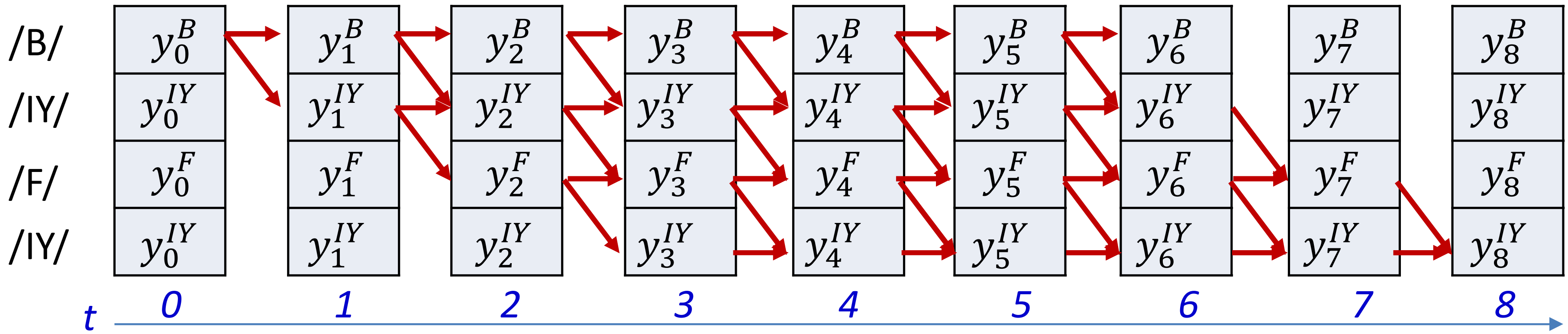
$$DIV = -\sum_t \sum_r \gamma(t, r) \log y_t^{S(r)}$$

- The derivative of the divergence w.r.t the output Y_t of the net at any time:

$$\nabla_{Y_t} DIV = \begin{bmatrix} \frac{dDIV}{dy_t^{S_0}} & \frac{dDIV}{dy_t^{S_1}} & \cdots & \frac{dDIV}{dy_t^{S_{L-1}}} \end{bmatrix}$$

  – Components will be non-zero only for symbols that occur in the training instance

The expected divergence



$$DIV = -\sum_{t} \sum_{s \in S_0 \ldots S_{K-1}} P(s_t = s | \mathbf{S}, \mathbf{X}) \log Y(t, s_t = s)$$

$$DIV = -\sum_{t} \sum_{r} \gamma(t, r) \log y_t^{S(r)}$$

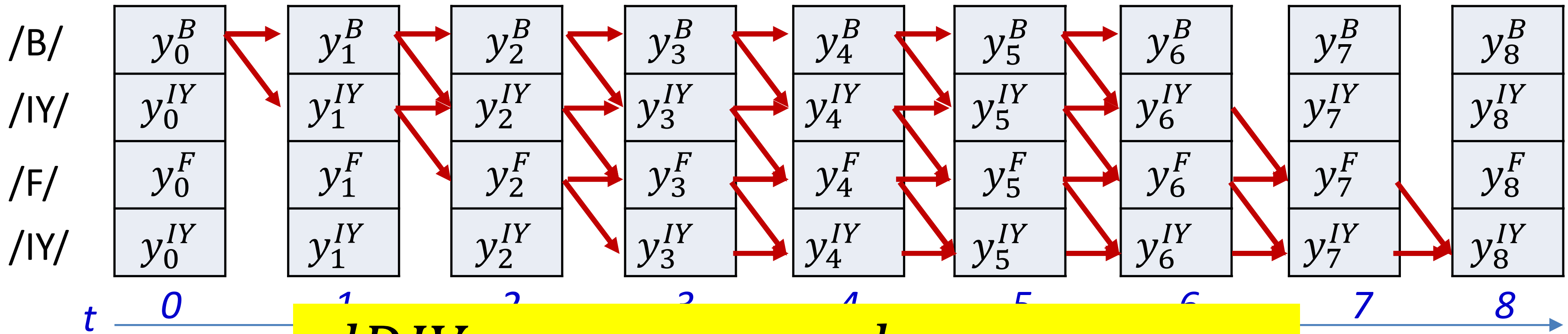
- The derivative of the divergence w.r.t the output Y_t of the net at any time:

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^{S_0}} \; \frac{dDIV}{dy_t^{S_1}} \; \cdots \right]$$

Must compute these terms from here

  – Components will be non-zero only for symbols that occur in the training instance

The expected divergence



$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{d}{dy_t^l} \gamma(t,r) \log y_t^l$$

$$DIV = -\sum_t \sum_r \gamma(t,r) \log y_t^{S(r)}$$

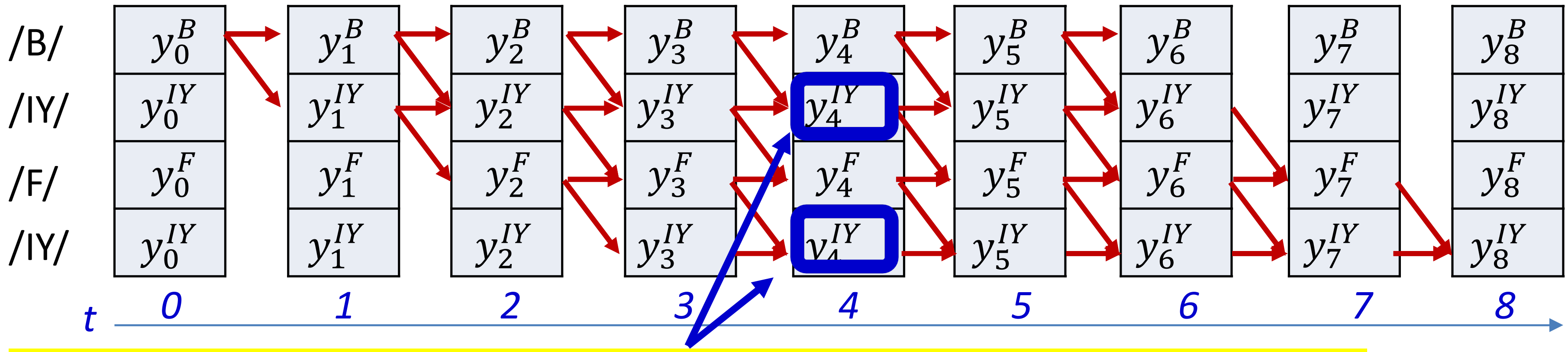
- The derivative of the divergence w.r.t the output Y_t of the net at any time:

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^{S_0}} \quad \frac{dDIV}{dy_t^{S_1}} \quad \cdots \right]$$

Must compute these terms from here

  – Components will be non-zero only for symbols that occur in the training instance

The expected divergence



The derivatives at both these locations must be summed to get $\frac{dDIV}{dy_4^{IY}}$

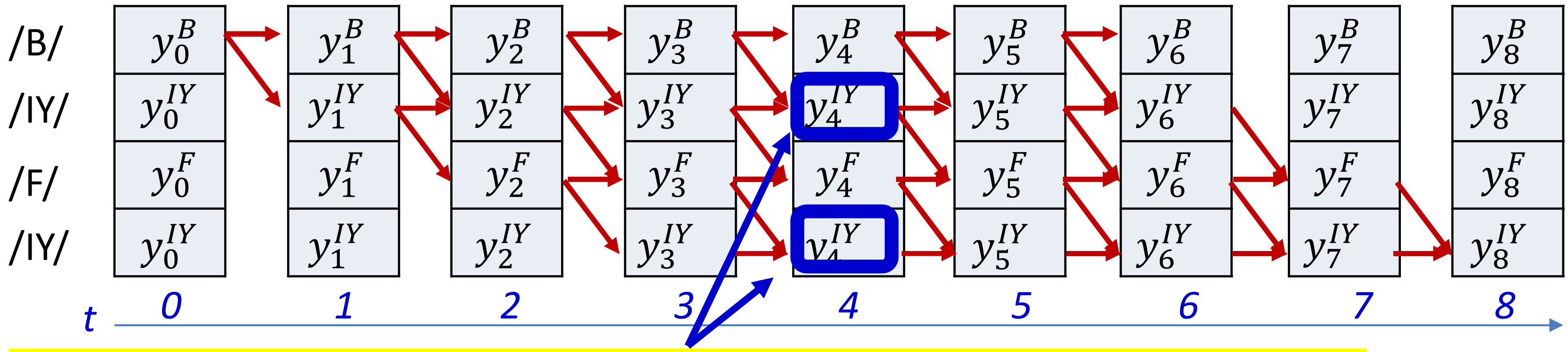
$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{d}{dy_t^l}\gamma(t,r)\log y_t^l$$

- The derivative of the divergence w.r.t the output Y_t of the net at any time:

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^{S_0}} \quad \frac{dDIV}{dy_t^{S_1}} \quad \cdots \quad \frac{dDIV}{dy_t^{S_{L-1}}}\right]$$

  – Components will be non-zero only for symbols that occur in the training instance

The expected divergence



The derivatives at both these locations must be summed to get $\frac{dDIV}{dy_4^{IY}}$

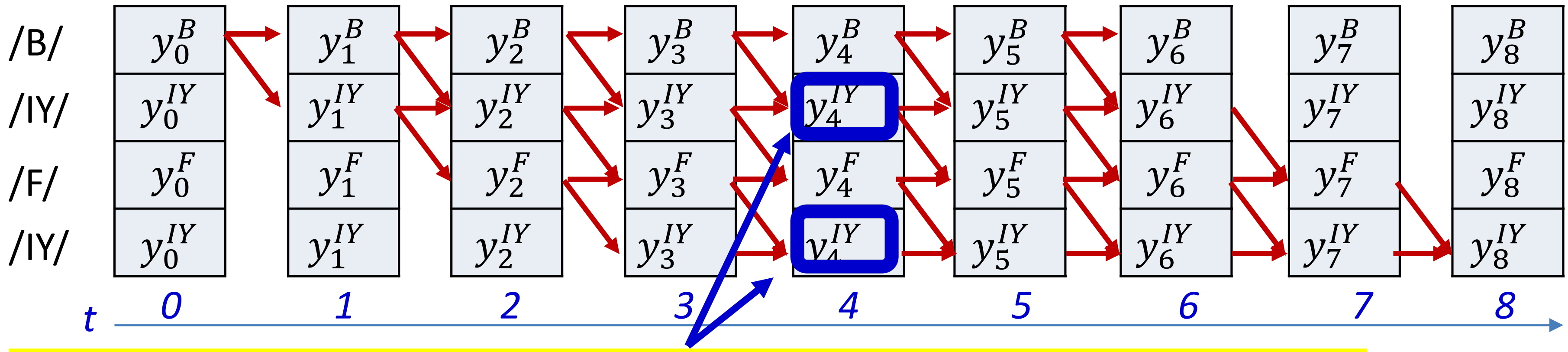
$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{d}{dy_t^l}\gamma(t,r)\log y_t^l$$

- The derivative of the divergence w.r.t the output Y_t of the net at any time:

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^{S_0}} \quad \frac{dDIV}{dy_t^{S_1}} \quad \cdots \quad \frac{dDIV}{dy_t^{S_{L-1}}}\right]$$

  – Components will be non-zero only for symbols that occur in the training instance

The expected divergence

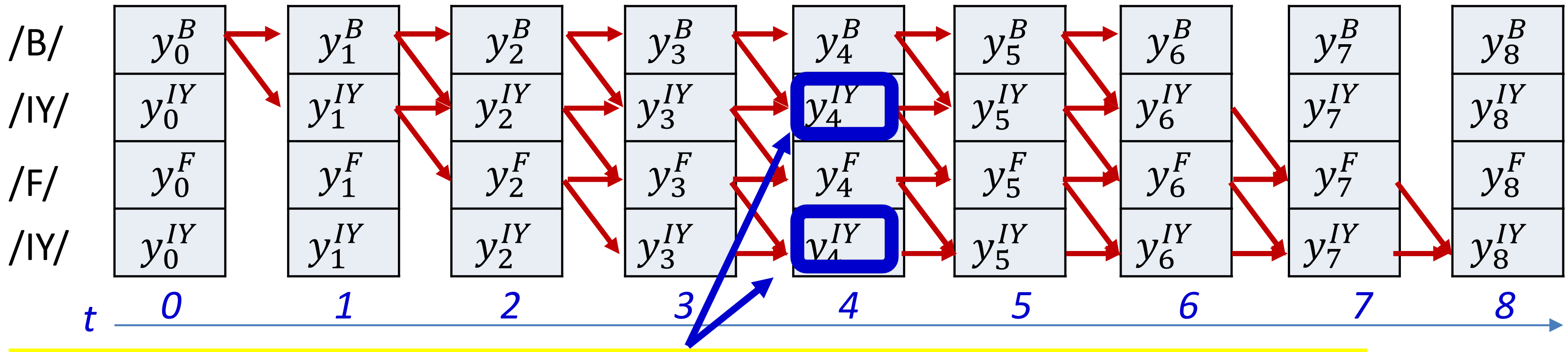


The derivatives at both these locations must be summed to get $\frac{dDIV}{dy_4^{IY}}$

$$\frac{dDIV}{dy_t^l} = - \sum_{r\,:S(r)=l} \frac{d}{dy_t^l} \gamma(t,r) \log y_t^l$$

- $$\frac{d}{dy_t^l} \gamma(t,r) \log y_t^l = \frac{\gamma(t,r)}{y_t^l} + \frac{d\gamma(t,r)}{dy_t^l} \log y_t^l$$ any time:
  - Components will be non-zero only for symbols that occur in the training instance

The expected divergence



The derivatives at both these locations must be summed to get $\frac{dDIV}{dy_4^{IY}}$

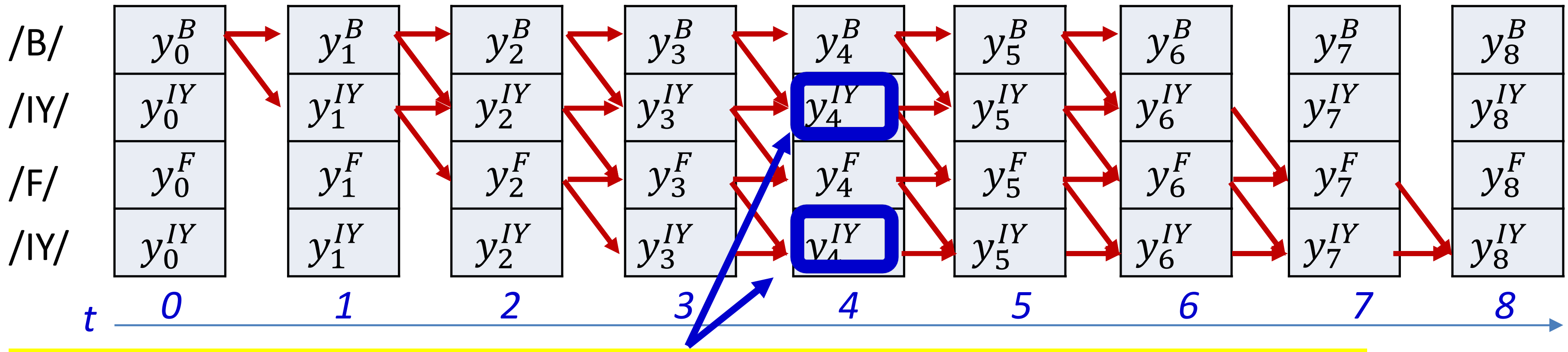
$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{d}{dy_t^l}\gamma(t,r)\log y_t^l$$

- The derivative of t… …net at any time:

$$\frac{d}{dy_t^l}\gamma(t,r)\log y_t^l \approx \frac{\gamma(t,r)}{y_t^l}$$

The approximation is exact if we think of this as a maximum-likelihood estimate

Derivative of the expected divergence



The derivatives at both these locations must be summed to get $\frac{dDIV}{dy_4^{IY}}$

$$DIV = -\sum_t \sum_r \gamma(t,r) \log y_t^{S(r)}$$

- The derivative of the divergence w.r.t any particular output of the network must sum over all instances of that symbol in the target sequence

$$\frac{dDIV}{dy_t^l} = -\frac{1}{y_t^l} \sum_{r\,:S(r)=l} \gamma(t,r)$$

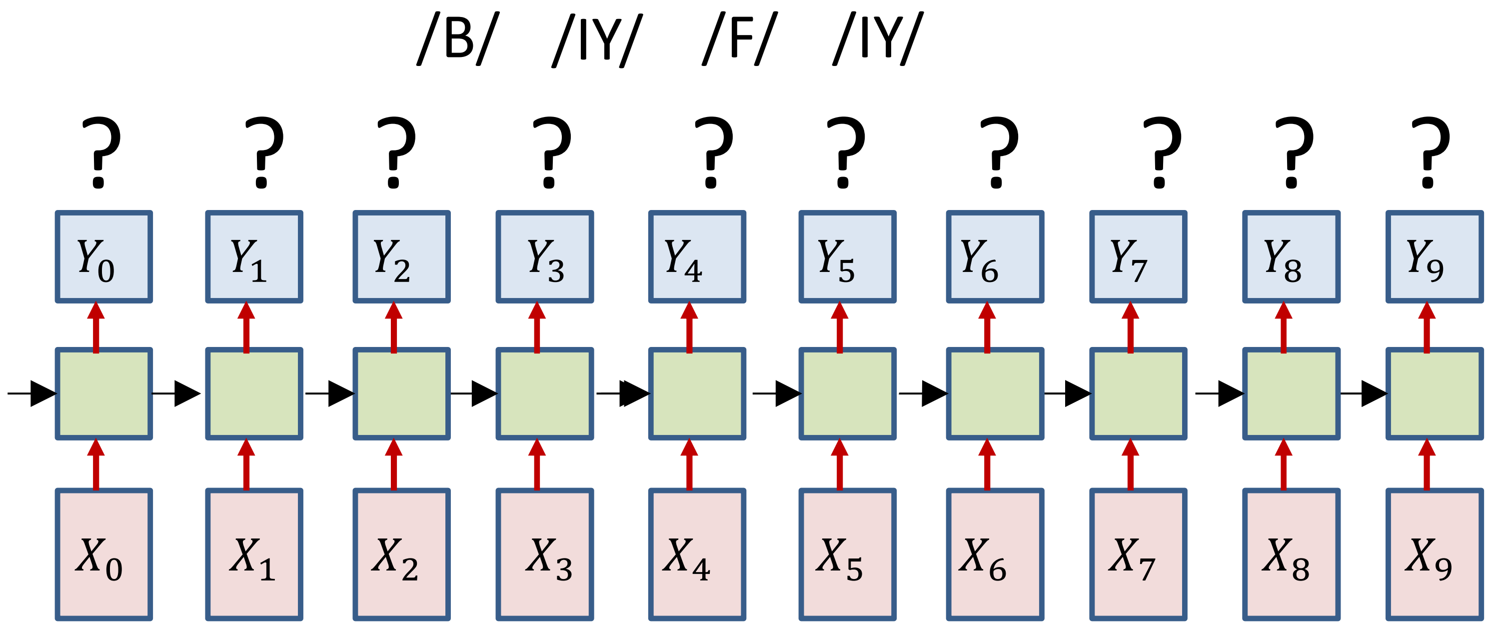
  - E.g. the derivative w.r.t y_t^{IY} will sum over both rows representing /IY/ in the above figure

COMPUTING DERIVATIVES

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#y(t,i) is the output of the network for the ith symbol at time t
#T = length of input

#Assuming the forward are completed first
alpha = forward(y, S)   # forward probabilities computed
beta  = backward(y, S)  # backward probabilities computed

# Compute posteriors from alpha and beta
gamma = computeposteriors(alpha, beta)

#Compute derivatives
for t = 1:T
    dy(t,1:L) = 0  # Initialize all derivatives at time t to 0
    for i = 1:N
        dy(t,S(i)) -= gamma(t,i) / y(t,S(i))
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

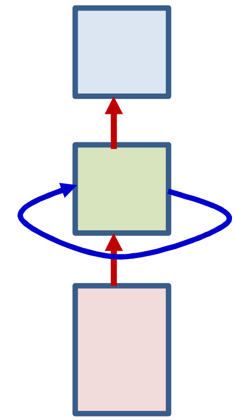
Overall training procedure for Seq2Seq case 1



- Problem: Given input and output sequences without alignment, train models

Overall training procedure for Seq2Seq case 1

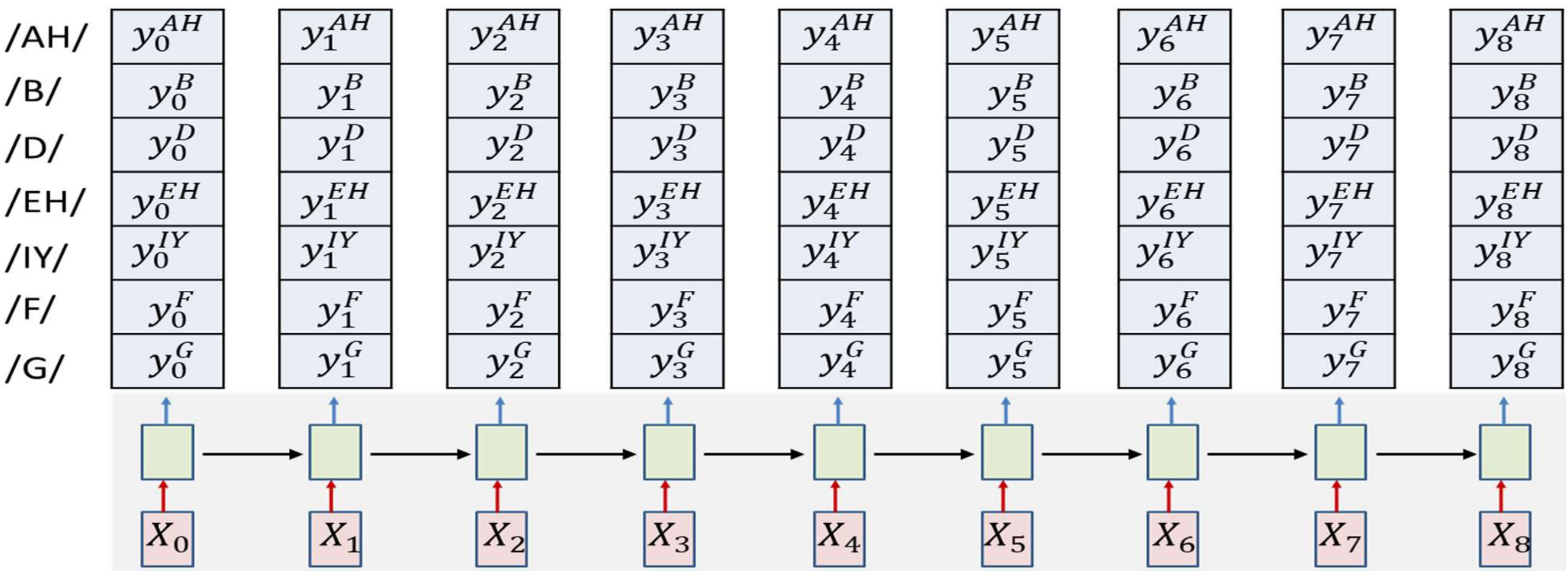
- **Step 1**: Setup the network
  - Typically many-layered LSTM



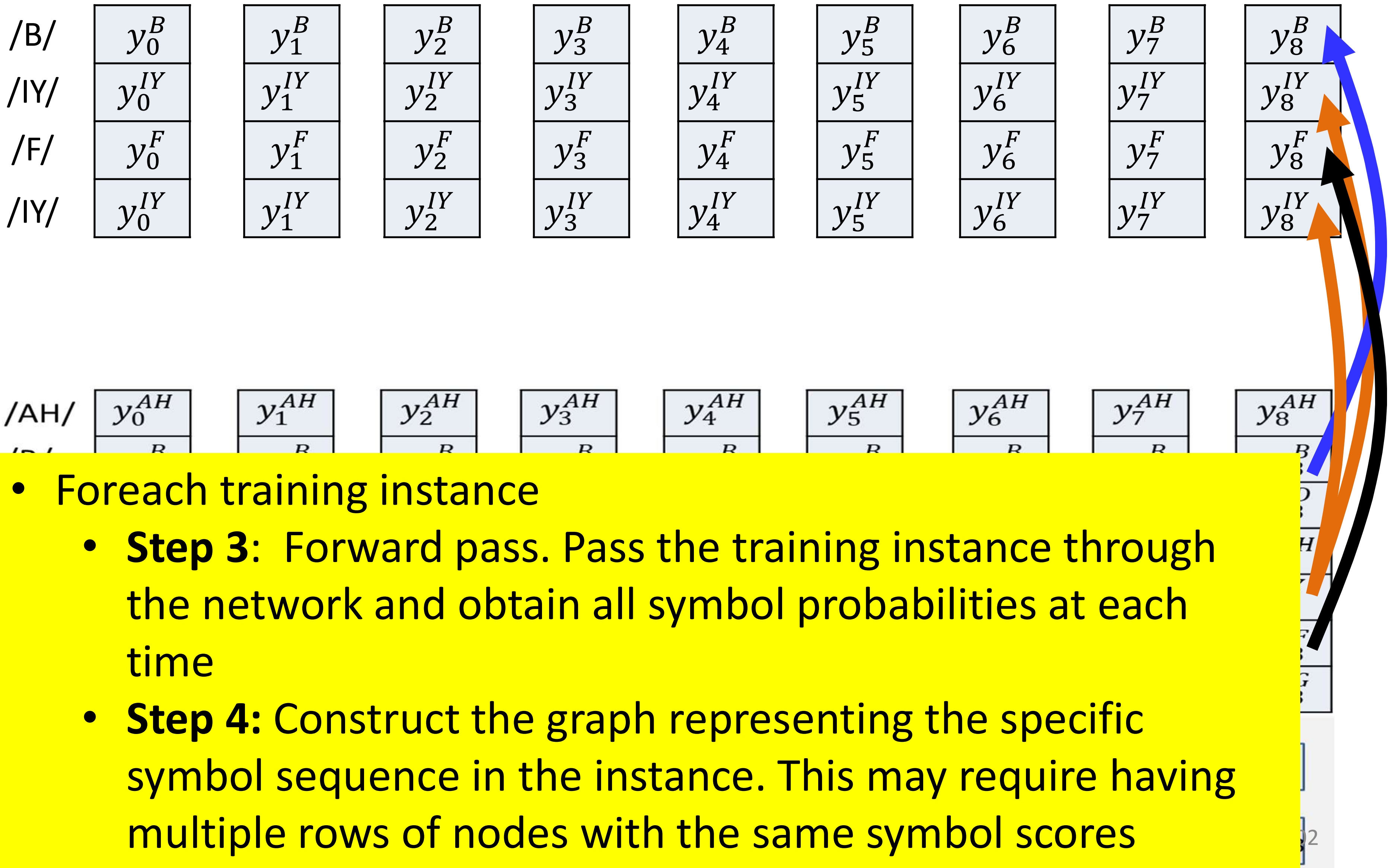
- **Step 2**: Initialize all parameters of the network

Overall Training: Forward pass

- Foreach training instance
  - **Step 3**: Forward pass. Pass the training instance through the network and obtain all symbol probabilities at each time

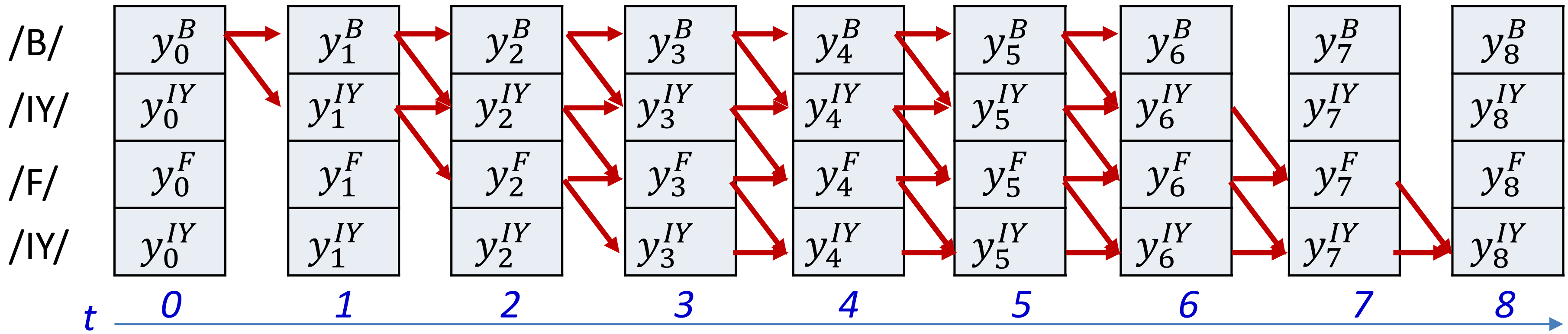


Overall training: Backward pass



- Foreach training instance
  - **Step 3**: Forward pass. Pass the training instance through the network and obtain all symbol probabilities at each time
  - **Step 4:** Construct the graph representing the specific symbol sequence in the instance. This may require having multiple rows of nodes with the same symbol scores

Overall training: Backward pass



- Foreach training instance:
  - **Step 5:** Perform the forward backward algorithm to compute $\alpha(t,r)$ and $\beta(t,r)$ at each time, for each row of nodes in the graph. Compute $\gamma(t,r)$.
  - **Step 6:** Compute derivative of divergence $\nabla_{Y_t} DIV$ for each Y_t

Overall training: Backward pass

- Foreach instance
  – **Step 6:** Compute derivative of divergence $\nabla_{Y_t} DIV$ for each Y_t

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^0} \quad \frac{dDIV}{dy_t^1} \quad \cdots \quad \frac{dDIV}{dy_t^{L-1}}\right]$$

$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{\gamma(t,r)}{y_t^l}$$

- **Step 7**: Backpropagate $\frac{dDIV}{dy_t^l}$ and aggregate derivatives over minibatch and update parameters

Story so far: CTC models

- Sequence-to-sequence networks which irregularly output symbols can be "decoded" by Viterbi decoding
  – Which assumes that a symbol is output at each time and ***merges*** adjacent symbols

- They require alignment of the output to the symbol sequence for training
  – This alignment is generally not given

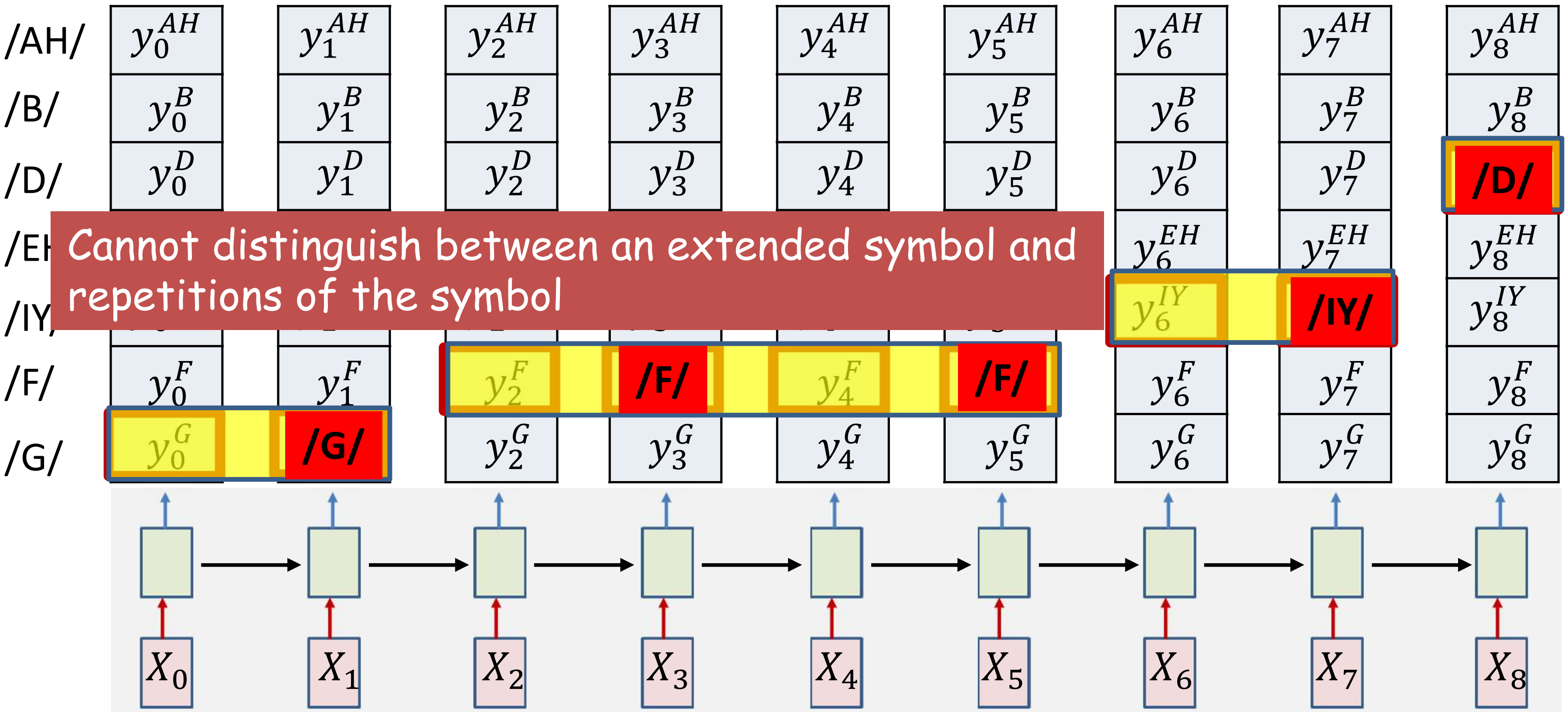
- Training can be performed by iteratively estimating the alignment by Viterbi-decoding and time-synchronous training

- Alternately, it can be performed by optimizing the expected error over *all* possible alignments
  – Posterior probabilities for the expectation can be computed using the forward backward algorithm

A key *decoding* problem

- Consider a problem where the output symbols are characters
- We have a decode: R R R E E E E D
- Is this the compressed symbol sequence RED or REED?

We've seen this before



- /G/ /F/ /F/ /IY/ /D/ or /G/ /F/ /IY/ /D/ ?

A key *decoding* problem

- We have a decode: R R R E E E E E D
- Is this the symbol sequence RED or REED?

- Solution: Introduce an explicit extra symbol which serves to separate discrete versions of a symbol
  - A “blank” (represented by “-”)
  - RRR---EE---DDD = RED
  - RR-E--EED = REED
  - RR-R---EE---D-DD = RREDD
  - R-R-R---E-EDD-DDDD-D = [redacted]
    - The next symbol at the end of a sequence of blanks is always a new character
    - When a symbol repeats, there must be at least one blank between the repetitions

- The symbol set recognized by the network must now include the extra blank symbol
  - Which too must be trained

A key *decoding* problem

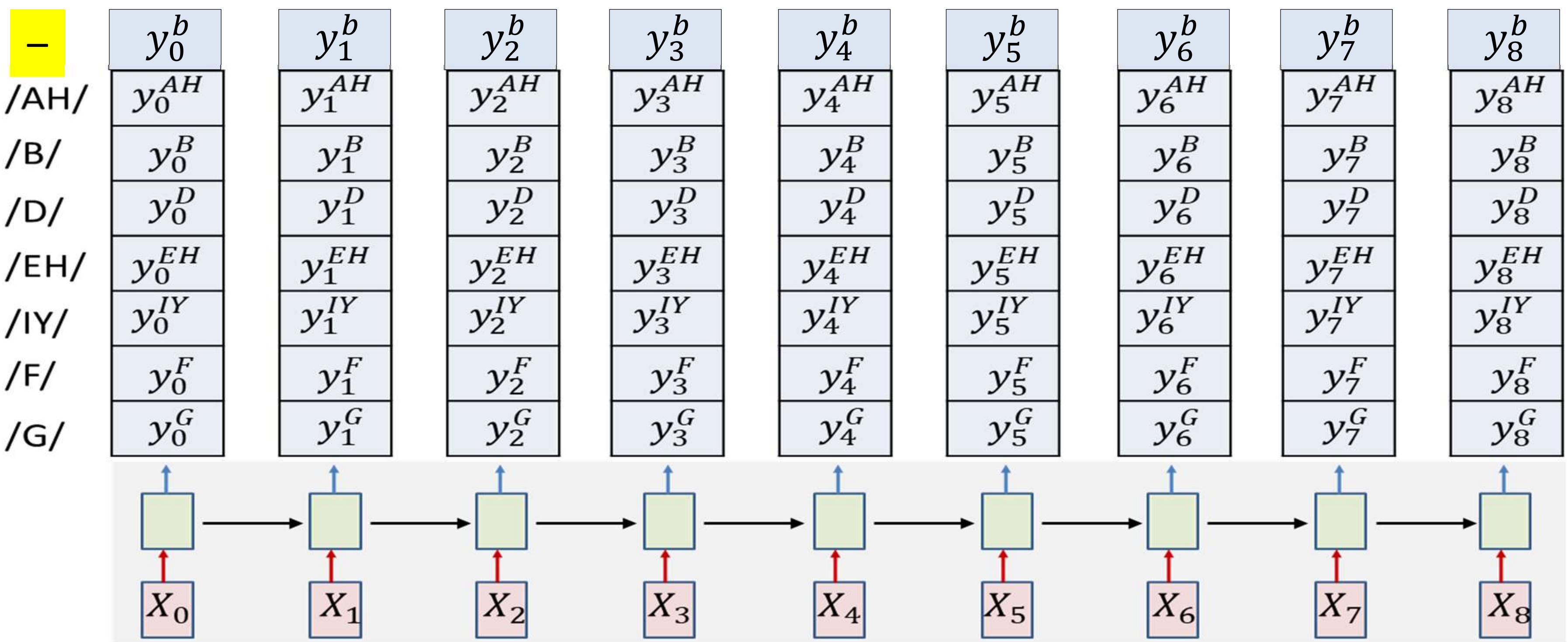
- We have a decode: R R R E E E E E D
- Is this the symbol sequence RED or REED?

- Solution: Introduce an explicit extra symbol which serves to separate discrete versions of a symbol
  – A “blank” (represented by “-”)
  – RRR---EE---DDD = RED
  – RR-E--EED = REED
  – RR-R---EE---D-DD = RREDD
  – R-R-R---E-EDD-DDDD-D = RRREEDDD
    - The next symbol at the end of a sequence of blanks is always a new character
    - When a symbol repeats, there must be at least one blank between the repetitions

- The symbol set recognized by the network must now include the extra blank symbol
  – Which too must be trained

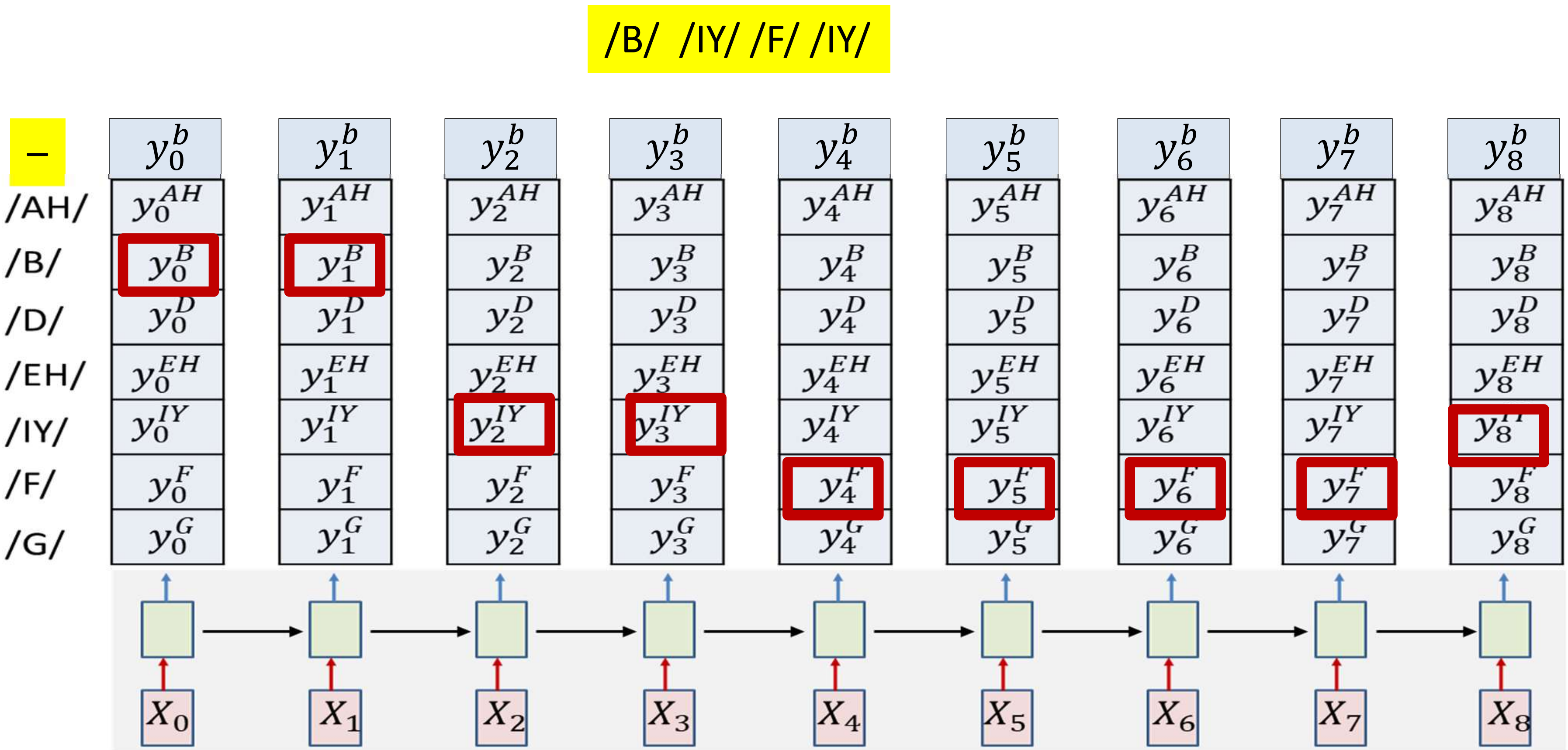
The modified forward output

- Note the extra "blank" at the output



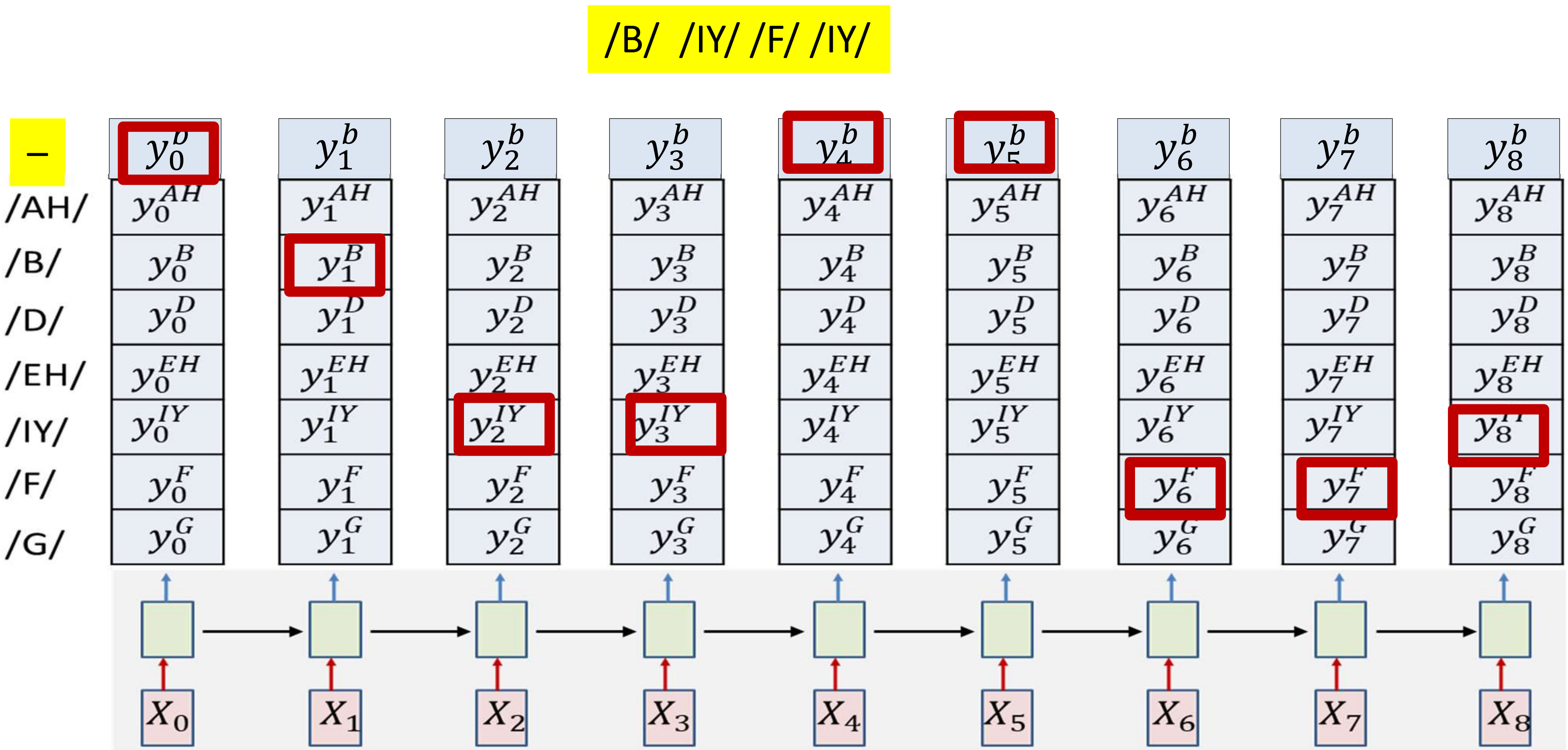
The modified forward output

- Note the extra "blank" at the output



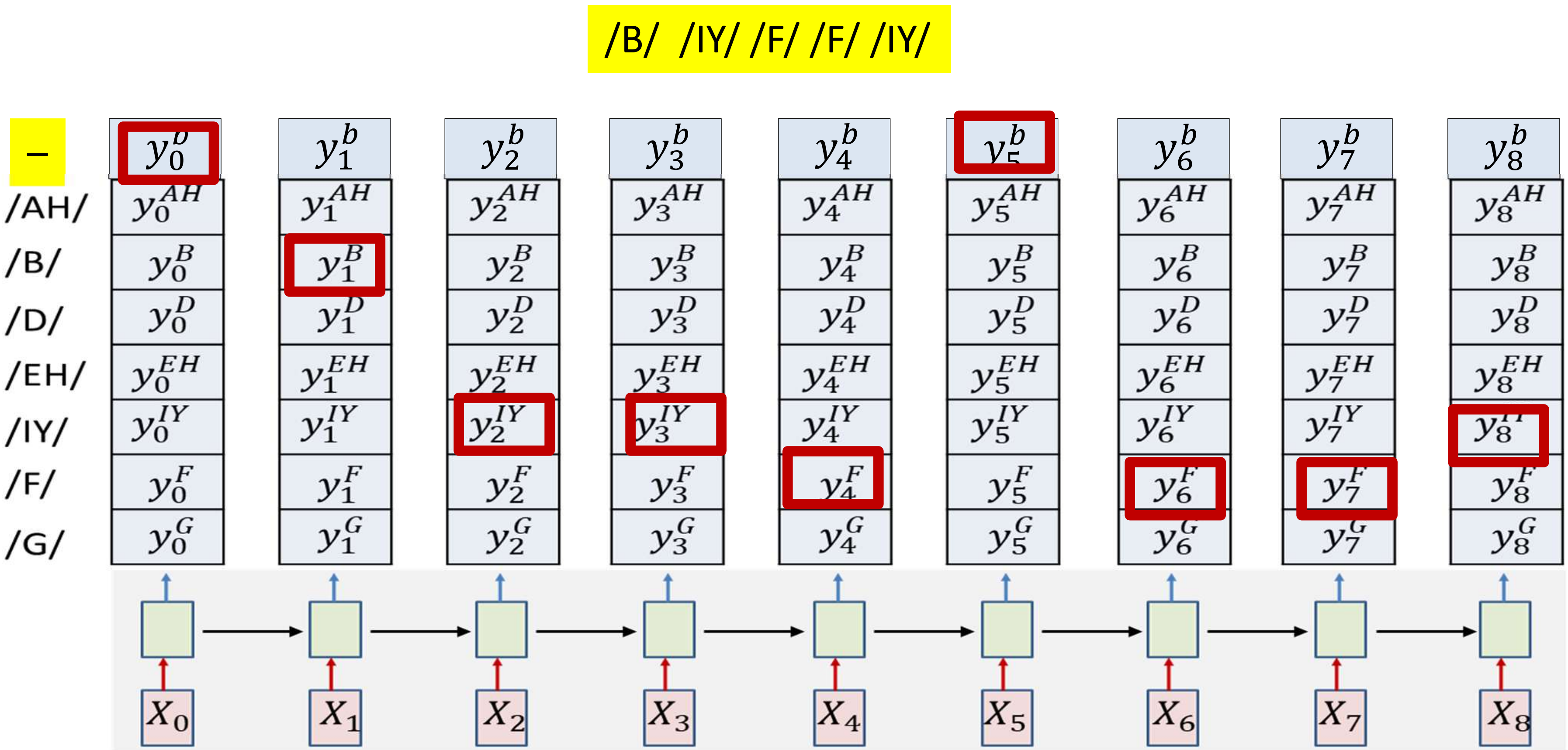
The modified forward output

- Note the extra "blank" at the output

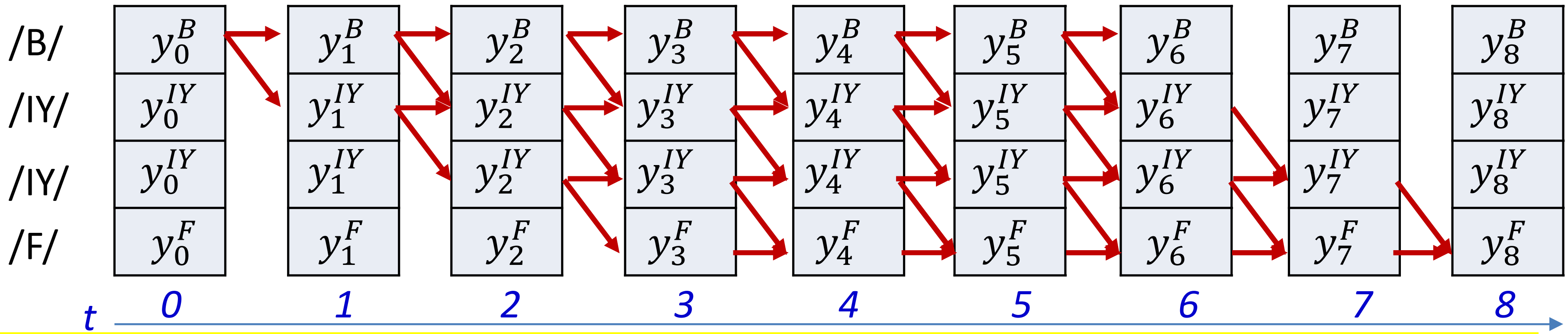


The modified forward output

- Note the extra “blank” at the output



Composing the graph for training



- The original method without blanks

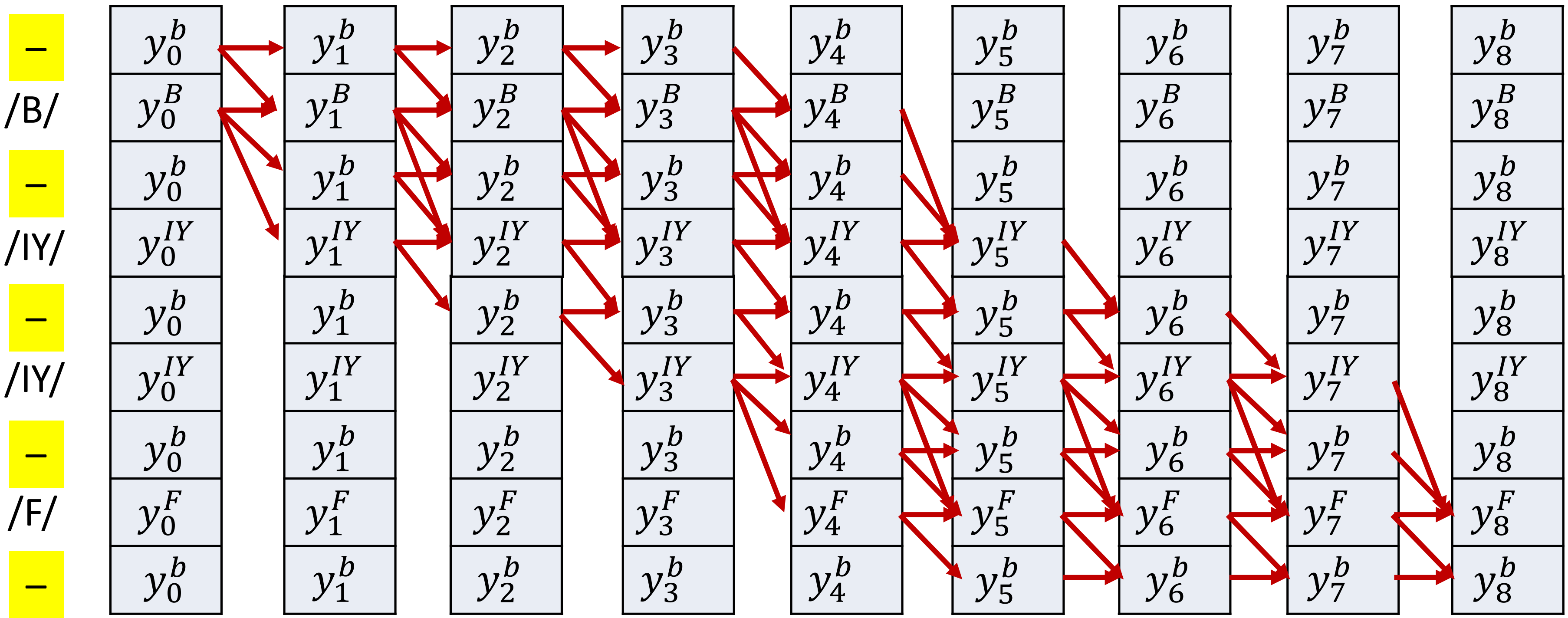
- **Changing the example to /B/ /IY/ /IY/ /F/ from /B/ /IY/ /F/ /IY/ for illustration**

Composing the graph for training

–	y_0^b	y_1^b	y_2^b	y_3^b	y_4^b	y_5^b	y_6^b	y_7^b	y_8^b
/B/	y_0^B	y_1^B	y_2^B	y_3^B	y_4^B	y_5^B	y_6^B	y_7^B	y_8^B
–	y_0^b	y_1^b	y_2^b	y_3^b	y_4^b	y_5^b	y_6^b	y_7^b	y_8^b
/IY/	y_0^{IY}	y_1^{IY}	y_2^{IY}	y_3^{IY}	y_4^{IY}	y_5^{IY}	y_6^{IY}	y_7^{IY}	y_8^{IY}
–	y_0^b	y_1^b	y_2^b	y_3^b	y_4^b	y_5^b	y_6^b	y_7^b	y_8^b
/IY/	y_0^{IY}	y_1^{IY}	y_2^{IY}	y_3^{IY}	y_4^{IY}	y_5^{IY}	y_6^{IY}	y_7^{IY}	y_8^{IY}
–	y_0^b	y_1^b	y_2^b	y_3^b	y_4^b	y_5^b	y_6^b	y_7^b	y_8^b
/F/	y_0^F	y_1^F	y_2^F	y_3^F	y_4^F	y_5^F	y_6^F	y_7^F	y_8^F
–	y_0^b	y_1^b	y_2^b	y_3^b	y_4^b	y_5^b	y_6^b	y_7^b	y_8^b

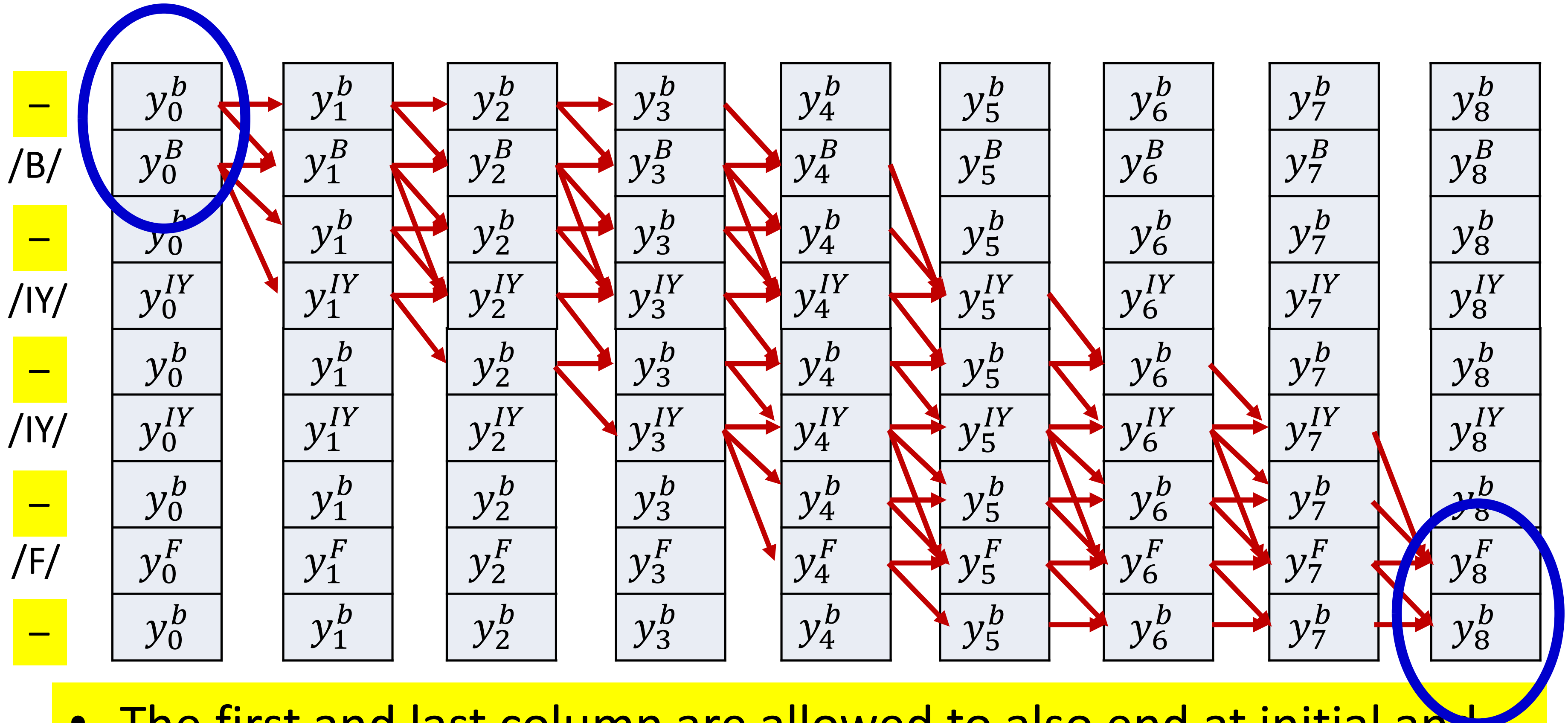
- With blanks
- Note: a row of blanks between any two symbols
- Also blanks at the very beginning and the very end

Composing the graph for training



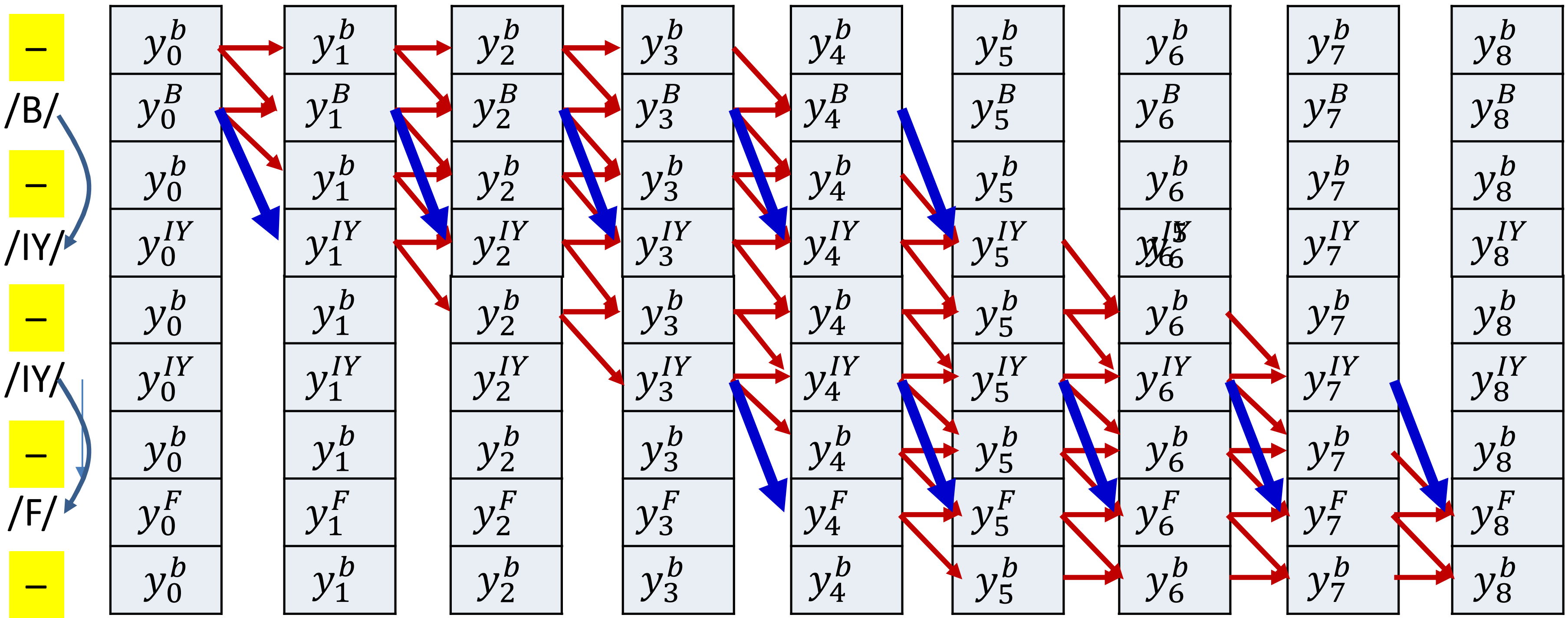
- Add edges such that all paths from initial node(s) to final node(s) unambiguously represent the target symbol sequence

Composing the graph for training



- The first and last column are allowed to also end at initial and final blanks

Composing the graph for training



- The first and last column are allowed to also end at initial and final blanks
- Skips are permitted across a blank, but only if the symbols on either side are different
  - Because a blank is *mandatory between repetitions of a symbol* but *not required between distinct symbols*

Composing the graph

```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output

#Compose an extended symbol sequence Sext from S, that has the blanks
#in the appropriate place
#Also keep track of whether an extended symbol Sext(j) is allowed to connect
#directly to Sext(j-2) (instead of only to Sext(j-1)) or not

function [Sext,skipconnect] = extendedsequencewithblanks(S)
    j = 1
    for i = 1:N
        Sext(j) = 'b' # blank
        skipconnect(j) = 0
        j = j+1

        Sext(j) = S(i)
        if (i > 1 && S(i) != S(i-1))
            skipconnect(j) = 1
        else
            skipconnect(j) = 0
        j = j+1
    end
    Sext(j) = 'b'
    skipconnect(j) = 0

    return Sext, skipconnect
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

Example of using blanks for alignment: Viterbi alignment with blanks

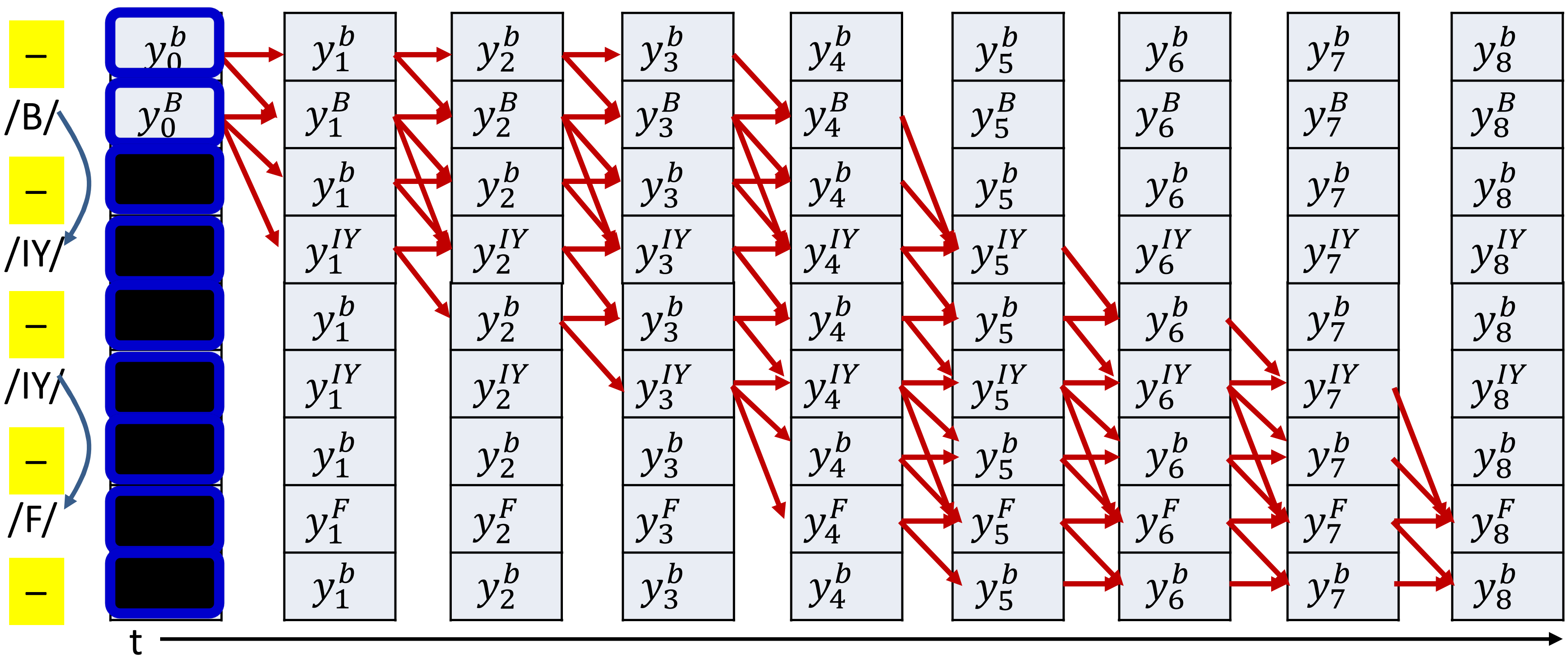
MODIFIED VITERBI ALIGNMENT WITH BLANKS

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext)   # length of extended sequence

# Viterbi starts here
BP(1,1) = -1
Bscr(1,1) = y(1,Sext(1))   # Blank
Bscr(1,2) = y(1,Sext(2))
Bscr(1,2:N) = -infty
for t = 2:T
    BP(t,1) = BP(t-1,1);
    Bscr(t,1) = Bscr(t-1,1)*y(t,Sext(1))
    for i = 1:N
        if skipconnect(i)
           BP(t,i) = argmax_i(Bscr(t-1,i), Bscr(t-1,i-1), Bscr(t-1,i-2)
        else
           BP(t,i) = argmax_i(Bscr(t-1,i), Bscr(t-1,i-1))
        Bscr(t,i) = Bscr(t-1,BP(t,i))*y(t,Sext(i))

# Backtrace
AlignedSymbol(T) = Bscr(T,N) > Bscr(T,N-1) ? N, N-1;
for t = T downto 1
    AlignedSymbol(t-1) = BP(t,AlignedSymbol(t))
```

Without explicit construction of output table

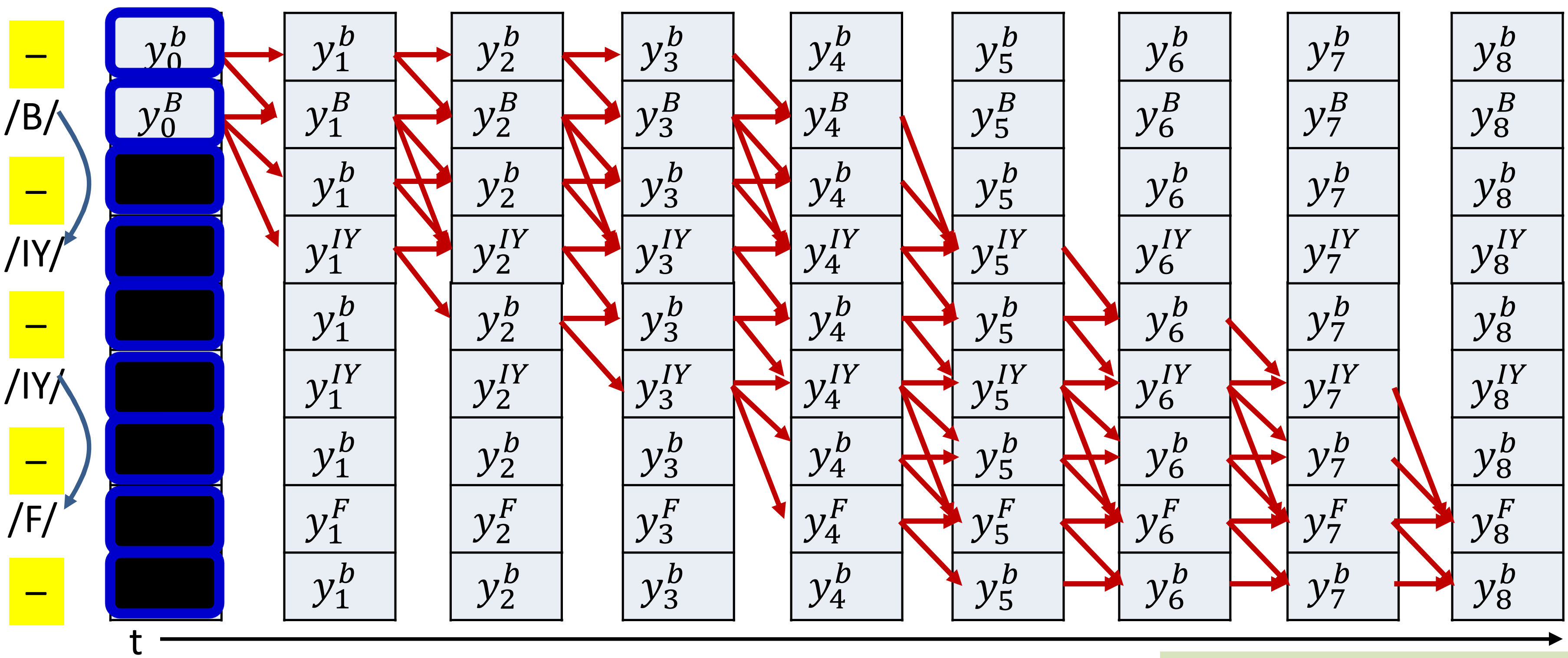
Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

Modified Forward Algorithm



- Initialization:
  - $\alpha(0,0) = y_0^b, \alpha(0,1) = y_0^b, \alpha(0,r) = 0 \quad r > 1$

Modified Forward Algorithm



$$\alpha(t,r) = \sum_{q:S_q \in pred(S_r)} \alpha(t-1,q)\, Y_t^{S(r)}$$

- Iteration:

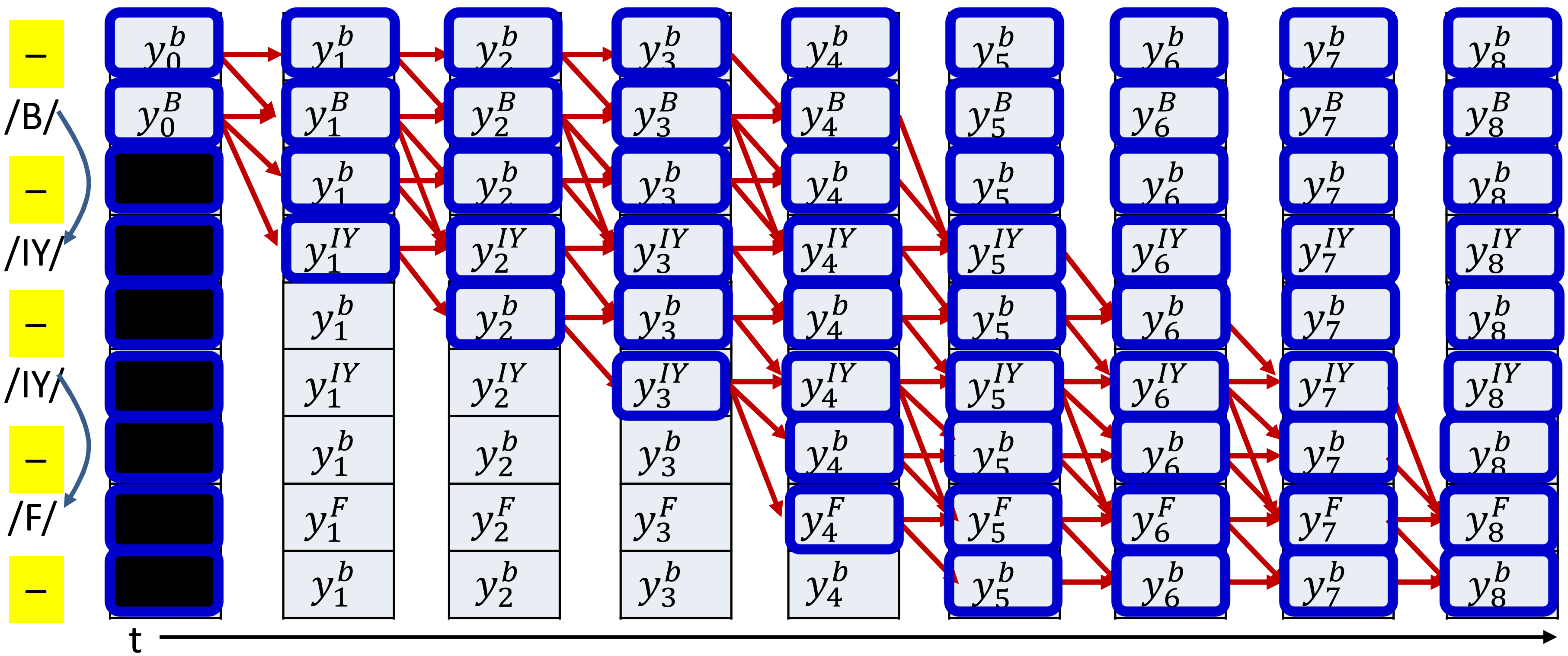
$$\alpha(t,r) = \left(\alpha(t-1,r) + \alpha(t-1,r-1)\right) y_t^{S(r)}$$

  - If $S(r) = "-"$ or $S(r) = S(r-2)$

$$\alpha(t,r) = \left(\alpha(t-1,r) + \alpha(t-1,r-1) + \alpha(t-1,r-2)\right) y_t^{S(r)}$$

  - Otherwise

Modified Forward Algorithm



- Iteration:

$$\alpha(t,r) = \big(\alpha(t-1,r) + \alpha(t-1,r-1)\big)y_t^{S(r)}$$

  - If $S(r) = "-"$ or $S(r) = S(r-2)$

$$\alpha(t,r) = \big(\alpha(t-1,r) + \alpha(t-1,r-1) + \alpha(t-1,r-2)\big)y_t^{S(r)}$$

  - Otherwise

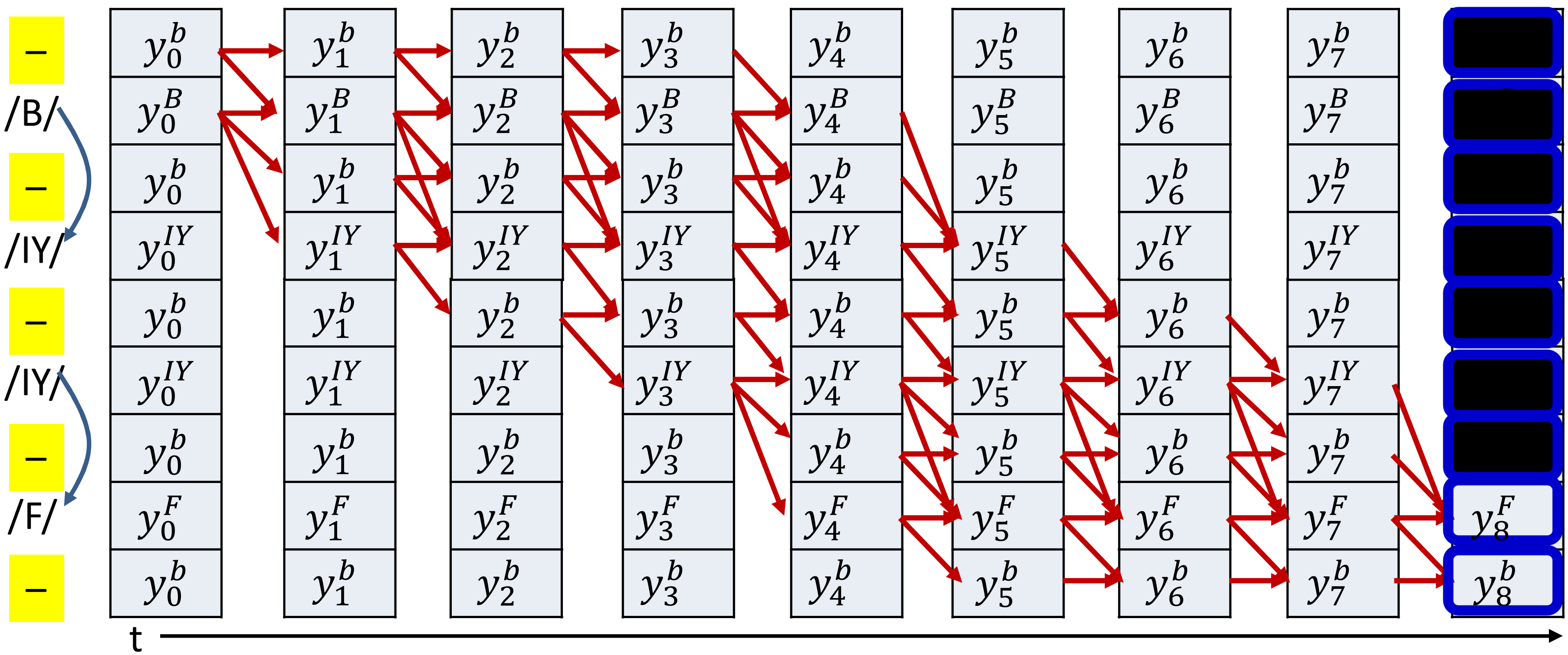
FORWARD ALGORITHM (with blanks)

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#The forward recursion
# First, at t = 1
alpha(1,1) = y(1,Sext(1))  #This is the blank
alpha(1,2) = y(1,Sext(2))
alpha(1,3:N) = 0
for t = 2:T
    alpha(t,1) = alpha(t-1,1)*y(t,Sext(1))
    for i = 2:N
        alpha(t,i) = alpha(t-1,i-1) + alpha(t-1,i))
        if (skipconnect(i))
            alpha(t,i) += alpha(t-1,i-2)
        alpha(t,i) *= y(t,Sext(i))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

Modified Backward Algorithm

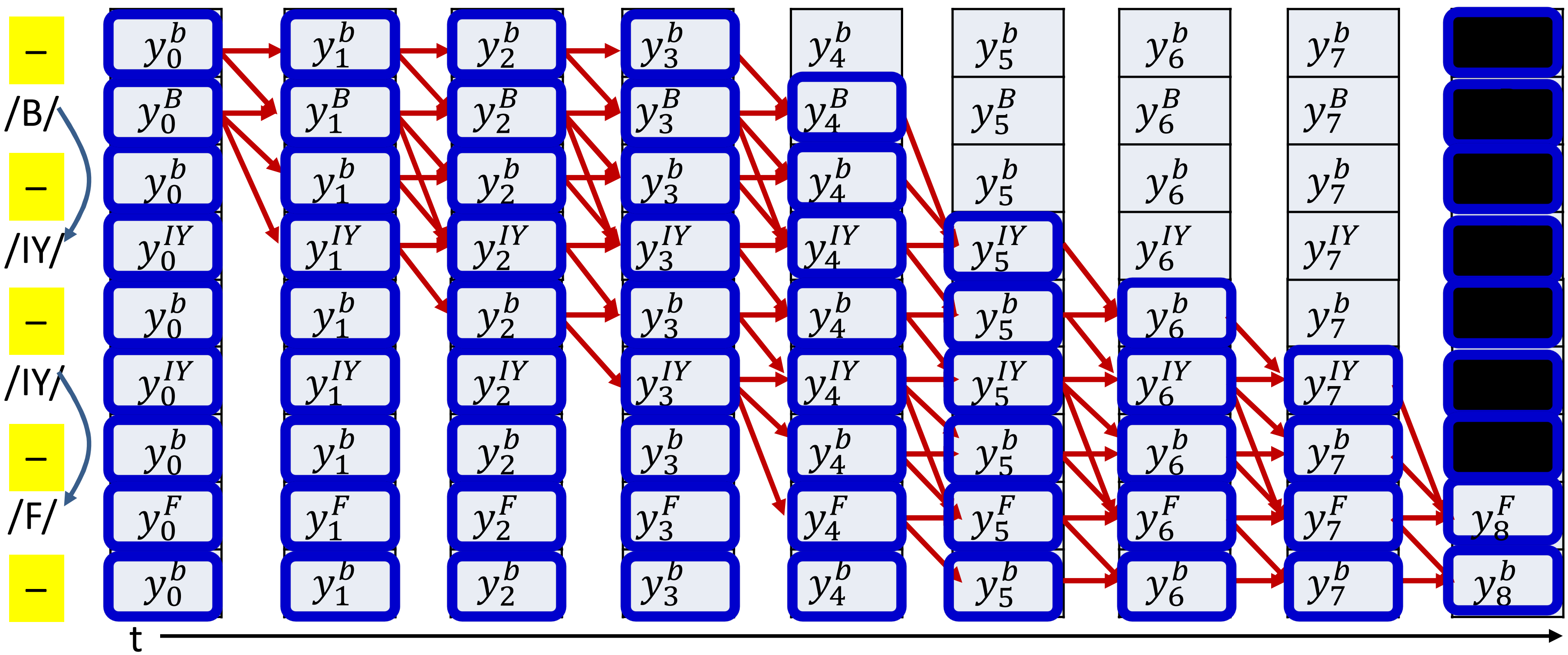


- Initialization:

$$\beta(T-1, 2K-1) = \beta(T-1, 2K-2) = 1$$

$$\beta(T-1, r) = 0 \quad r < 2K-2$$

Modified Backward Algorithm



$$\beta(t,r) = \sum_{q:S_q \in succ(S_r)} \beta(t+1,q) y_{t+1}^{S_q}$$

- Iteration:

$$\beta(t,r) = \beta(t+1,r) y_{t+1}^{S(r)} + \beta(t+1,r+1) y_{t+1}^{S(r+1)}$$

  - If $S(r) = "-"$ or $S(r) = S(r+2)$

$$\beta(t,r) = \beta(t+1,r) y_{t+1}^{S(r)} + \beta(t+1,r+1) y_{t+1}^{S(r+1)} + \beta(t+1,r+2) y_{t+1}^{S(r+2)}$$

  - Otherwise

BACKWARD ALGORITHM WITH BLANKS

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#The backward recursion
# First, at t = T
beta(T,N) = 1
beta(T,N-1) = 1
beta(T,1:N-2) = 0
for t = T-1 downto 1
    beta(t,N) = beta(t+1,N)*y(t+1,Sext(N))
    for i = N-1 downto 1
        beta(t,i) = beta(t+1,i)*y(t+1,Sext(i)) + beta(t+1,i+1))*y(t+1,Sext(i+1))
        if (i<N-2 && skipconnect(i+2))
            beta(t,i) += beta(t+1,i+2)*y(t+1,Sext(i+2))
```

Without explicitly composing the output table

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

The rest of the computation

- Posteriors and derivatives are computed exactly as before

- But using the extended graphs with blanks

COMPUTING POSTERIORS

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#Assuming the forward are completed first
alpha = forward(y, Sext)   # forward probabilities computed
beta  = backward(y, Sext)  # backward probabilities computed

#Now compute the posteriors
for t = 1:T
    sumgamma(t) = 0
    for i = 1:N
        gamma(t,i) = alpha(t,i) * beta(t,i)
        sumgamma(t) += gamma(t,i)
    end
    for i=1:N
        gamma(t,i) = gamma(t,i) / sumgamma(t)
```

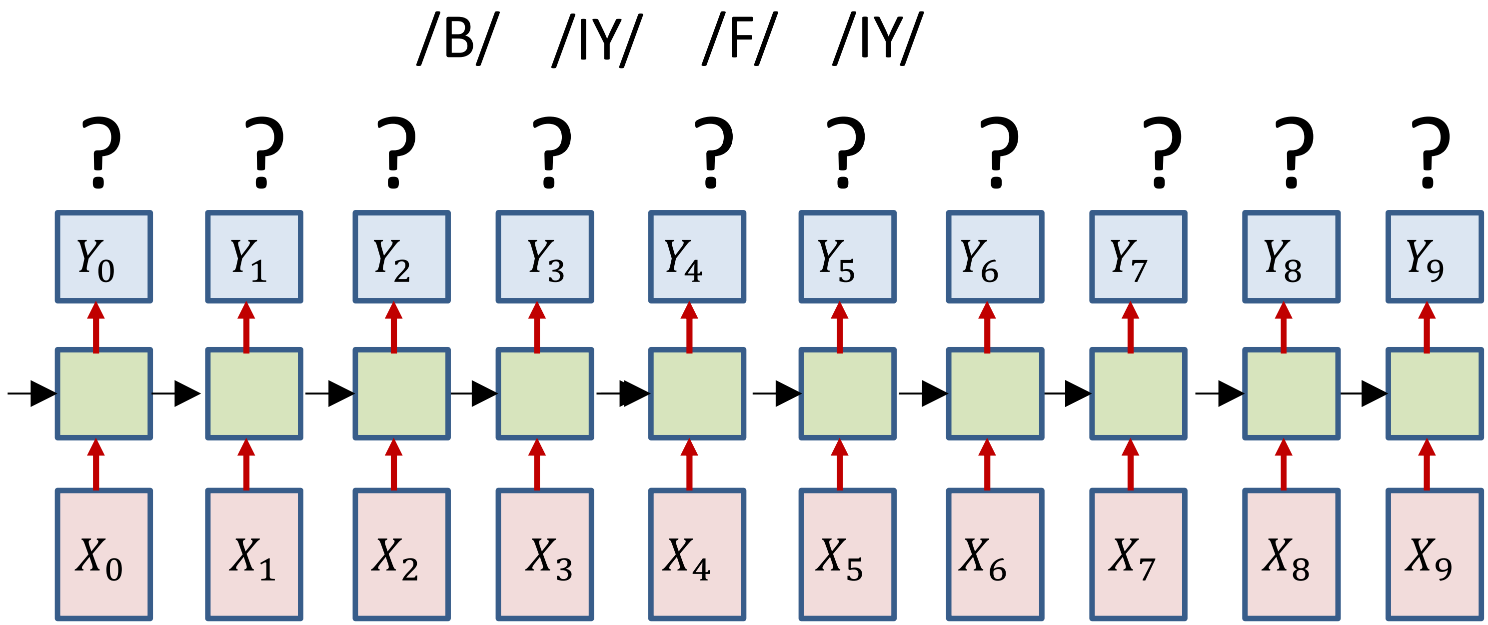
Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

COMPUTING DERIVATIVES

```
[Sext, skipconnect] = extendedsequencewithblanks(S)
N = length(Sext) # Length of extended sequence

#Assuming the forward are completed first
alpha = forward(y, Sext)   # forward probabilities computed
beta  = backward(y, Sext)  # backward probabilities computed

# Compute posteriors from alpha and beta
gamma = computeposteriors(alpha, beta)

#Compute derivatives
for t = 1:T
    dy(t,1:L) = 0 #Initialize all derivatives at time t to 0
    for i = 1:N
        dy(t,Sext(i)) -= gamma(t,i) / y(t,Sext(i))
```

Using 1..N and 1..T indexing, instead of 0..N-1, 0..T-1, for convenience of notation

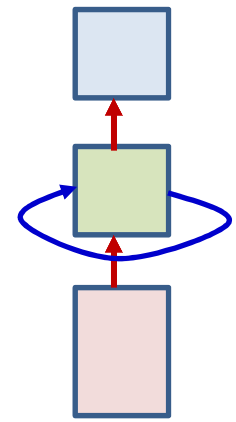
Overall training procedure for Seq2Seq with blanks



- Problem: Given input and output sequences without alignment, train models

Overall training procedure

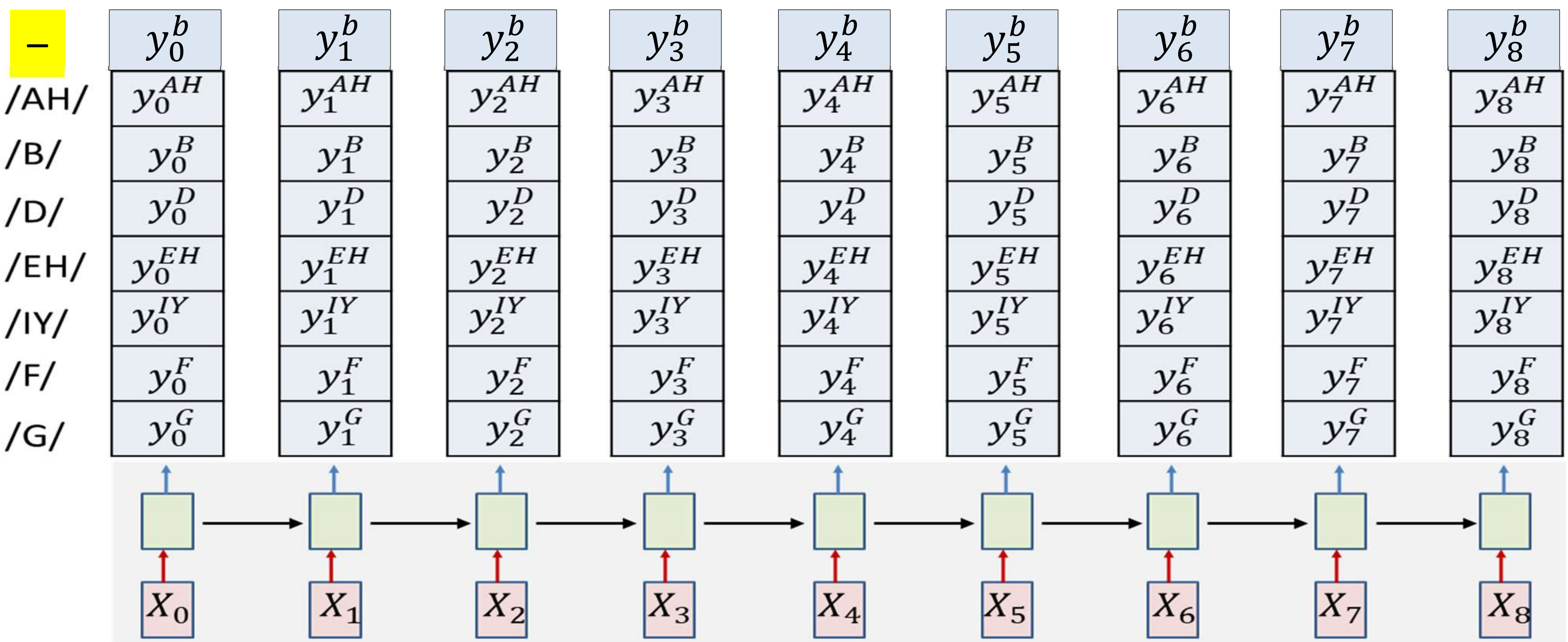
- **Step 1**: Setup the network
  – Typically many-layered LSTM



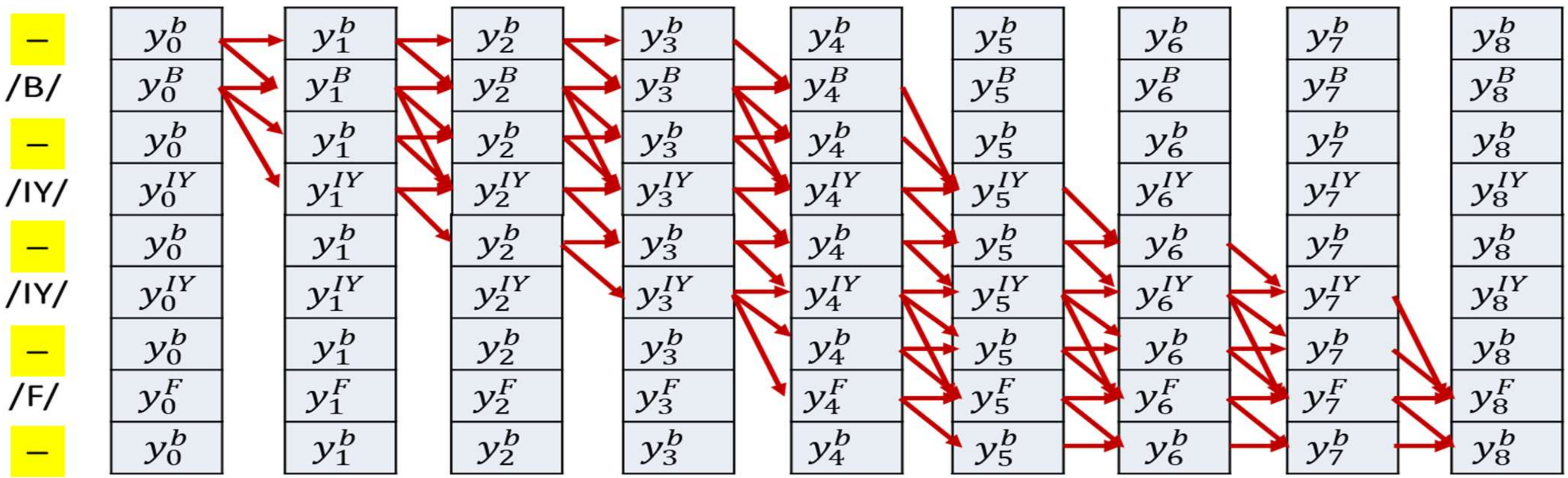
- **Step 2**: Initialize all parameters of the network
  – Include a “blank” symbol in vocabulary

Overall Training: Forward pass

- Foreach training instance
  - **Step 3**: Forward pass. Pass the training instance through the network and obtain all symbol probabilities at each time, including blanks

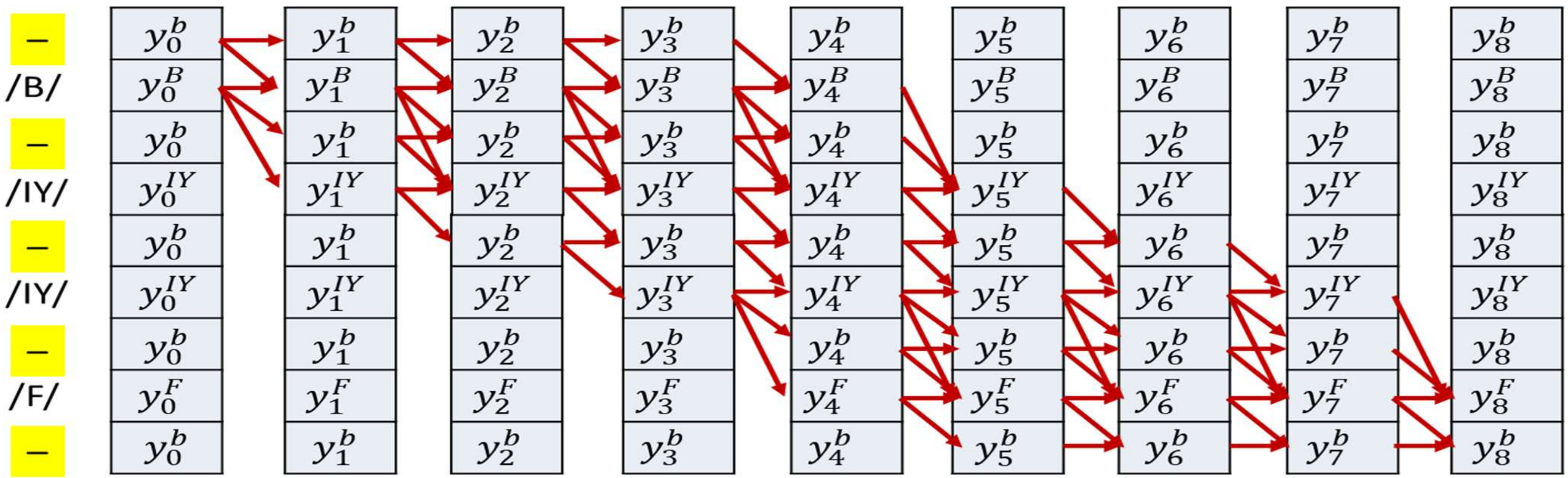


Overall training: Backward pass



- Foreach training instance
  - **Step 3**: Forward pass. Pass the training instance through the network and obtain all symbol probabilities at each time
  - **Step 4:** Construct the graph representing the specific symbol sequence in the instance. Use appropriate connections if blanks are included

Overall training: Backward pass



- Foreach training instance:
  - **Step 5:** Perform the forward backward algorithm to compute $\alpha(t,r)$ and $\beta(t,r)$ at each time, for each row of nodes in the graph using the modified forward-backward equations. Compute a posteriori probabilities $\gamma(t,r)$ from them
  - **Step 6:** Compute derivative of divergence $\nabla_{Y_t} DIV$ for each Y_t

Overall training: Backward pass

- Foreach instance
  – **Step 6:** Compute derivative of divergence $\nabla_{Y_t} DIV$ for each Y_t

$$\nabla_{Y_t} DIV = \left[\frac{dDIV}{dy_t^0} \quad \frac{dDIV}{dy_t^1} \quad \cdots \quad \frac{dDIV}{dy_t^{L-1}}\right]$$

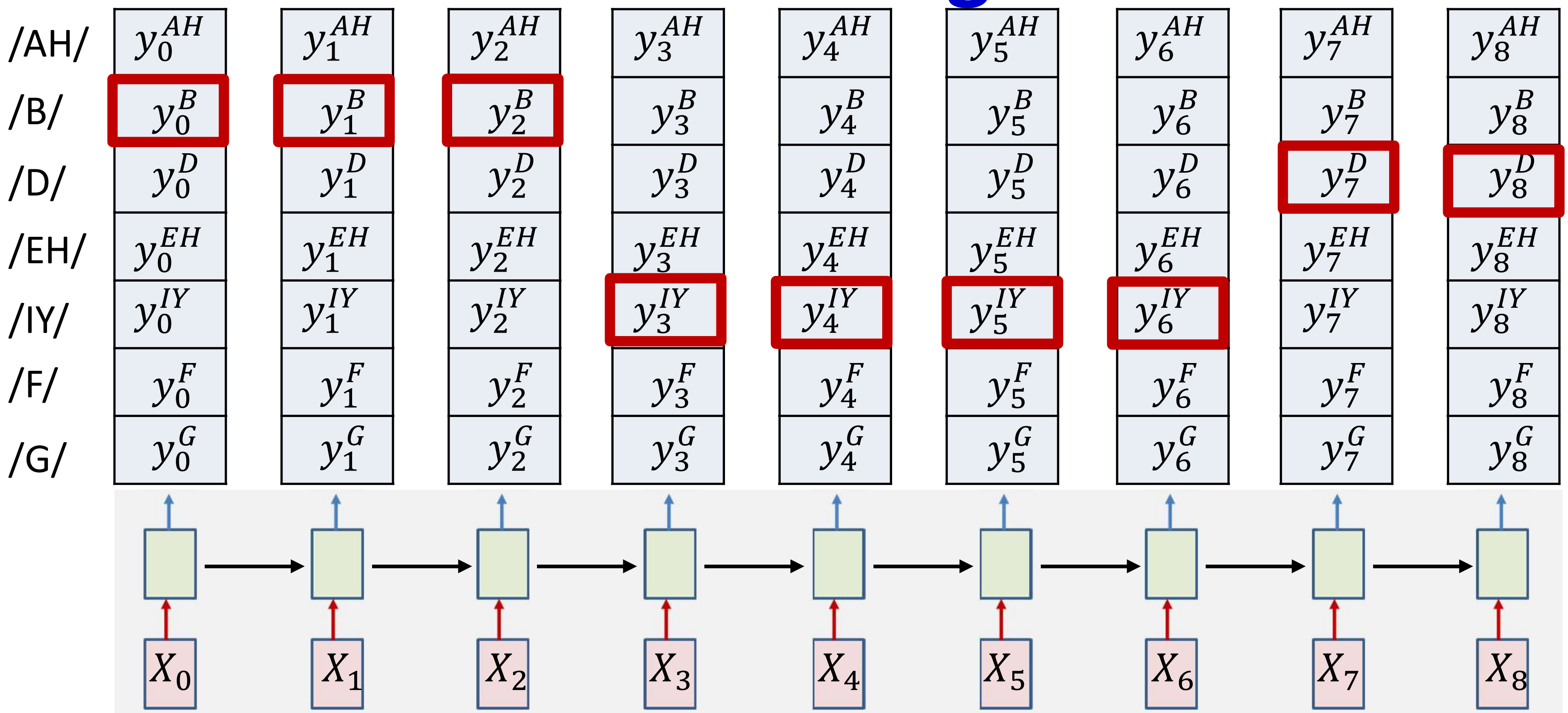
$$\frac{dDIV}{dy_t^l} = -\sum_{r\,:S(r)=l} \frac{\gamma(t,r)}{y_t^{S(r)}}$$

- **Step 7**: Backpropagate $\frac{dDIV}{dy_t^l}$ and aggregate derivatives over minibatch and update parameters

CTC: Connectionist Temporal Classification

- The overall framework we saw is referred to as CTC
- Applies to models that output order-aligned, but time-asynchronous outputs

Returning to an old problem: Decoding



- The greedy decode computes its output by finding the most likely symbol at each time and merging repetitions in the sequence

- This is in fact a *suboptimal* decode that actually finds the most likely *time-synchronous* output sequence
  - Which is not necessarily the most likely *order-synchronous* sequence

Greedy decodes are suboptimal

- Consider the following candidate decodes
  - R R – E E D (RED, 0.7)
  - R R – – E D (RED, 0.68)
  - R R E E E D (RED, 0.69)
  - T T E E E D (TED, 0.71)
  - T T – E E D (TED, 0.3)
  - T T – – E D (TED, 0.29)
- A greedy decode picks the most likely output: TED
- A decode that considers the sum of all alignments of the same final output will select RED
- Which is more reasonable?

Greedy decodes are suboptimal

- Consider the following candidate decodes
  - R R – E E D (RED, 0.7)
  - R R – – E D (RED, 0.68)
  - R R E E E D (RED, 0.69)
  - T T E E E D (TED, 0.71)
  - T T – E E D (TED, 0.3)
  - T T – – E D (TED, 0.29)
- A greedy decode picks the most likely output: TED
- A decode that considers the sum of all alignments of the same final output will select RED
- Which is more reasonable?
- *And yet, remarkably, greedy decoding can be surprisingly effective, when using decoding with blanks*

What a CTC system outputs

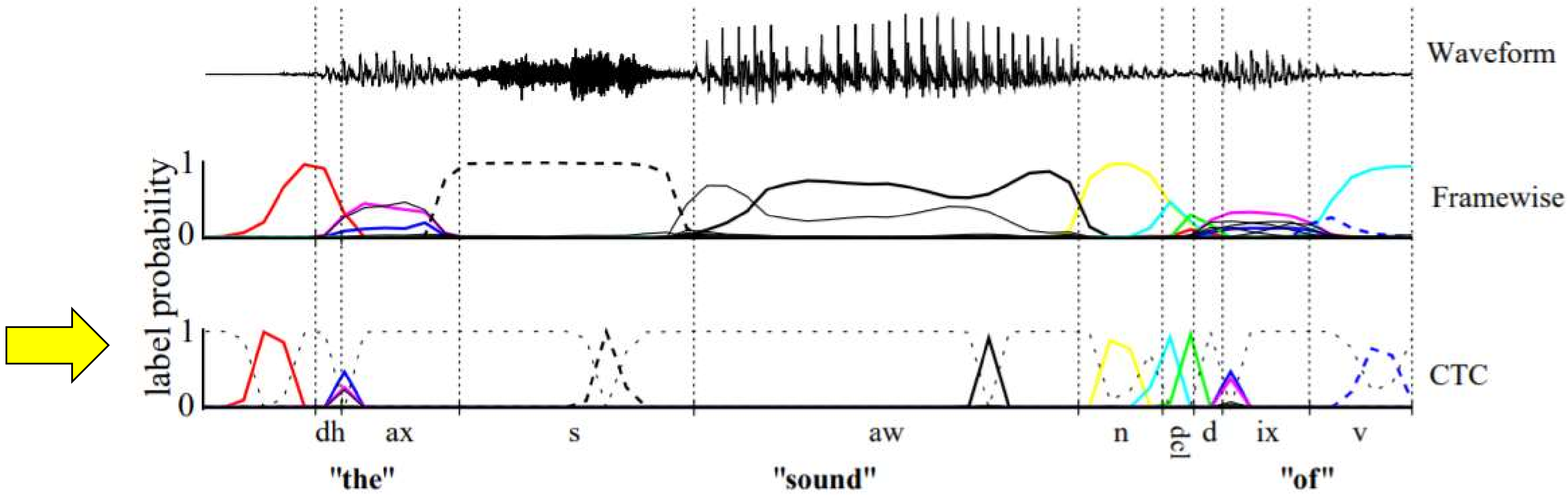


Figure 1. **Framewise and CTC networks classifying a speech signal.** The shaded lines are the output activations, corresponding to the probabilities of observing phonemes at particular times. The CTC network predicts only the sequence of phonemes (typically as a series of spikes, separated by 'blanks', or null predictions), while the framewise network attempts to align them with the manual segmentation (vertical lines). The framewise network receives an error for misaligning the segment boundaries, even if it predicts the correct phoneme (e.g. 'dh'). When one phoneme always occurs beside another (e.g. the closure 'dcl' with the stop 'd'), CTC tends to predict them together in a double spike. The choice of labelling can be read directly from the CTC outputs (follow the spikes), whereas the predictions of the framewise network must be post-processed before use.

- Ref: Graves
- Symbol outputs peak at the ends of the sounds
  - Typical output: - - R - - - E - - -D
  - Model output naturally eliminates alignment ambiguities
- But this is still suboptimal..

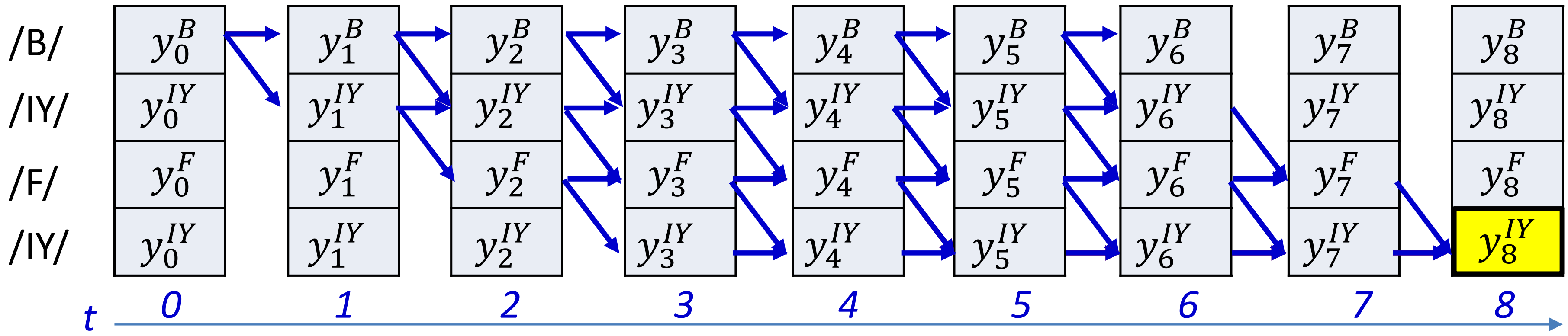
Actual objective of decoding

- Want to find most likely order-aligned symbol sequence
  - **R E D**
  - What greedy decode finds: most likely time synchronous symbol sequence
    - **– /R/ /R/ – – /EH//EH//D/**
    - Which must be compressed

- Find the order-aligned symbol sequence $\boldsymbol{S} = S_0, \ldots, S_{K-1}$, given an input $\boldsymbol{X} = X_0, \ldots, X_{T-1}$, that is most likely

$$= \underset{\boldsymbol{S}}{\operatorname{argmax}}\, P(S_0, \ldots, S_{K-1} | \boldsymbol{X})$$

Recall: The forward probability $\alpha(t, r)$



$$\alpha_{S_0..S_{K-1}}(T-1, K-1) = P(S_0..S_{K-1}|\mathbf{X})$$

- The probability of the entire symbol sequence is the alpha at the bottom right node

Actual decoding objective

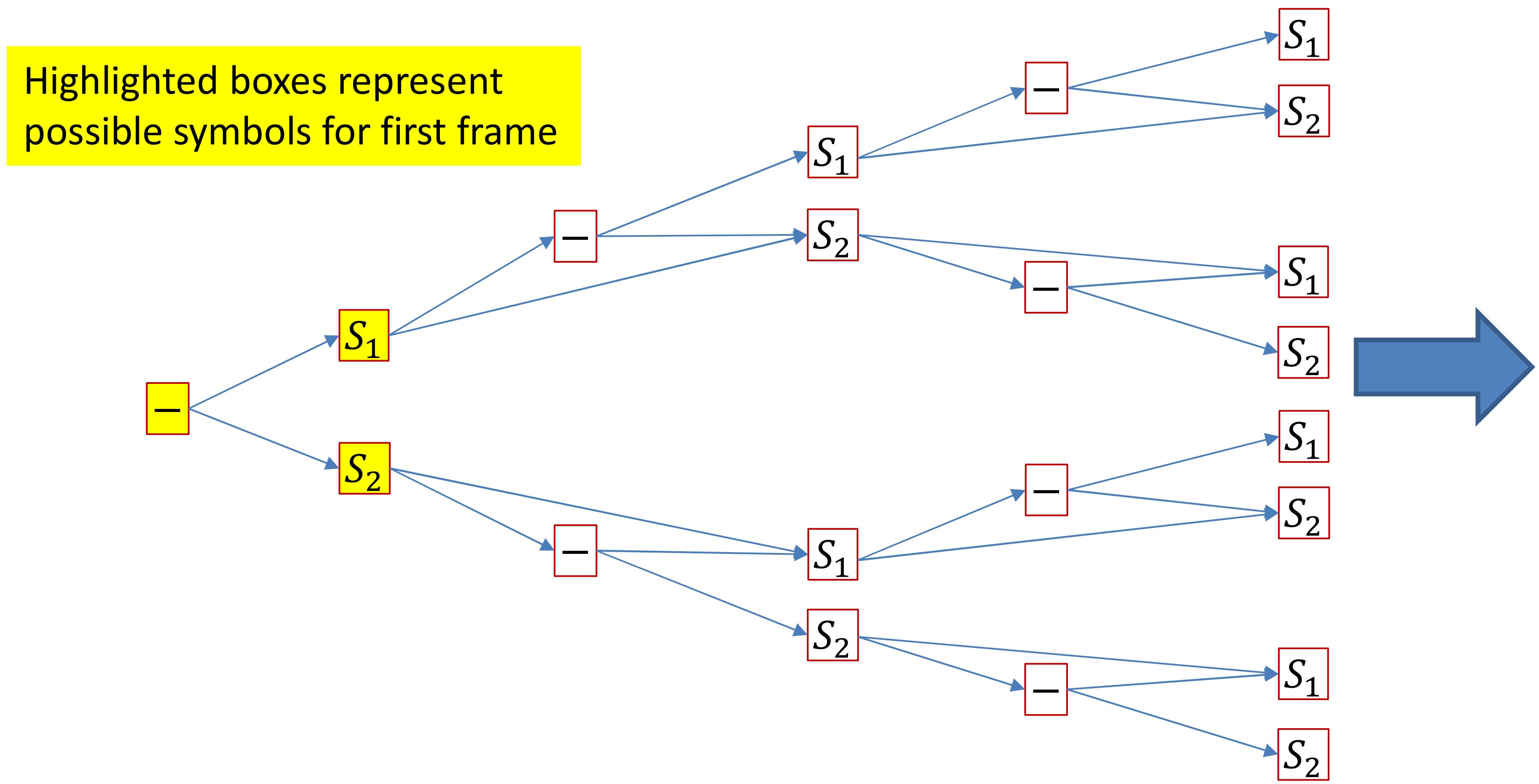
- Find the most likely (asynchronous) symbol sequence

$$\hat{\mathbf{S}} = \underset{\mathbf{S}}{\operatorname{argmax}}\, \alpha_{\mathbf{S}}(S_{K-1}, T-1)$$

- Unfortunately, explicit computation of this will require evaluate of an exponential number of symbol sequences

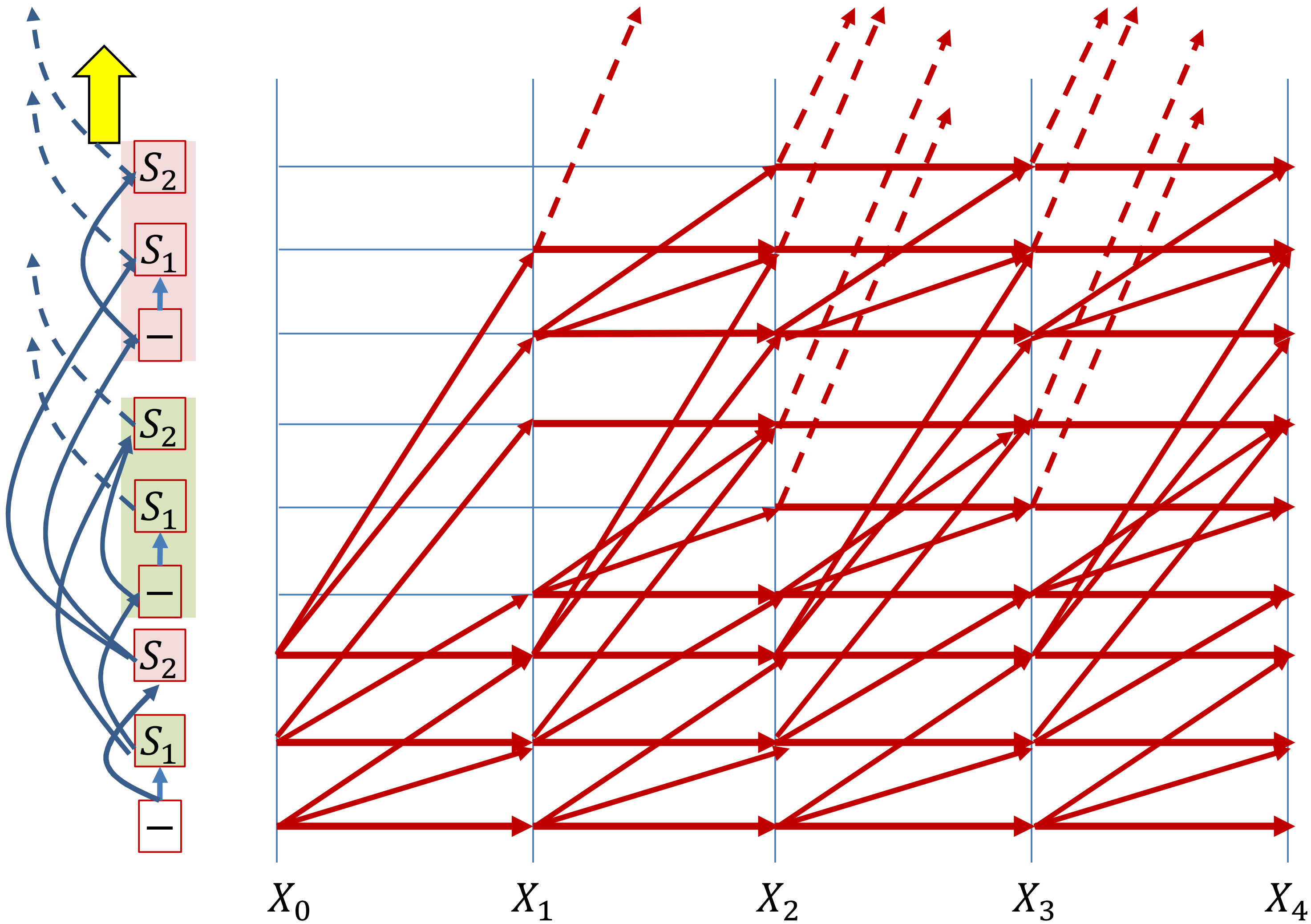
- Solution: Organize all possible symbol sequences as a (semi)tree

Hypothesis semi-tree



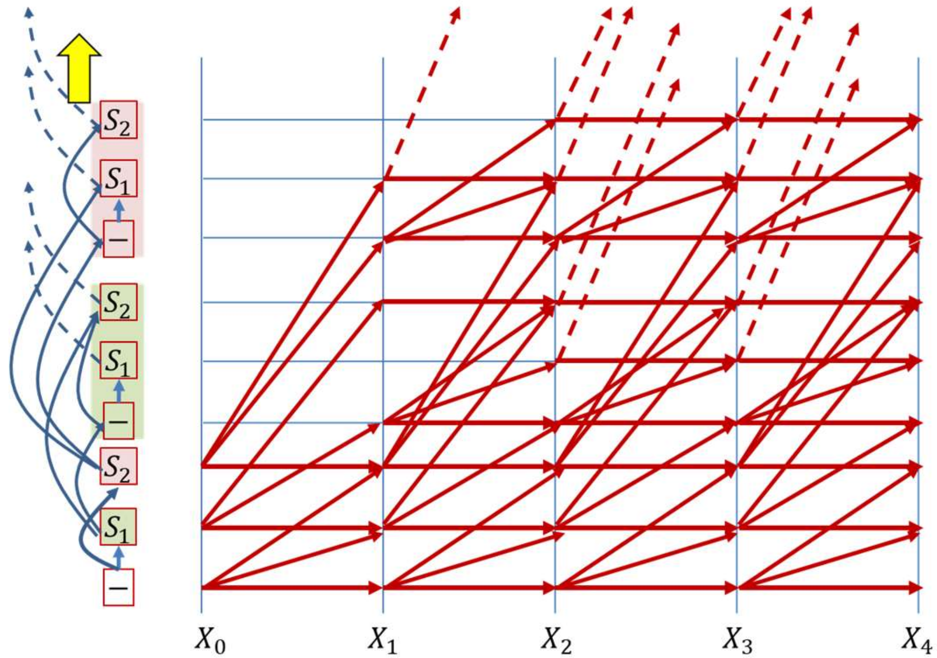
- The semi tree of hypotheses (assuming only 3 symbols in the vocabulary)
- Every symbol connects to every symbol other than itself
  - It also connects to a blank, which connects to every symbol including itself
- The simple structure repeats recursively
- Each node represents a unique (partial) symbol sequence!

The decoding graph for the tree



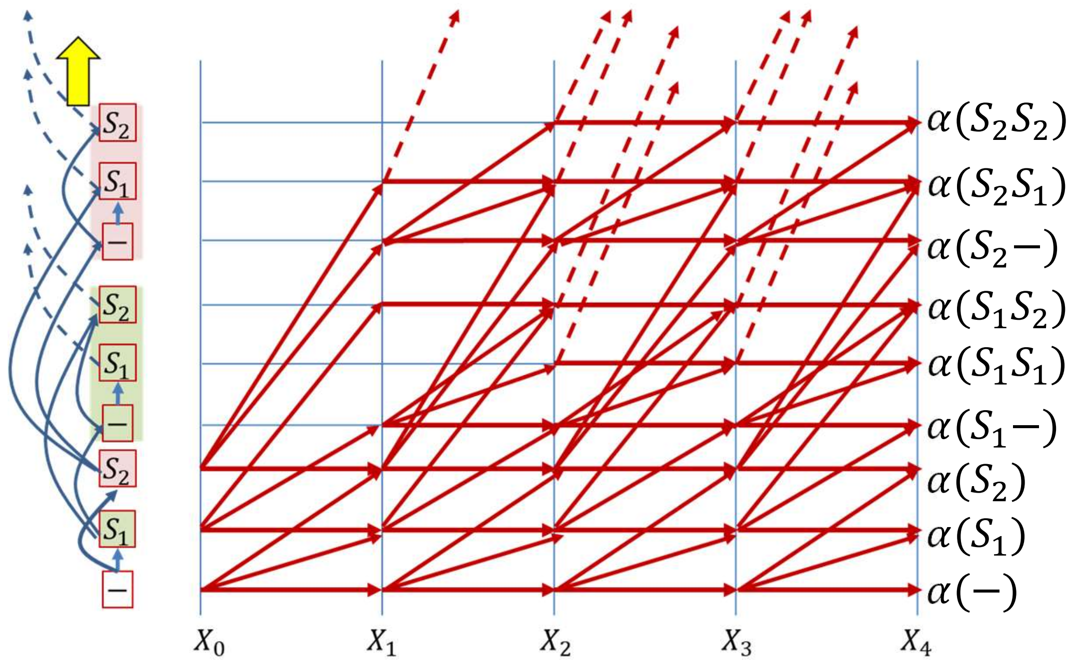
- Graph with more than 2 symbols will be similar but much more cluttered and complicated

The decoding graph for the tree



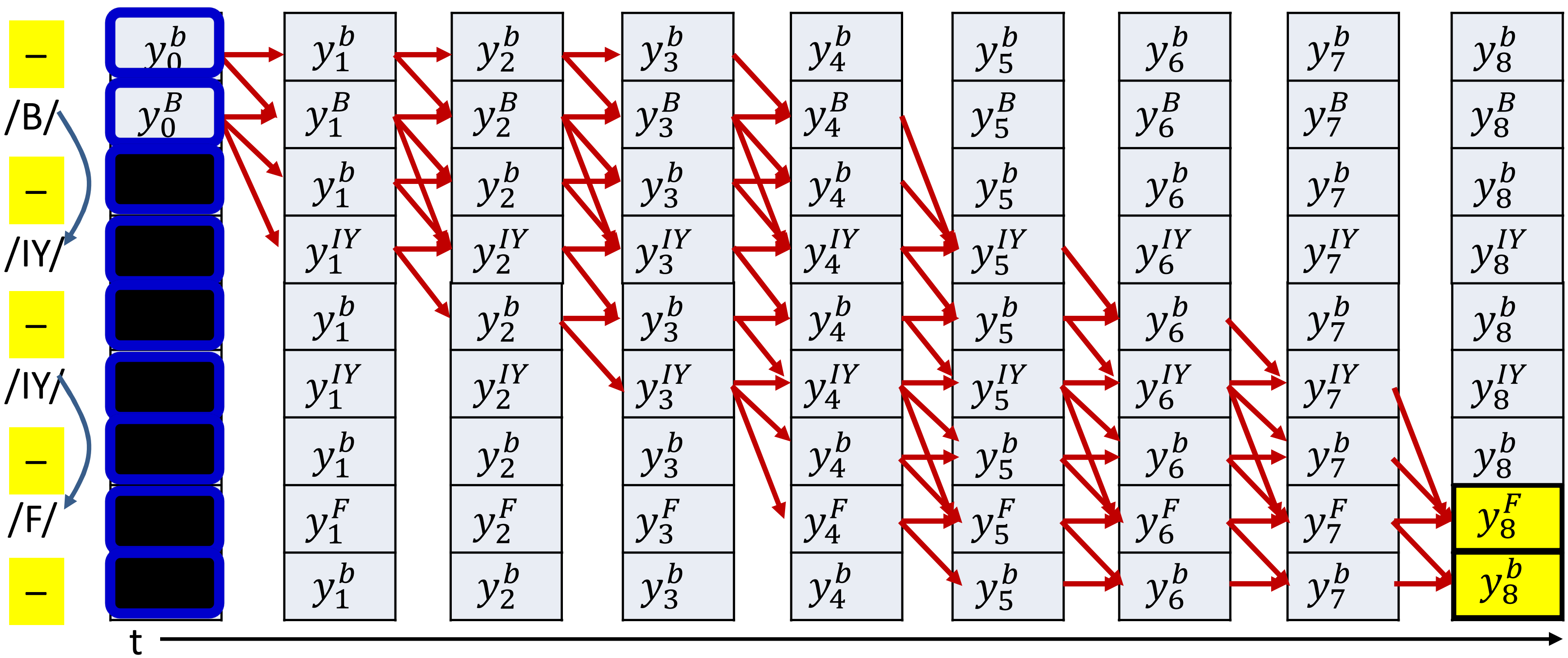
- The figure to the left is the tree, drawn in a vertical line
- The graph is just the tree unrolled over time
  - For a vocabulary of V symbols, every node connects out to V other nodes at the next time
- Every node in the graph represents a unique symbol sequence

The decoding graph for the tree



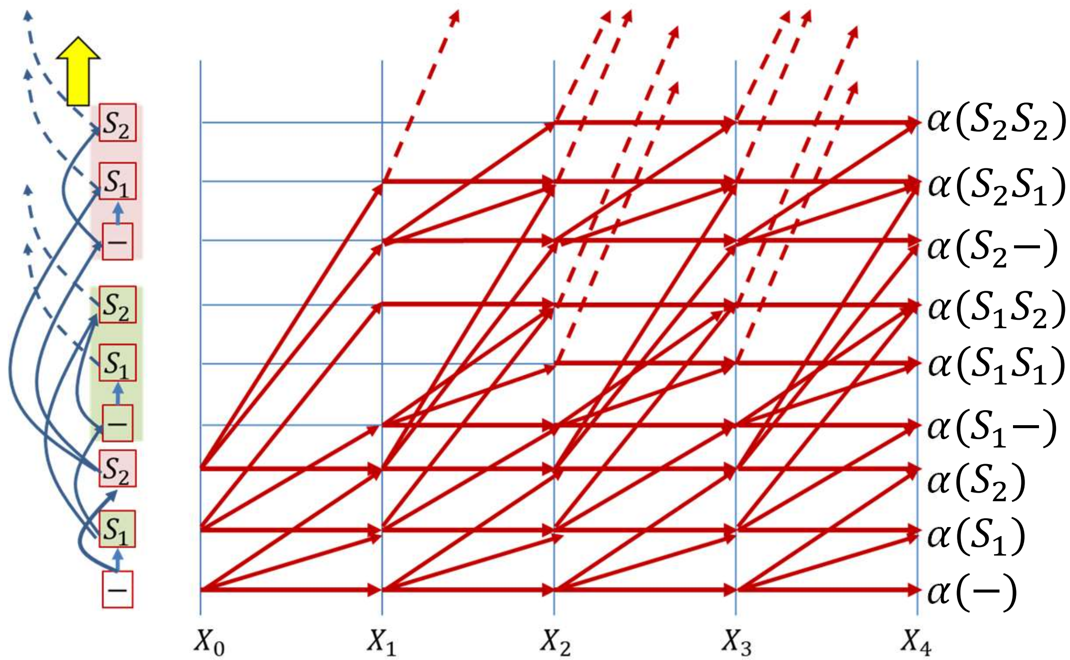
- The forward score $\alpha(r,T)$ at the final time represents the full forward score for a unique symbol sequence (including sequences terminating in blanks)
- Select the symbol sequence with the largest alpha at the final time

Recall: Forward Algorithm



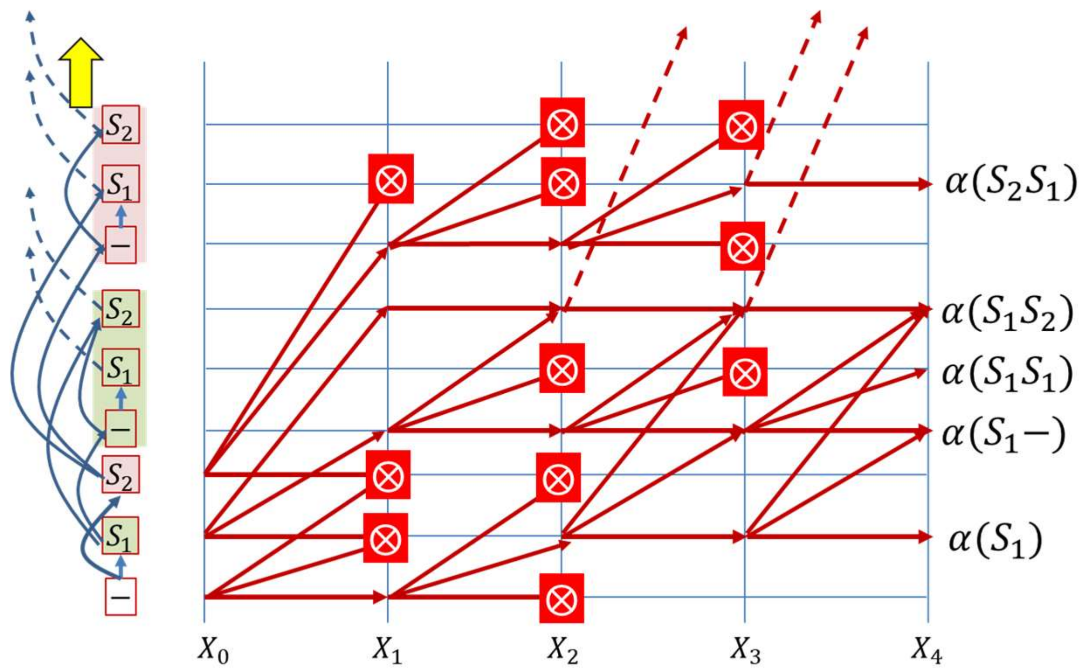
- $P(S_0, \ldots, S_{K-1} | \boldsymbol{X}) = \alpha(T-1, 2K) + \alpha(T-1, 2K+1)$

The decoding graph for the tree



- The forward score $\alpha(r, T)$ at the final time represents the full forward score for a unique symbol sequence (including sequences terminating in blanks)
- Select the symbol sequence with the largest alpha
  - Sequences may two alphas, one for the sequence itself, one for the sequence followed by a blank
  - Add the alphas before selecting the most likely

CTC decoding



- This is the “theoretically correct” CTC decoder
- In practice, the graph gets exponentially large very quickly
- To prevent this pruning strategies are employed to keep the graph (and computation) manageable
  - This may cause suboptimal decodes, however
  - The fact that CTC scores peak at symbol terminations minimizes the damage due to pruning

Beamsearch Pseudocode Notes

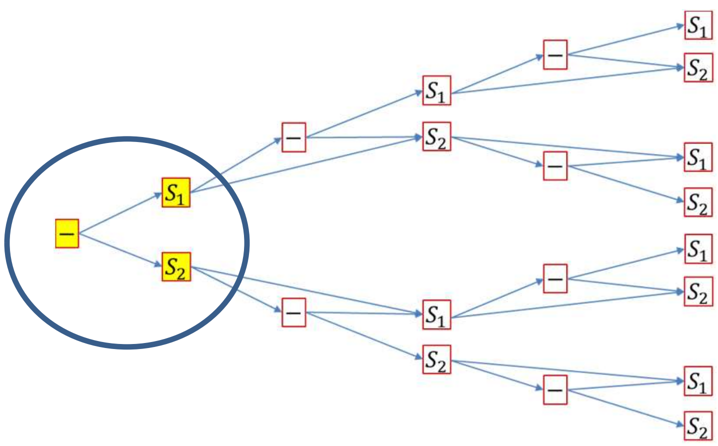
- Retaining separate lists of paths and pathscores for paths terminating in blanks, and those terminating in valid symbols
  - Since blanks are special
  - Do not explicitly represent blanks in the partial decode strings
- Pseudocode takes liberties (particularly w.r.t null strings)
  - I.e. you must be careful if you convert this to code
- Key
  - **`PathScore`** : array of scores for paths ending with symbols
  - **`BlankPathScore`** : array of scores for paths ending with blanks
  - **`SymbolSet`** : A list of symbols *not* including the blank

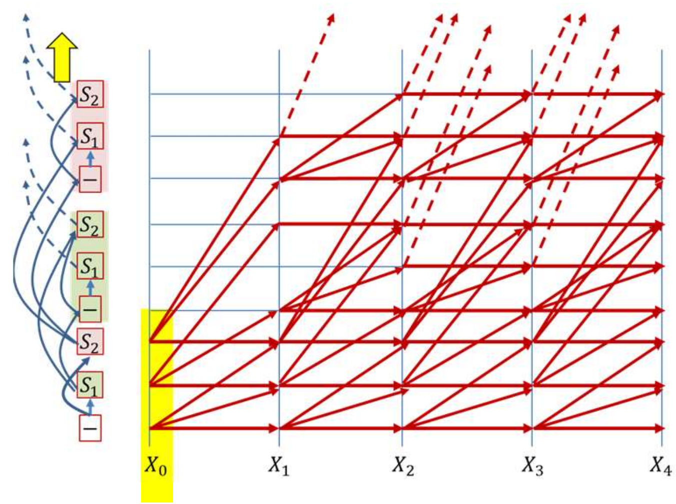
BEAM SEARCH

```
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols,
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore =
                           InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore =
                   Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol,
                                                 NewBlankPathScore, NewPathScore, BeamWidth)
    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank,
                                                               PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank,
                                                      PathsWithTerminalSymbol, SymbolSet, y[:,t])

end

# Merge identical paths differing only by the final blank
MergedPaths, FinalPathScore = MergeIdenticalPaths(NewPathsWithTerminalBlank, NewBlankPathScore
                                                  NewPathsWithTerminalSymbol, NewPathScore)

# Pick best path
BestPath = argmax(FinalPathScore) # Find the path with the best score
```

BEAM SEARCH

```
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols,
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore =
                          InitializePaths(SymbolSet, y[:,0])
```

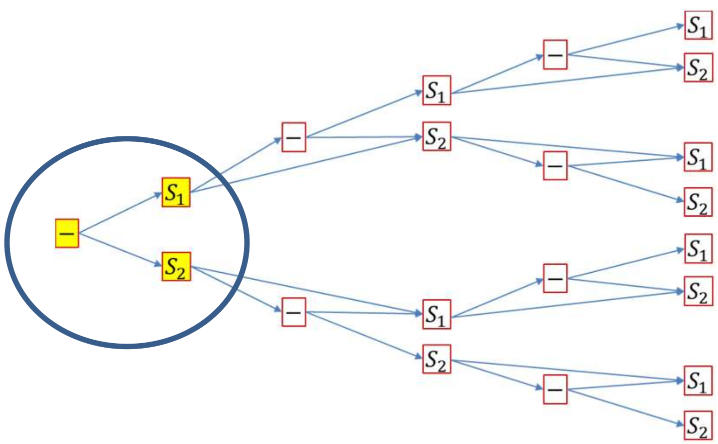


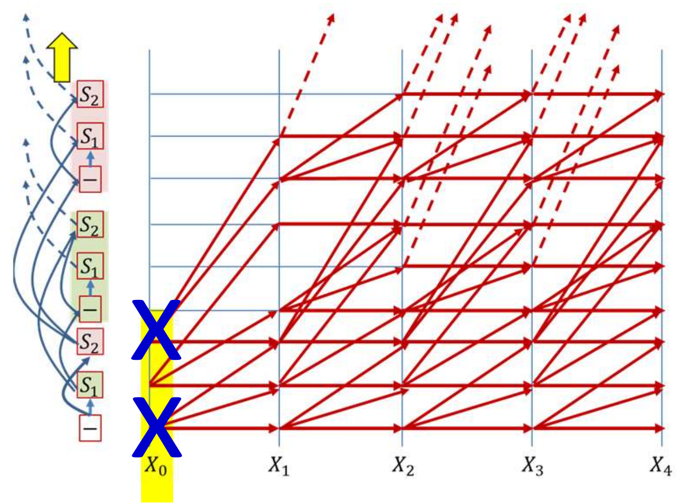


)

BEAM SEARCH

```
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols,
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore =
                           InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore =
                   Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol,
                                             NewBlankPathScore, NewPathScore, BeamWidth)
```

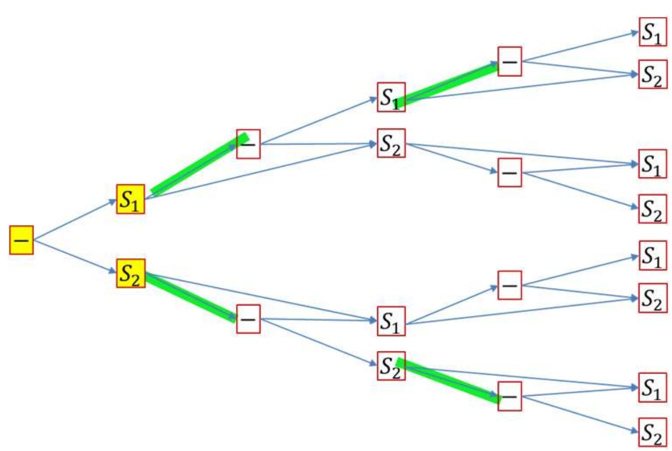


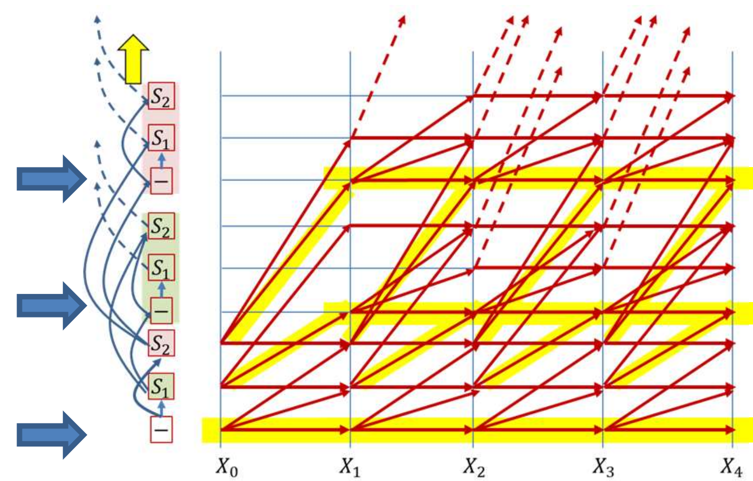


)

BEAM SEARCH

```
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols,
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore =
                          InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore =
                   Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol,
                                                    NewBlankPathScore, NewPathScore, BeamWidth)
    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank,
                                                             PathsWithTerminalSymbol, y[:,t])
```

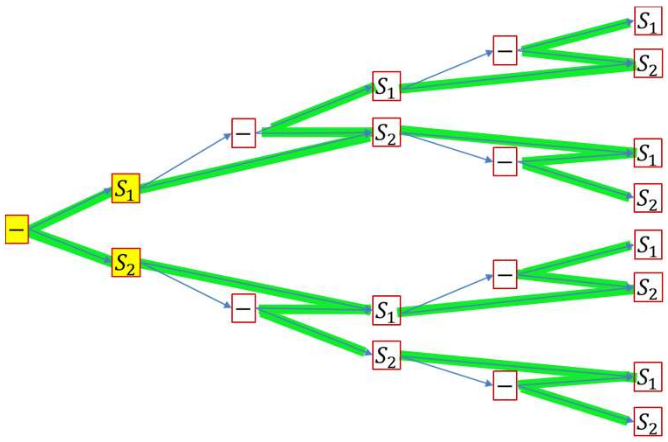


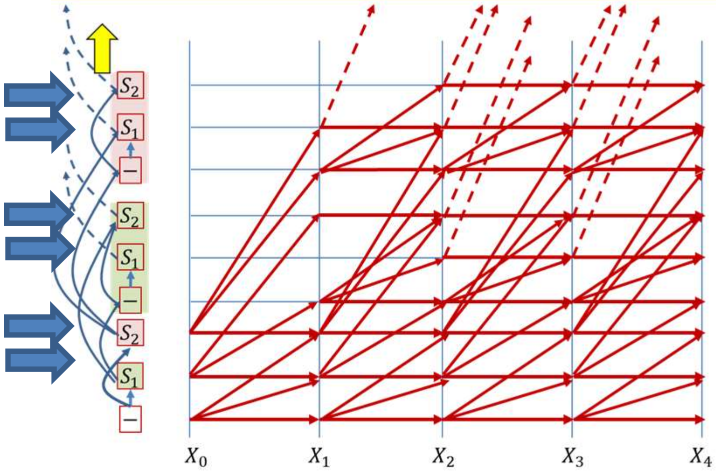


)

BEAM SEARCH

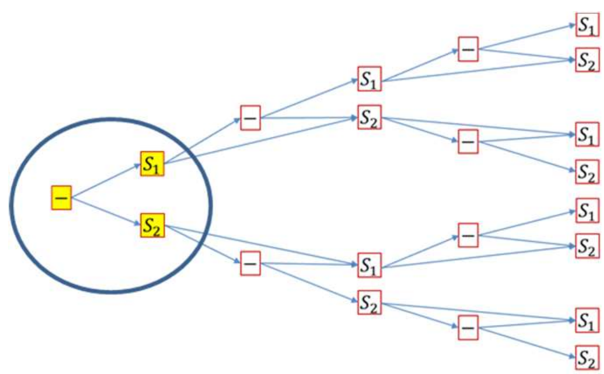
```
Global PathScore = [], BlankPathScore = []

# First time instant: Initialize paths with each of the symbols,
# including blank, using score at time t=1
NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol, NewBlankPathScore, NewPathScore =
                           InitializePaths(SymbolSet, y[:,0])

# Subsequent time steps
for t = 1:T
    # Prune the collection down to the BeamWidth
    PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore =
                   Prune(NewPathsWithTerminalBlank, NewPathsWithTerminalSymbol,
                                                  NewBlankPathScore, NewPathScore, BeamWidth)
    # First extend paths by a blank
    NewPathsWithTerminalBlank, NewBlankPathScore = ExtendWithBlank(PathsWithTerminalBlank,
                                                               PathsWithTerminalSymbol, y[:,t])

    # Next extend paths by a symbol
    NewPathsWithTerminalSymbol, NewPathScore = ExtendWithSymbol(PathsWithTerminalBlank,
                                                      PathsWithTerminalSymbol, SymbolSet, y[:,t])
```





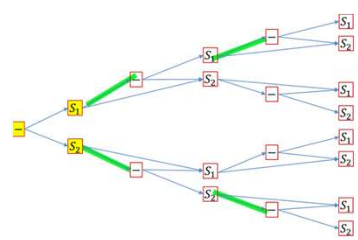
BEAM SEARCH InitializePaths: FIRST TIME INSTANT

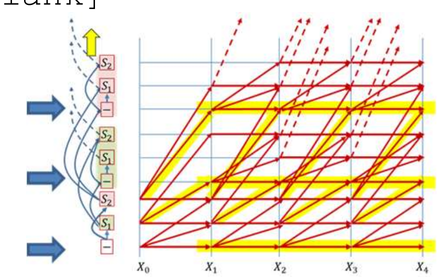
```
function InitializePaths(SymbolSet, y)

InitialBlankPathScore = [],  InitialPathScore = []
# First push the blank into a path-ending-with-blank stack. No symbol has been invoked yet
path = null
InitialBlankPathScore[path] = y[blank]  # Score of blank at t=1
InitialPathsWithFinalBlank = {path}

# Push rest of the symbols into a path-ending-with-symbol stack
InitialPathsWithFinalSymbol = {}
for c in SymbolSet  # This is the entire symbol set, without the blank
    path = c
    InitialPathScore[path] = y[c]  # Score of symbol c at t=1
    InitialPathsWithFinalSymbol += path # Set addition
end

return InitialPathsWithFinalBlank, InitialPathsWithFinalSymbol,
       InitialBlankPathScore, InitialPathScore
```



BEAM SEARCH: Extending with blanks

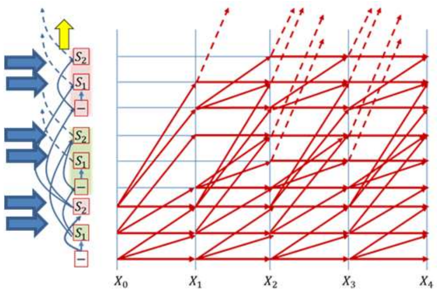
```
Global PathScore, BlankPathScore

function ExtendWithBlank(PathsWithTerminalBlank, PathsWithTerminalSymbol, y)
    UpdatedPathsWithTerminalBlank = {}
    UpdatedBlankPathScore = []
    # First work on paths with terminal blanks
    #(This represents transitions along horizontal trellis edges for blanks)
    for path in PathsWithTerminalBlank:
        # Repeating a blank doesn't change the symbol sequence
        UpdatedPathsWithTerminalBlank += path  # Set addition
        UpdatedBlankPathScore[path] = BlankPathScore[path]*y[blank]
    end

    # Then extend paths with terminal symbols by blanks
    for path in PathsWithTerminalSymbol:
        # If there is already an equivalent string in UpdatesPathsWithTerminalBlank
        # simply add the score. If not create a new entry
        if path in UpdatedPathsWithTerminalBlank
            UpdatedBlankPathScore[path] += Pathscore[path]* y[blank]
        else
            UpdatedPathsWithTerminalBlank += path # Set addition
            UpdatedBlankPathScore[path] = PathScore[path] * y[blank]
        end
    end

    return UpdatedPathsWithTerminalBlank,
           UpdatedBlankPathScore
```

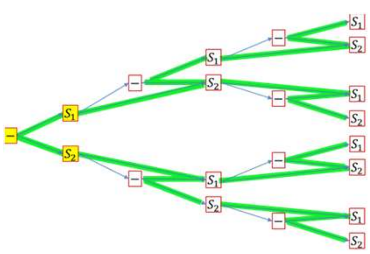




BEAM SEARCH: Extending with symbols

```
Global PathScore, BlankPathScore

function ExtendWithSymbol(PathsWithTerminalBlank, PathsWithTerminalSymbol, SymbolSet, y)
    UpdatedPathsWithTerminalSymbol = {}
    UpdatedPathScore = []

    # First extend the paths terminating in blanks. This will always create a new sequence
    for path in PathsWithTerminalBlank:
        for c in SymbolSet:  # SymbolSet does not include blanks
            newpath = path + c  # Concatenation
            UpdatedPathsWithTerminalSymbol += newpath # Set addition
            UpdatedPathScore[newpath] = BlankPathScore[path] * y(c)
        end
    end

    # Next work on paths with terminal symbols
    for path in PathsWithTerminalSymbol:
        # Extend the path with every symbol other than blank
        for c in SymbolSet:  # SymbolSet does not include blanks
            newpath = (c == path[end]) ? path : path + c  # Horizontal transitions don't extend the sequence
            if newpath in UpdatedPathsWithTerminalSymbol: # Already in list, merge paths
                UpdatedPathScore[newpath] += PathScore[path] * y[c]
            else # Create new path
                UpdatedPathsWithTerminalSymbol += newpath  # Set addition
                UpdatedPathScore[newpath] = PathScore[path] * y[c]
            end
        end
    end

    return UpdatedPathsWithTerminalSymbol,
           UpdatedPathScore
```





BEAM SEARCH: Pruning low-scoring entries

```
Global PathScore, BlankPathScore

function Prune(PathsWithTerminalBlank, PathsWithTerminalSymbol, BlankPathScore, PathScore, BeamWidth)
    PrunedBlankPathScore = []
    PrunedPathScore = []
    # First gather all the relevant scores
    i = 1
    for p in PathsWithTerminalBlank
        scorelist[i] = BlankPathScore[p]
        i++
    end
    for p in PathsWithTerminalSymbol
        scorelist[i] = PathScore[p]
        i++
    end

    # Sort and find cutoff score that retains exactly BeamWidth paths
    sort(scorelist)  # In decreasing order
    cutoff = BeamWidth < length(scorelist) ?  scorelist[BeamWidth] : scorelist[end]

    PrunedPathsWithTerminalBlank = {}
    for p in PathsWithTerminalBlank
        if BlankPathScore[p] >= cutoff
            PrunedPathsWithTerminalBlank += p # Set addition
            PrunedBlankPathScore[p] = BlankPathScore[p]
        end
    end

    PrunedPathsWithTerminalSymbol = {}
    for p in PathsWithTerminalSymbol
        if PathScore[p] >= cutoff
            PrunedPathsWithTerminalSymbol += p # Set addition
            PrunedPathScore[p] = PathScore[p]
        end
    end

    return PrunedPathsWithTerminalBlank, PrunedPathsWithTerminalSymbol, PrunedBlankPathScore, PrunedPathScore
```

BEAM SEARCH: Merging final paths

```
# Note : not using global variable here

function MergeIdenticalPaths(PathsWithTerminalBlank, BlankPathScore,
                              PathsWithTerminalSymbol, PathScore)

    # All paths with terminal symbols will remain
    MergedPaths = PathsWithTerminalSymbol
    FinalPathScore = PathScore

    # Paths with terminal blanks will contribute scores to existing identical paths from
    # PathsWithTerminalSymbol if present, or be included in the final set, otherwise
    for p in PathsWithTerminalBlank
        if p in MergedPaths
            FinalPathScore[p] += BlankPathScore[p]
        else
            MergedPaths += p # Set addition
            FinalPathScore[p] = BlankPathScore[p]
        end
    end

    return MergedPaths, FinalPathScore
```

Story so far: CTC models

- Sequence-to-sequence networks which irregularly produce output symbols can be trained by
  - Iteratively aligning the target output to the input and time-synchronous training
  - Optimizing the expected error over *all* possible alignments: CTC training
- Distinct repetition of symbols can be disambiguated from repetitions representing the extended output of a single symbol by the introduction of blanks
- Decoding the models can be performed by
  - Best-path decoding, i.e. Viterbi decoding
  - Optimal CTC decoding based on the application of the forward algorithm to a tree-structured representation of all possible output strings

CTC caveats

- The “blank” structure (with concurrent modifications to the forward-backward equations) is only one way to deal with the problem of repeating symbols

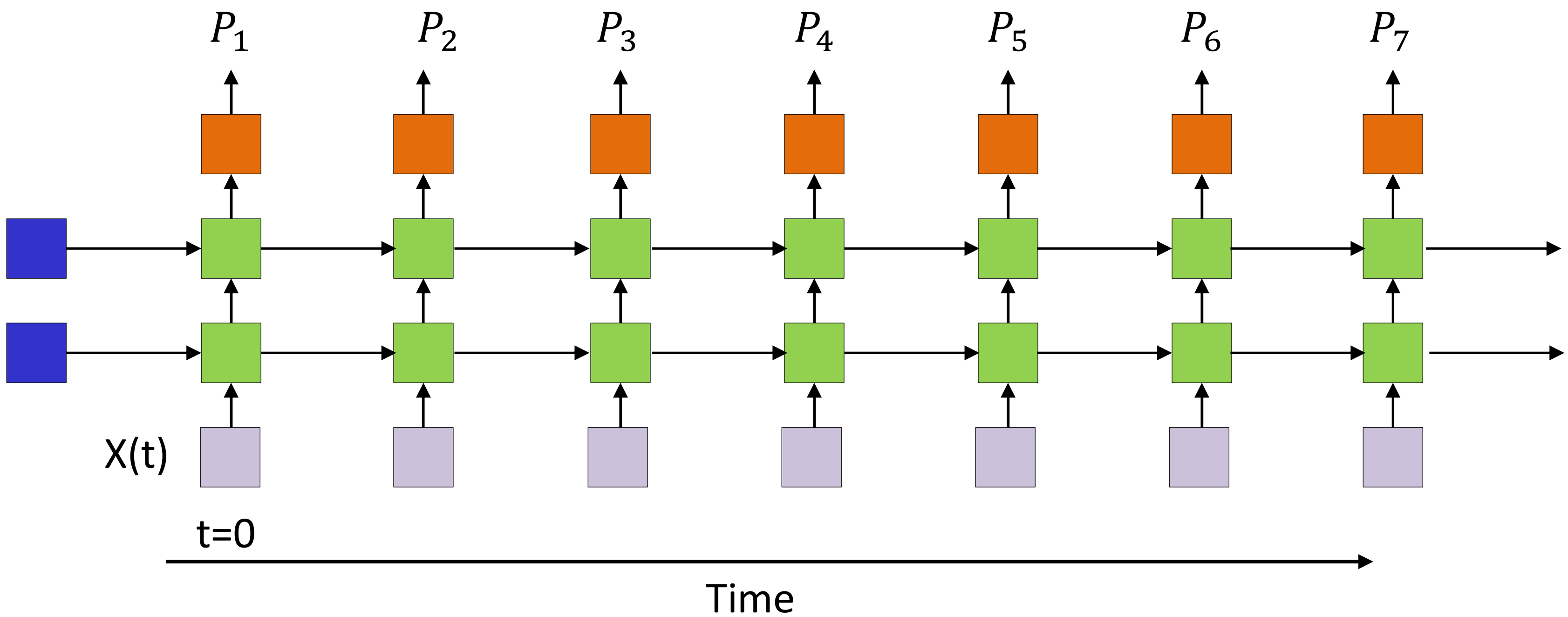
- Possible variants:
  - Symbols partitioned into two or more sequential subunits
    - No blanks are required, since subunits must be visited in order
  - Symbol-specific blanks
    - Doubles the “vocabulary”
  - CTC can use *bidirectional* recurrent nets
    - And frequently does
  - Other variants possible..

Most common CTC applications

- Speech recognition
  - Speech in, phoneme sequence out
  - Speech in, character sequence (spelling out)

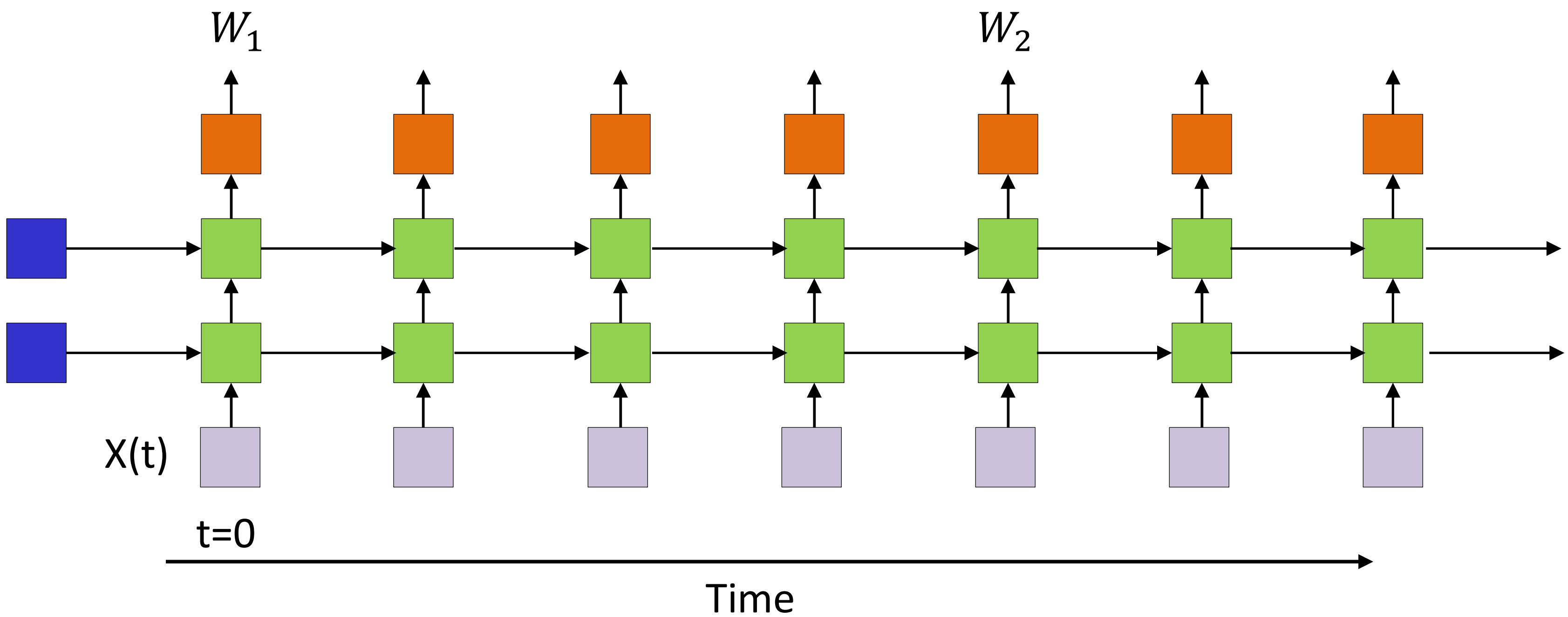
- Handwriting recognition

Speech recognition using Recurrent Nets



- Recurrent neural networks (with LSTMs) can be used to perform speech recognition
  - Input: Sequences of audio feature vectors
  - Output: Phonetic label of each vector

Speech recognition using Recurrent Nets



- Alternative: Directly output phoneme, character or word sequence

Next up: Attention models

CNN-LSTM-DNN for speech recognition

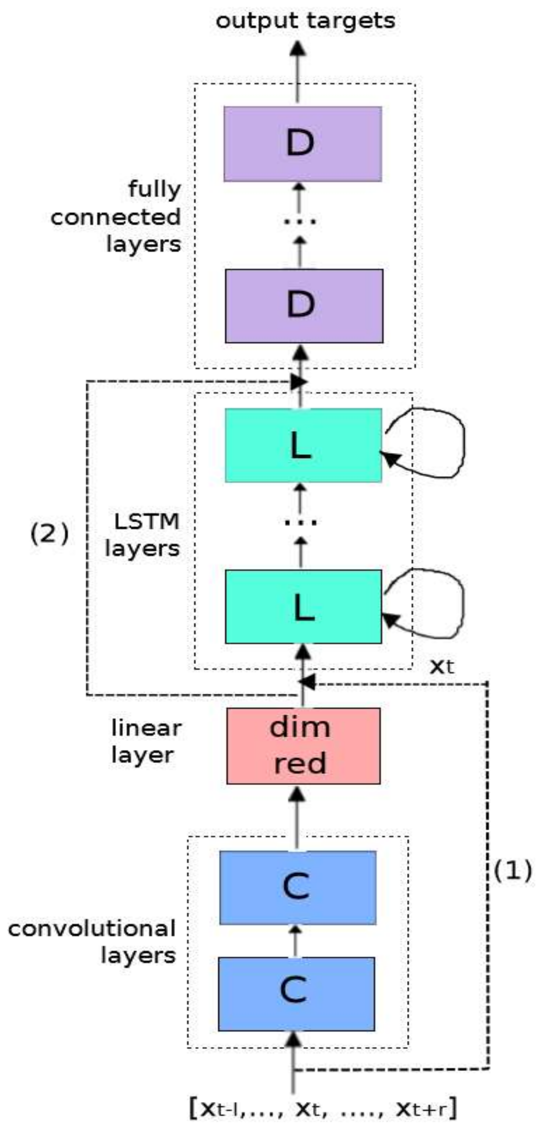
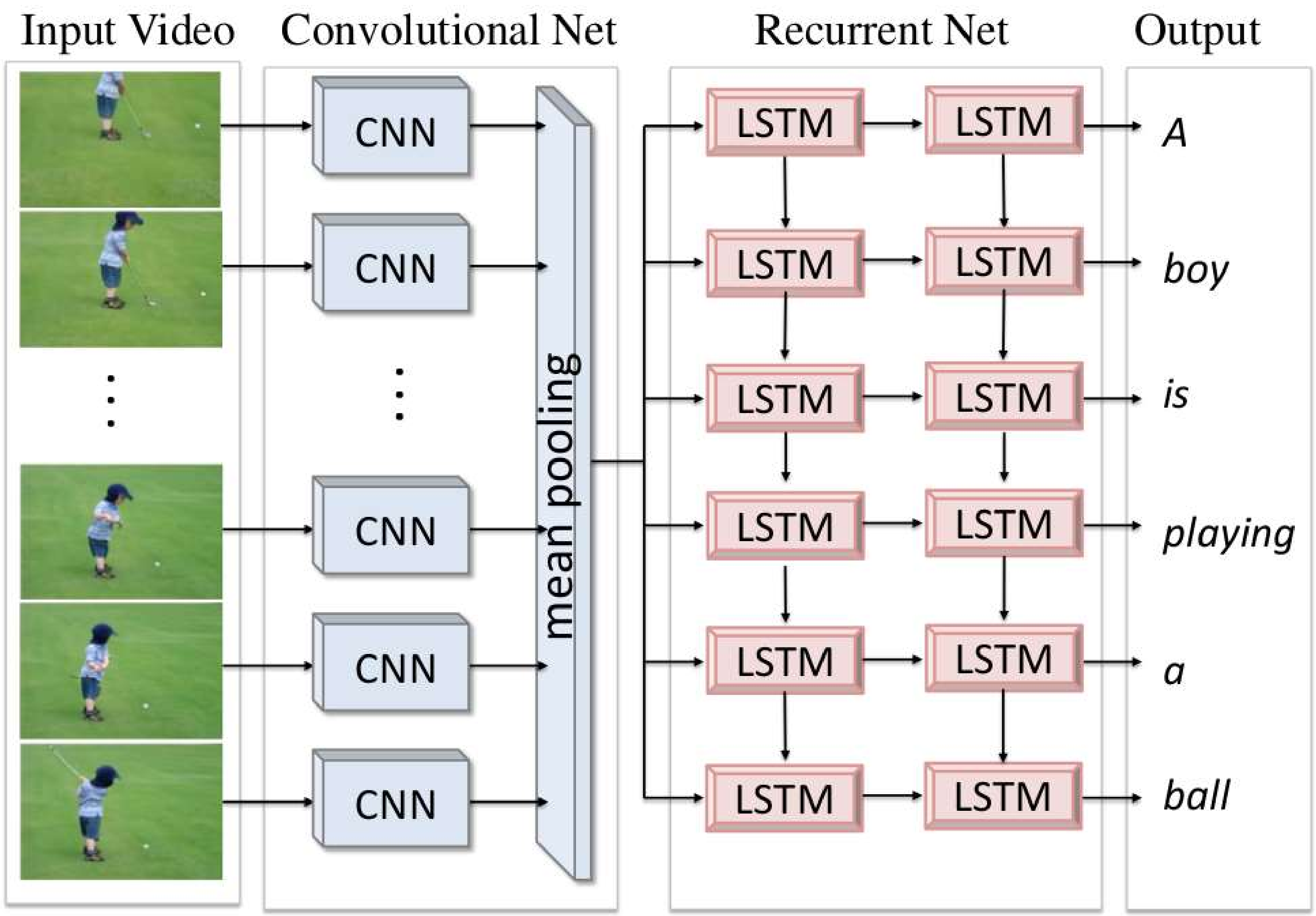


Fig. 1. CLDNN Architecture

Ensembles of RNN/LSTM, DNN, & Conv Nets (CNN) :

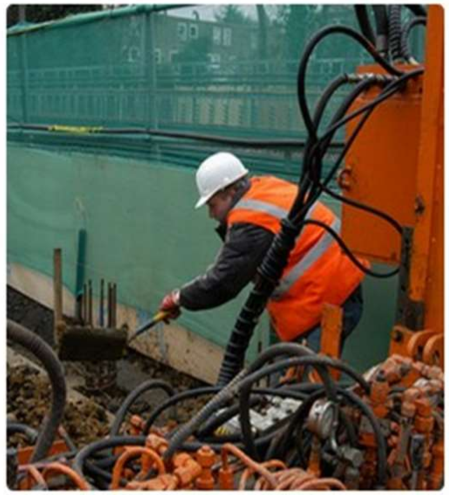
T. Sainath, O. Vinyals, A. Senior, H. Sak. "Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks," ICASSP 2015.

Translating Videos to Natural Language Using Deep Recurrent Neural Networks



Translating Videos to Natural Language Using Deep Recurrent Neural Networks
Subhashini Venugopalan, Huijun Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, Kate Saenko
North American Chapter of the Association for Computational Linguistics, Denver, Colorado, June 2015.

"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"boy is doing backflip on wakeboard."

"a young boy is holding a baseball bat."

"a cat is sitting on a couch with a remote control."

"a woman holding a teddy bear in front of a mirror."

"a horse is standing in the middle of a road."

Not explained

- Can be combined with CNNs
  - Lower-layer CNNs to extract features for RNN
- Can be used in tracking
  - Incremental prediction