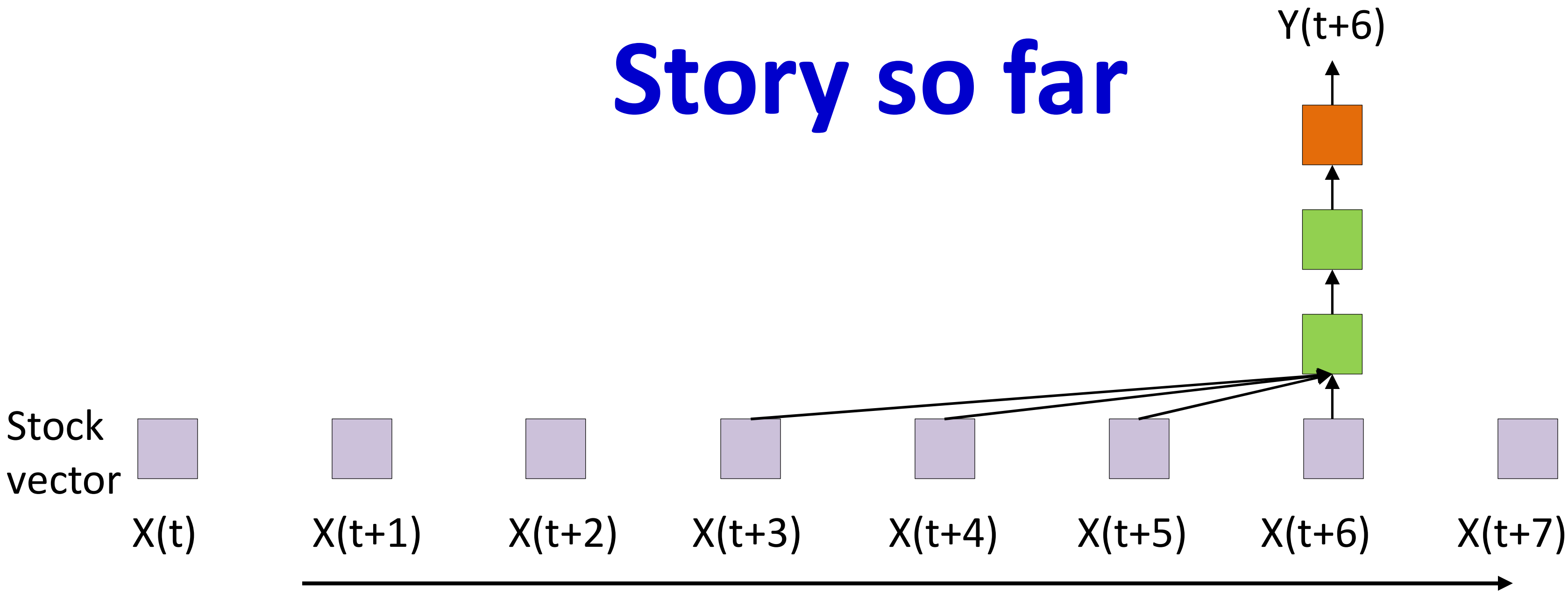
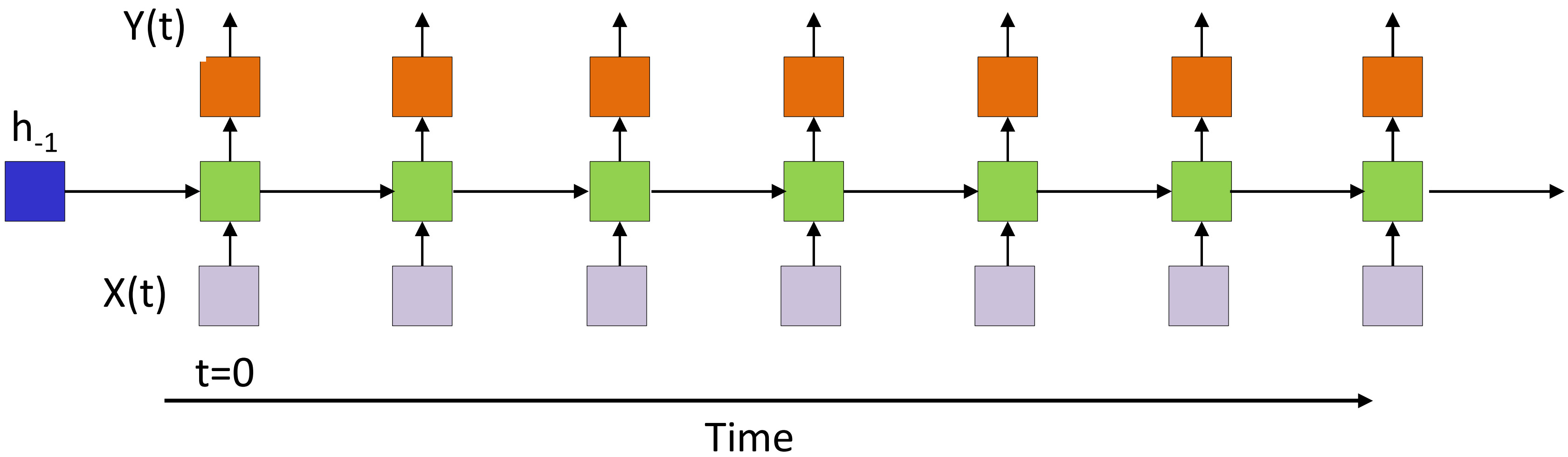
Deep Learning
Recurrent Networks: Part 3
Fall 2020

Story so far



- ***Iterated structures*** are good for analyzing time series data with short-time dependence on the past
  - These are "***Time delay***" neural nets, AKA ***convnets***

Story so far

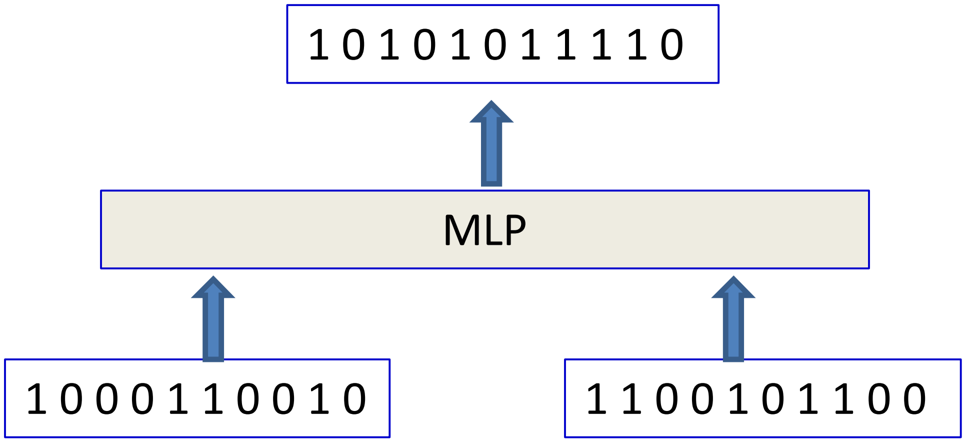


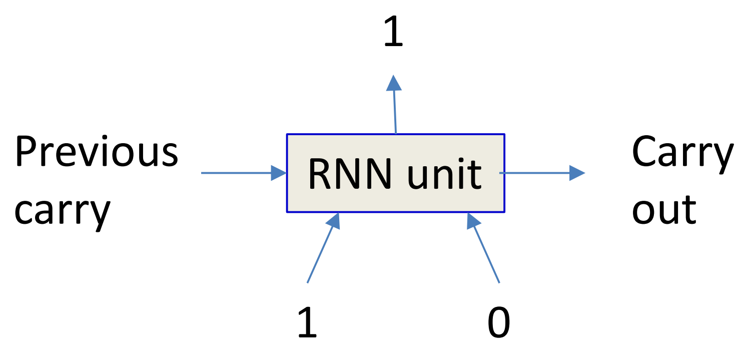
- Iterated structures are good for analyzing time series data with short-time dependence on the past
  - These are "Time delay" neural nets, AKA convnets
- ***Recurrent structures*** are good for analyzing time series data with ***long-term*** dependence on the past
  - These are ***recurrent*** neural networks

Recap: Recurrent networks can be incredibly effective at modeling long-term dependencies

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECON
  return segtable;
}
```

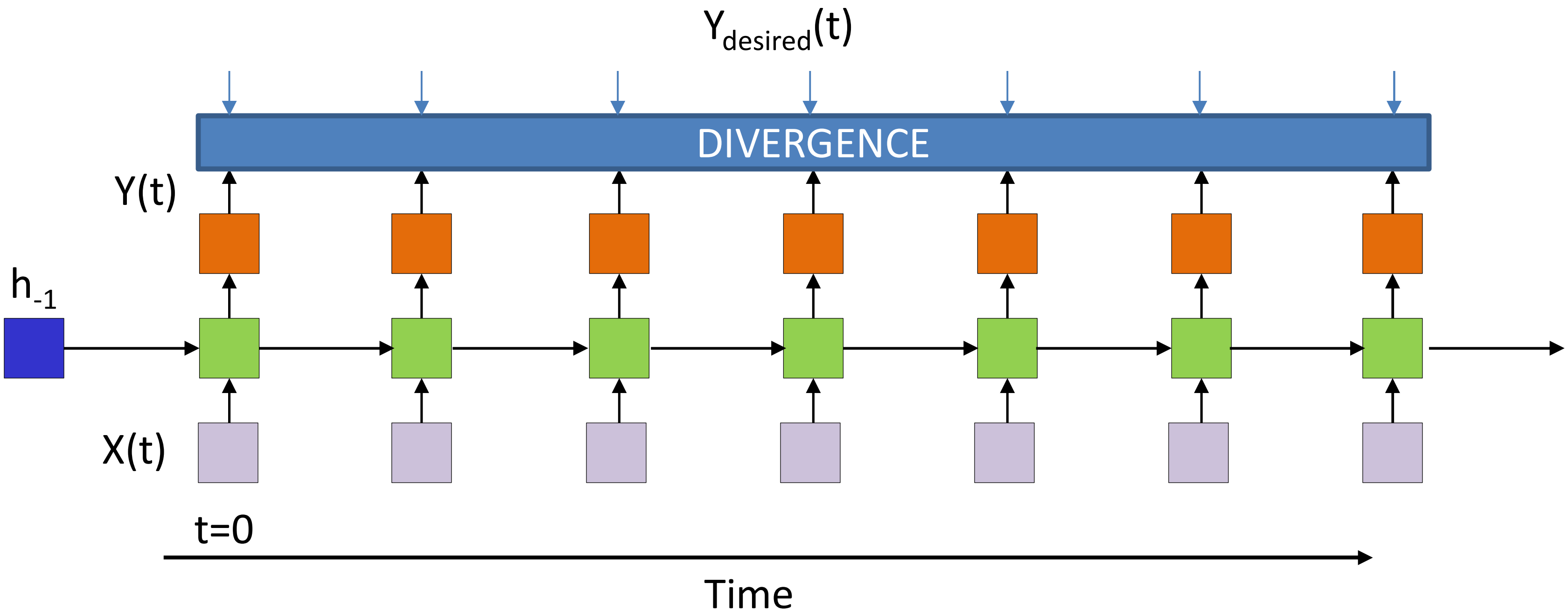
Recurrent structures can do what static structures cannot





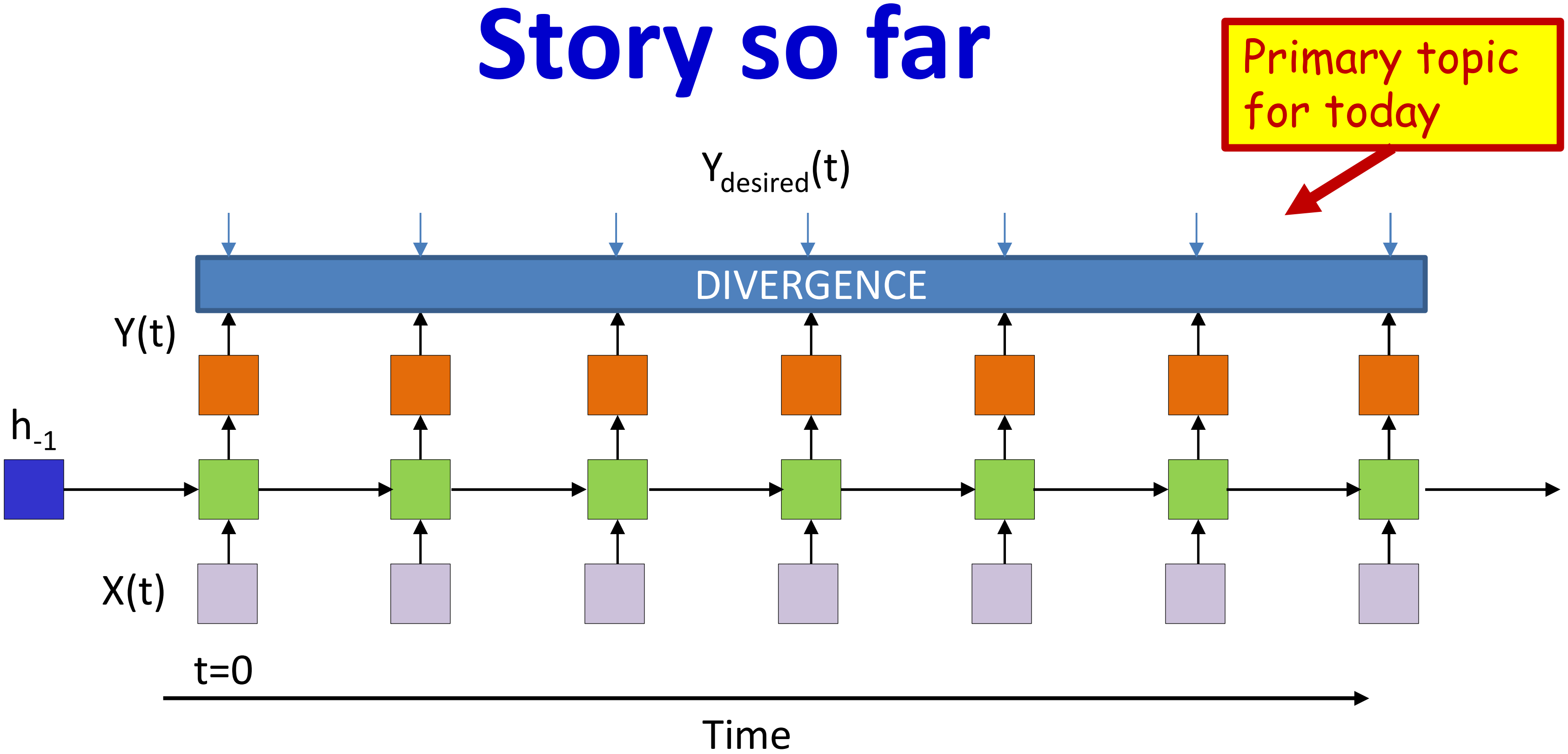
- The addition problem: Add two N-bit numbers to produce a N+1-bit number
  - Input is binary
  - Will require large number of training instances
    - Output must be specified for every pair of inputs
    - Weights that generalize will make errors
  - Network trained for N-bit numbers will not work for N+1 bit numbers
- An RNN learns to do this very quickly
  - With very little training data!

Story so far



- Recurrent structures can be trained by minimizing the divergence between the ***sequence*** of outputs and the ***sequence*** of desired outputs
  - Through gradient descent and backpropagation

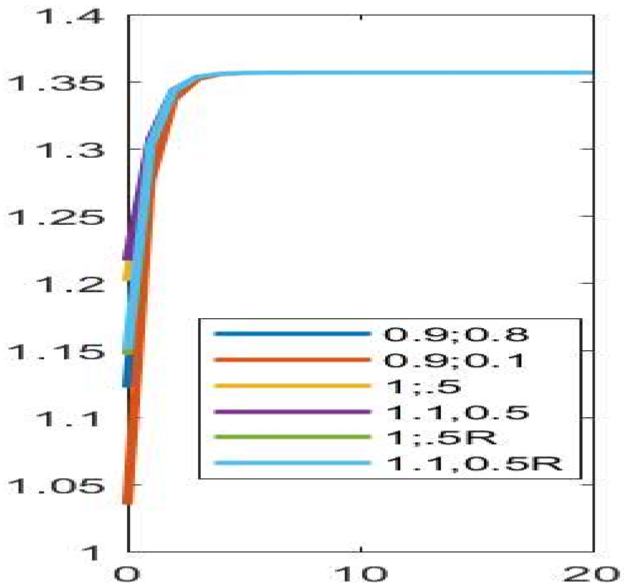
Story so far



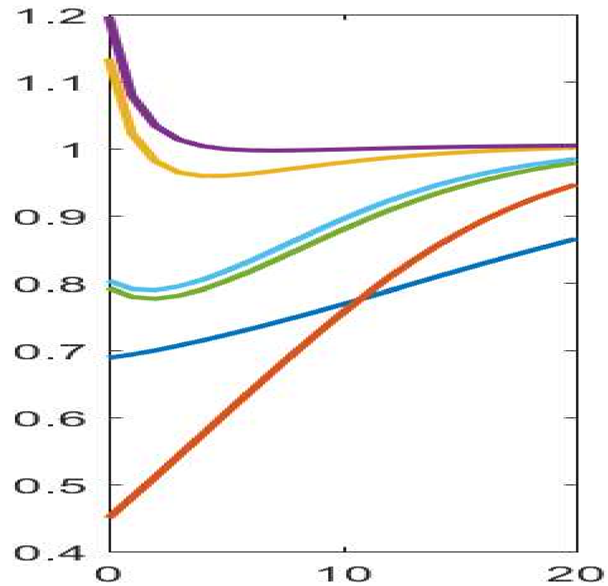
- Recurrent structures can be trained by minimizing the divergence between the ***sequence*** of outputs and the ***sequence*** of desired outputs
  - Through gradient descent and backpropagation

Story so far: stability

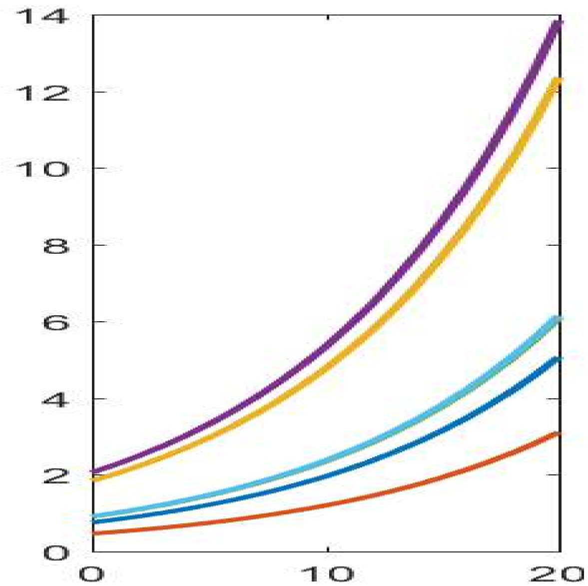
- Recurrent networks can be unstable
  - And not very good at remembering at other times



sigmoid



tanh

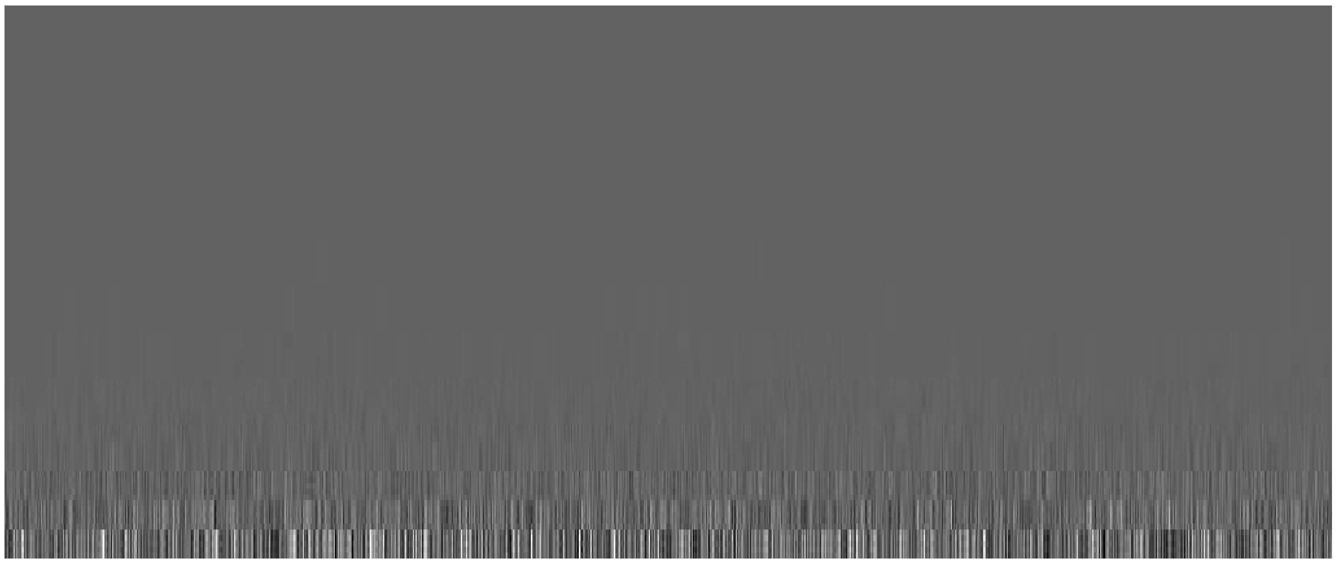


relu

Recap: Vanishing gradient examples..

ELU activation, Batch gradients

Input layer



Output layer

- Learning is difficult: gradients tend to vanish..

The long-term dependency problem

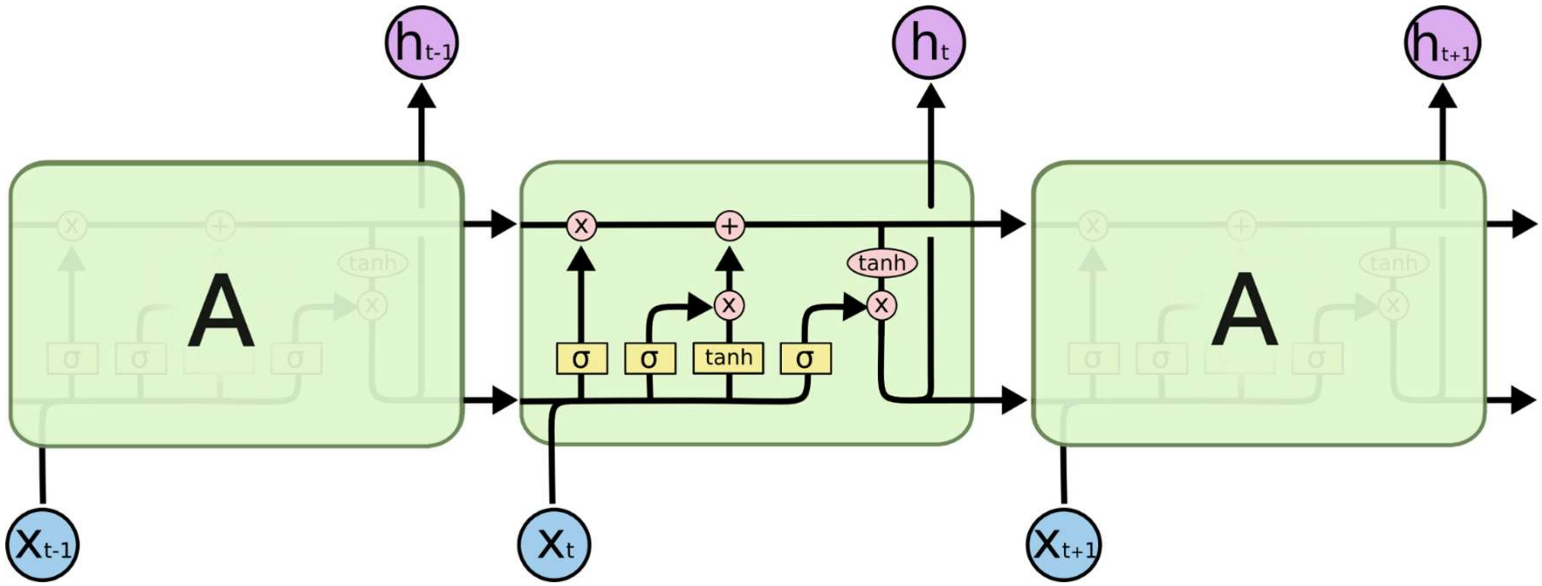
1

PATTERN1 [..............................] PATTERN 2

Jane had a quick lunch in the bistro. Then *she..*

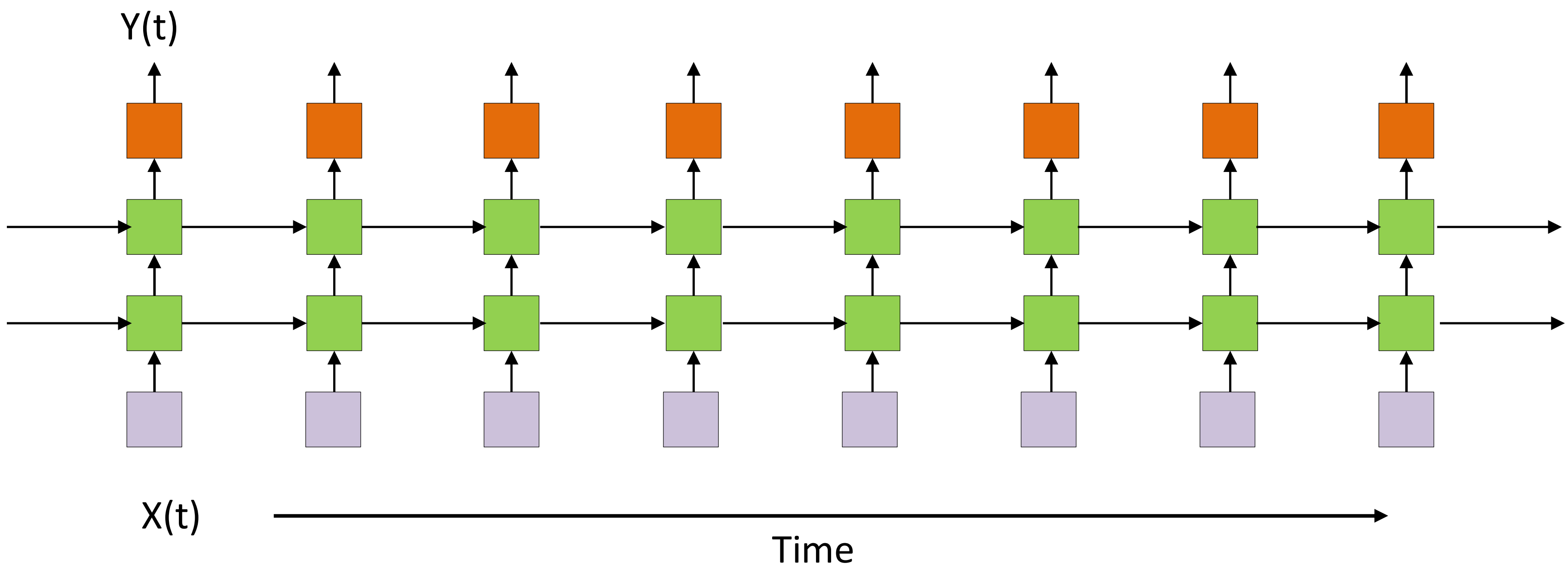
- Long-term dependencies are hard to learn in a network where memory behavior is an untriggered function of the *network*
  - Need it to be a triggered response to *input*

Long Short-Term Memory



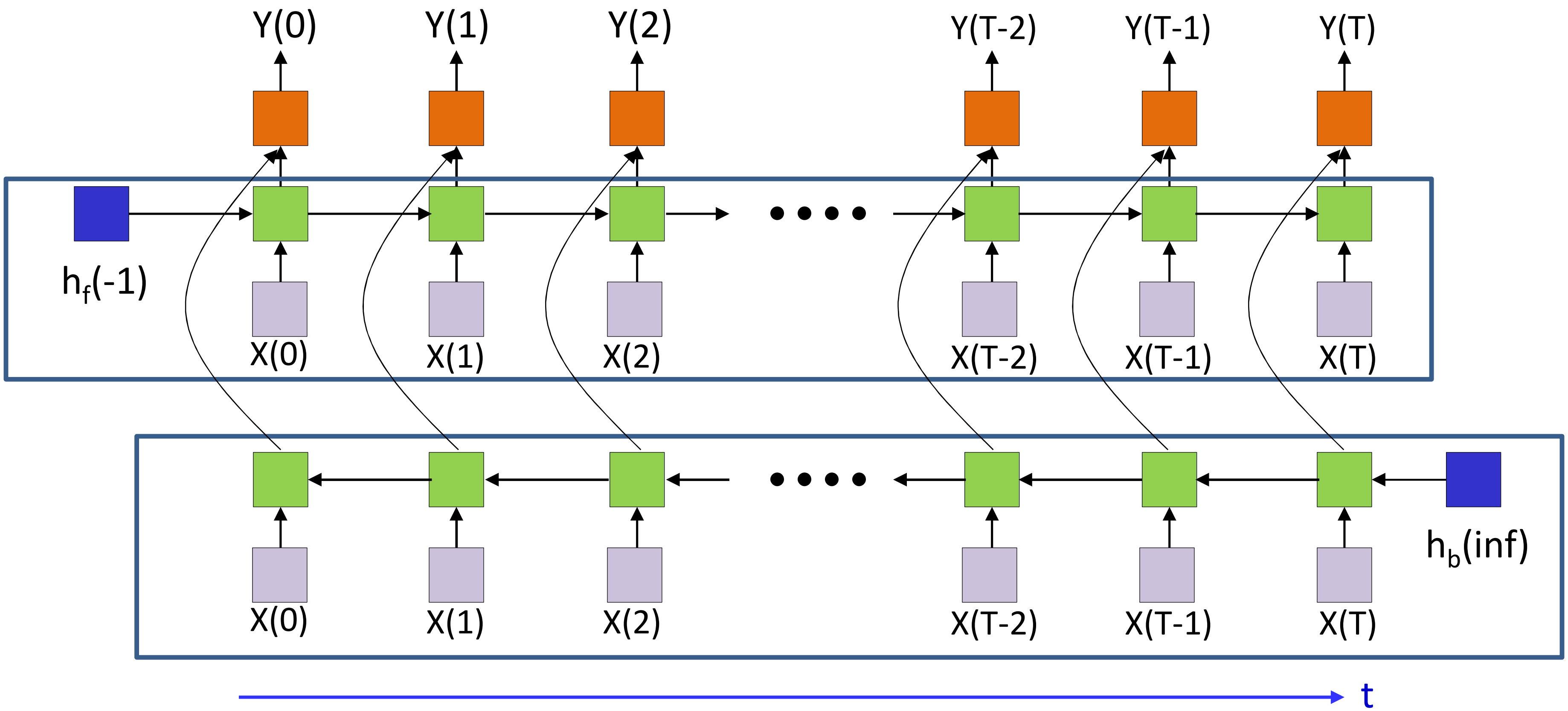
- The LSTM addresses the problem of *input-dependent* memory behavior

Recap: LSTM-based architecture



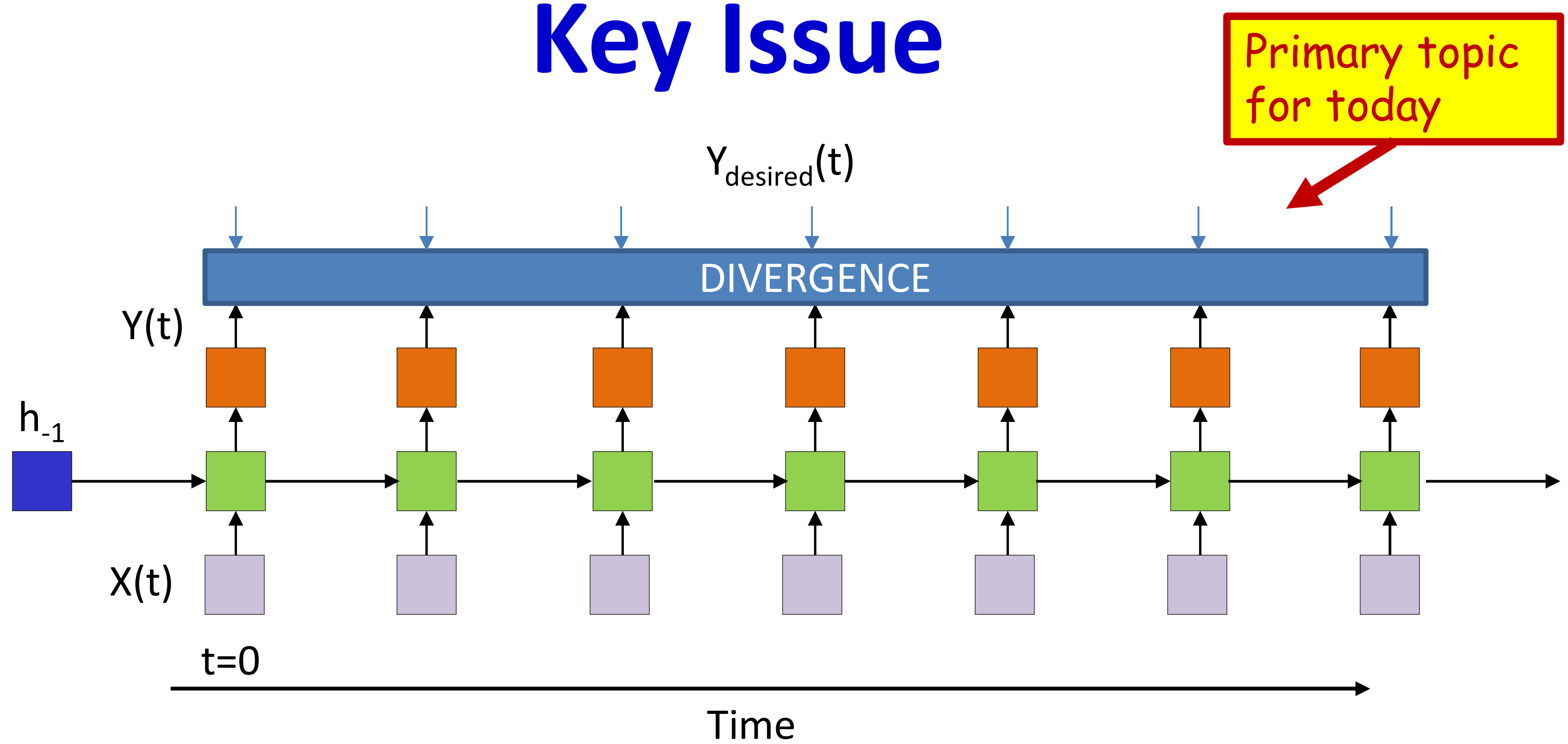
- LSTM based architectures are identical to RNN-based architectures

Recap: Bidirectional LSTM



- Bidirectional version..

Key Issue



- How do we define the divergence
- Also: how do we compute the outputs..

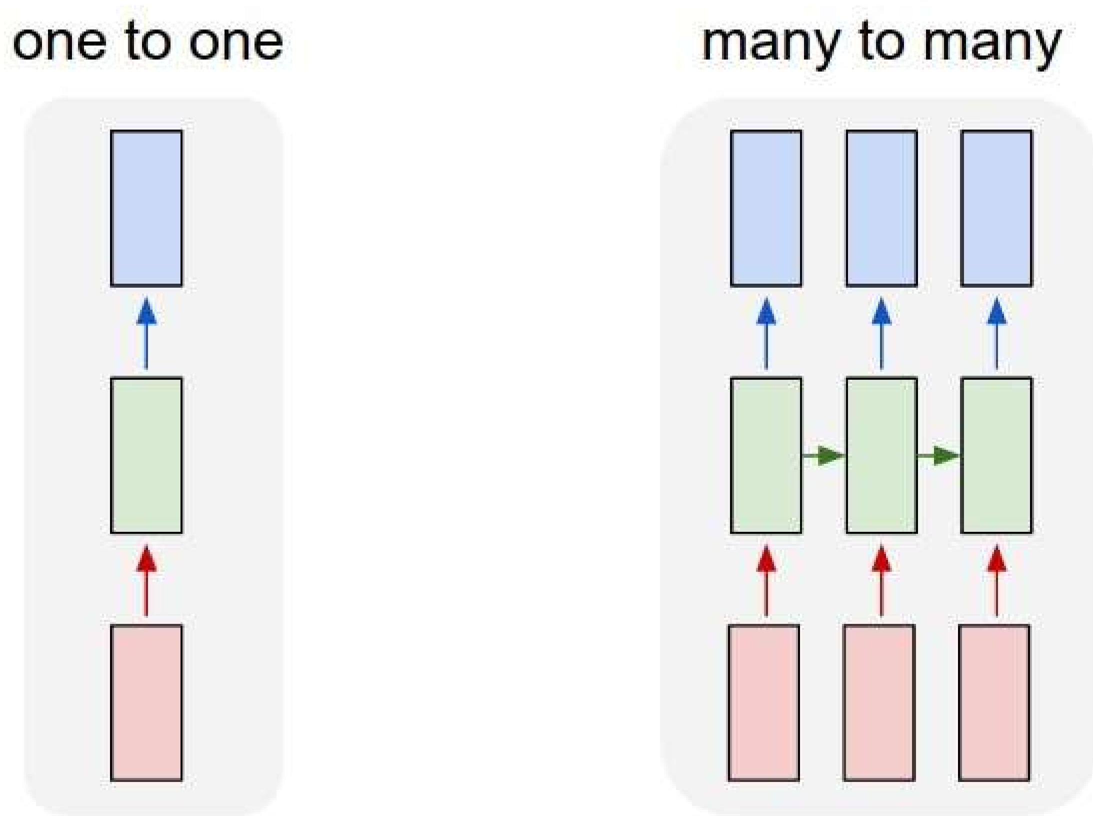
What follows in this series on recurrent nets

- Architectures: How to train recurrent networks of different architectures

- Synchrony: How to train recurrent networks when
  - The target output is time-synchronous with the input
  - The target output is order-synchronous, but not time synchronous
  - Applies to only some types of nets

- How to make predictions/inference with such networks

Variants of recurrent nets

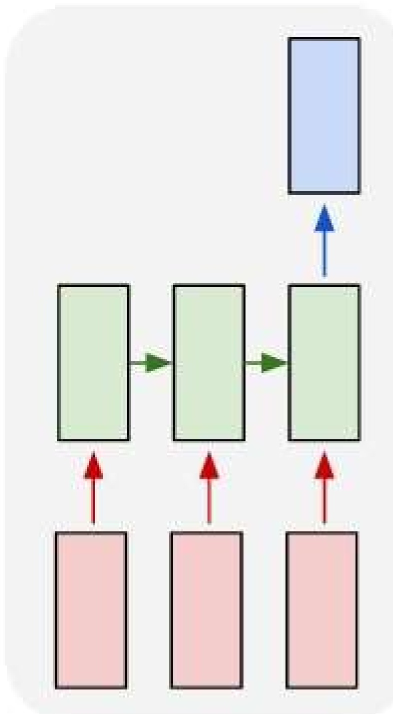


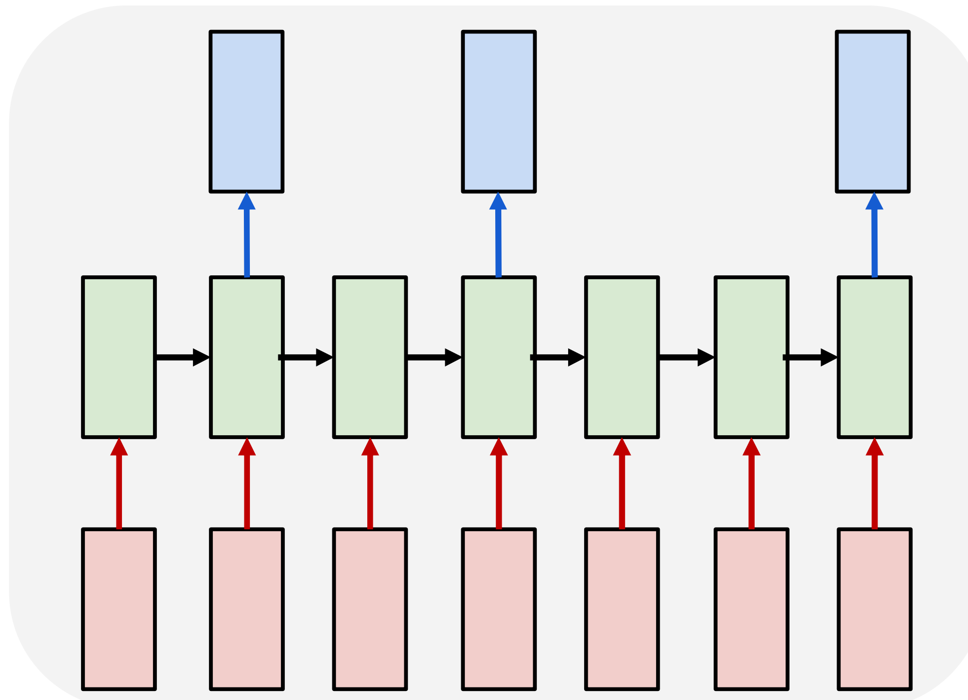
Images from Karpathy

- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Variants of recurrent nets

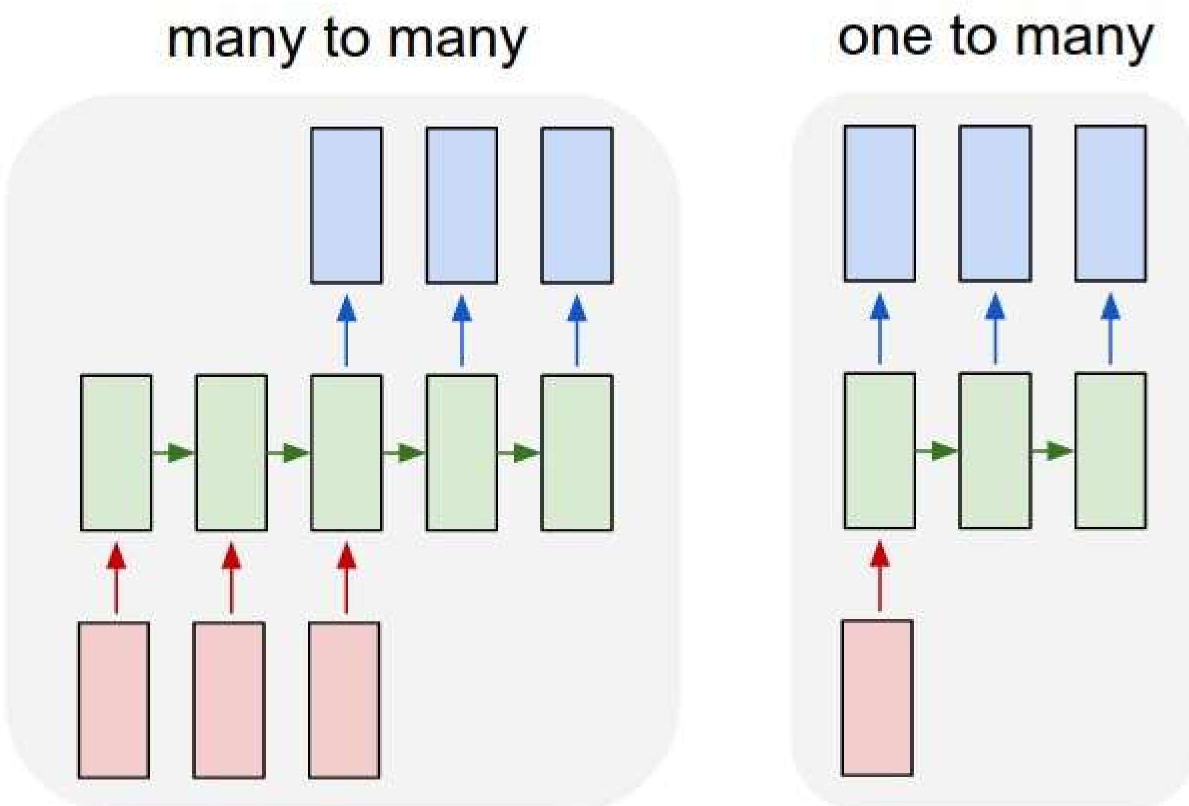
many to one





- Sequence classification: Classifying a full input sequence
  - E.g isolated word/phrase recognition
- Order synchronous , time asynchronous sequence-to-sequence generation
  - E.g. speech recognition
  - Exact location of output is unknown a priori

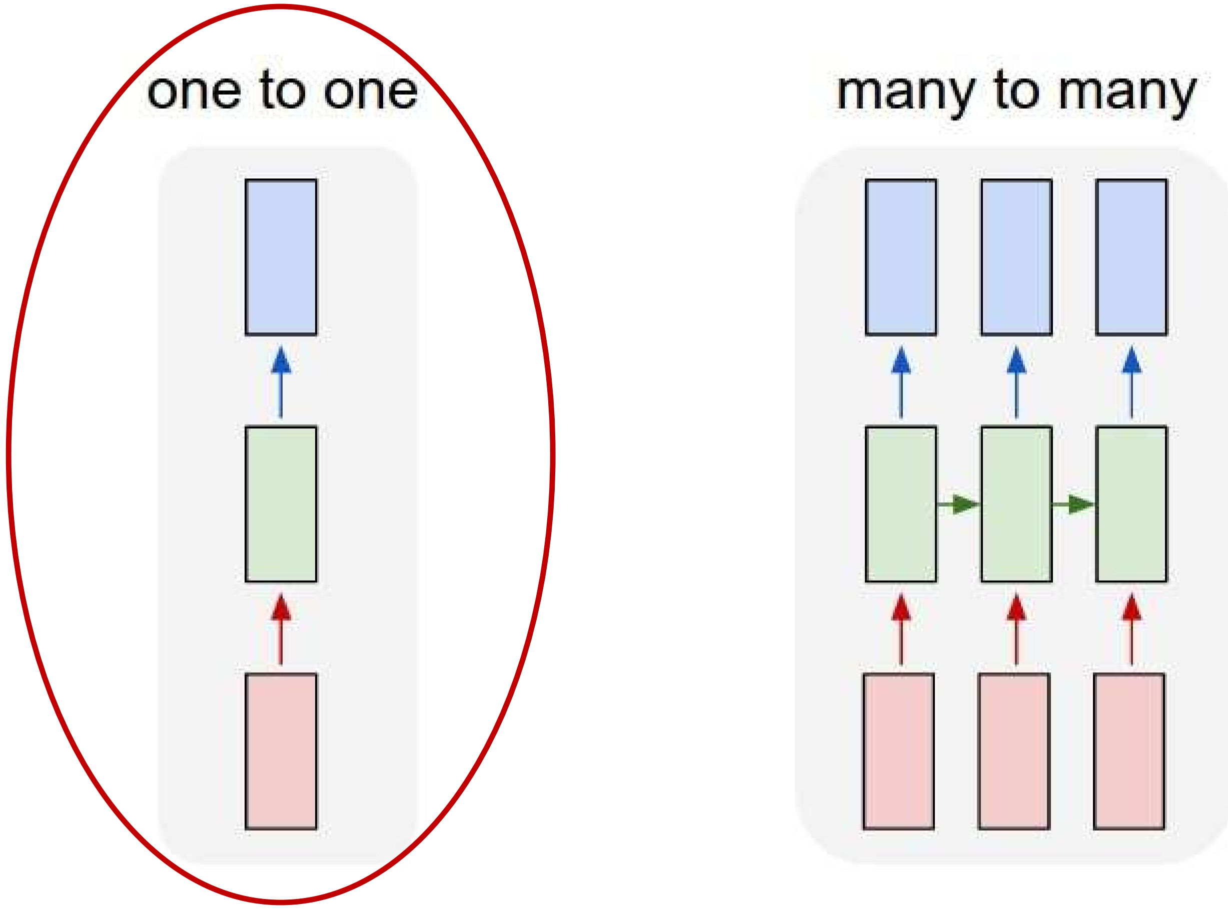
More variants



Images from Karpathy

- A posteriori sequence to sequence: Generate output sequence after processing input
  - E.g. language translation
- Single-input a posteriori sequence generation
  - E.g. captioning an image

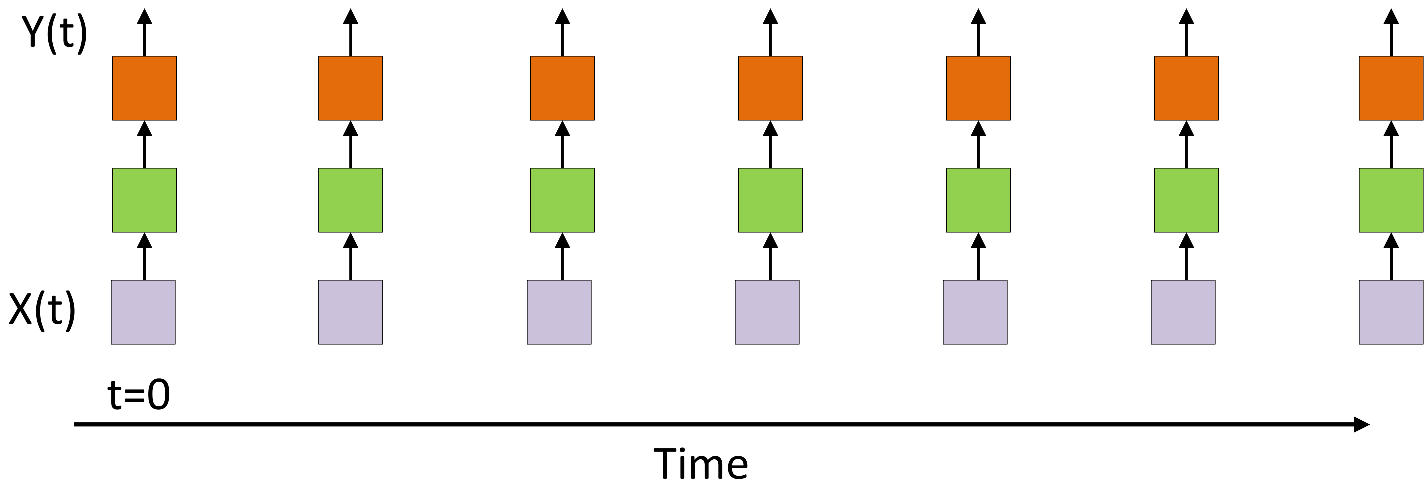
Variants of recurrent nets



Images from Karpathy

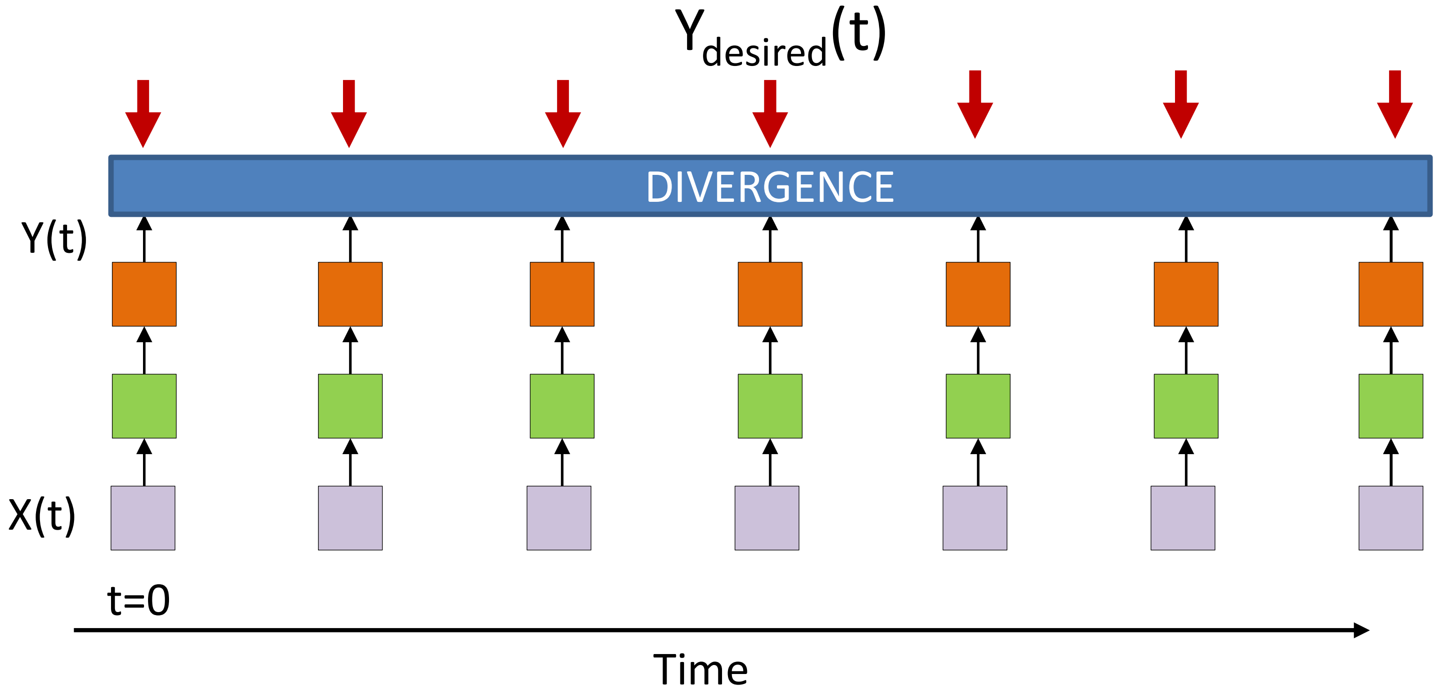
- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Regular MLP for processing sequences



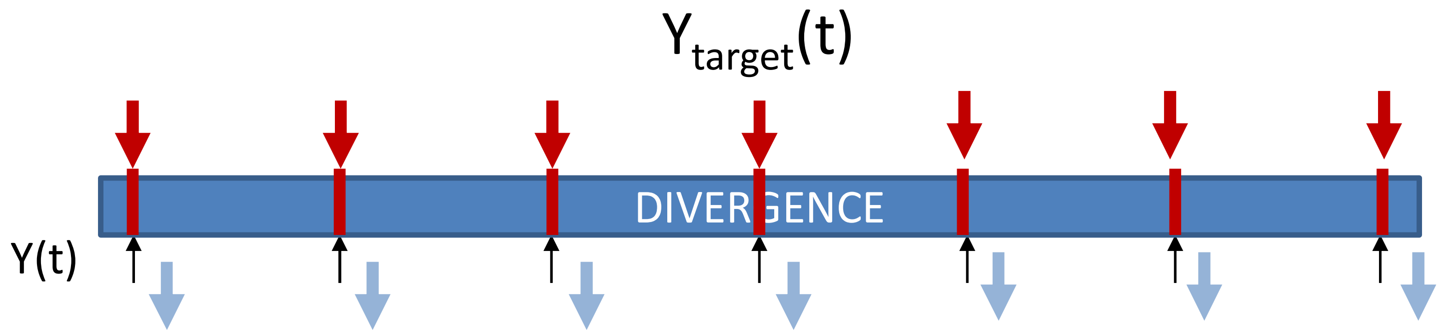
- No recurrence in model
  - Exactly as many outputs as inputs
  - Every input produces a unique output
  - The output at time t is unrelated to the output at $t' \neq t$

Learning in a Regular MLP



- No recurrence
  - Exactly as many outputs as inputs
    - **One to one correspondence between desired output and actual output**
  - The output at time t is unrelated to the output at $t' \neq t$.

Regular MLP



- Gradient backpropagated at each time

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big)$$

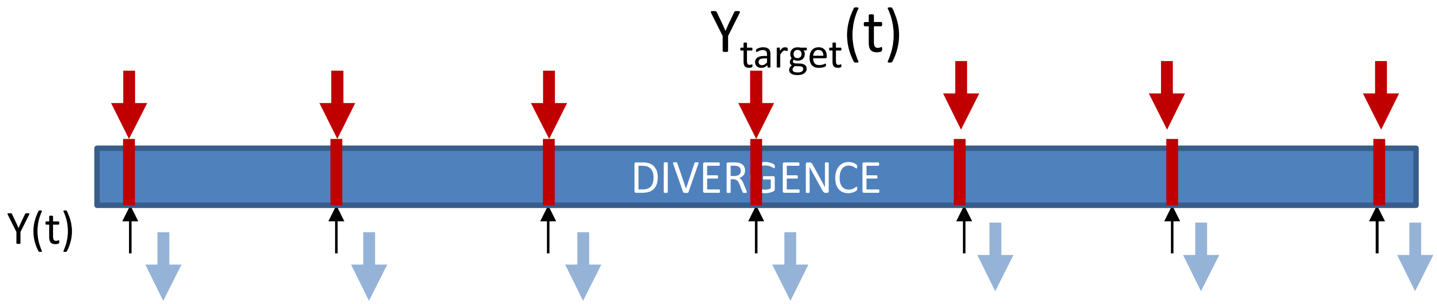
- Common assumption:

$$Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \sum_t w_t Div\big(Y_{target}(t), Y(t)\big)$$

$$\nabla_{Y(t)} Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = w_t \nabla_{Y(t)} Div\big(Y_{target}(t), Y(t)\big)$$

  - w_t is typically set to 1.0
  - This is further backpropagated to update weights etc

Regular MLP



- Gradient backpropagated at each time

$$\nabla_{Y(t)} Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right)$$

- Common assumption:

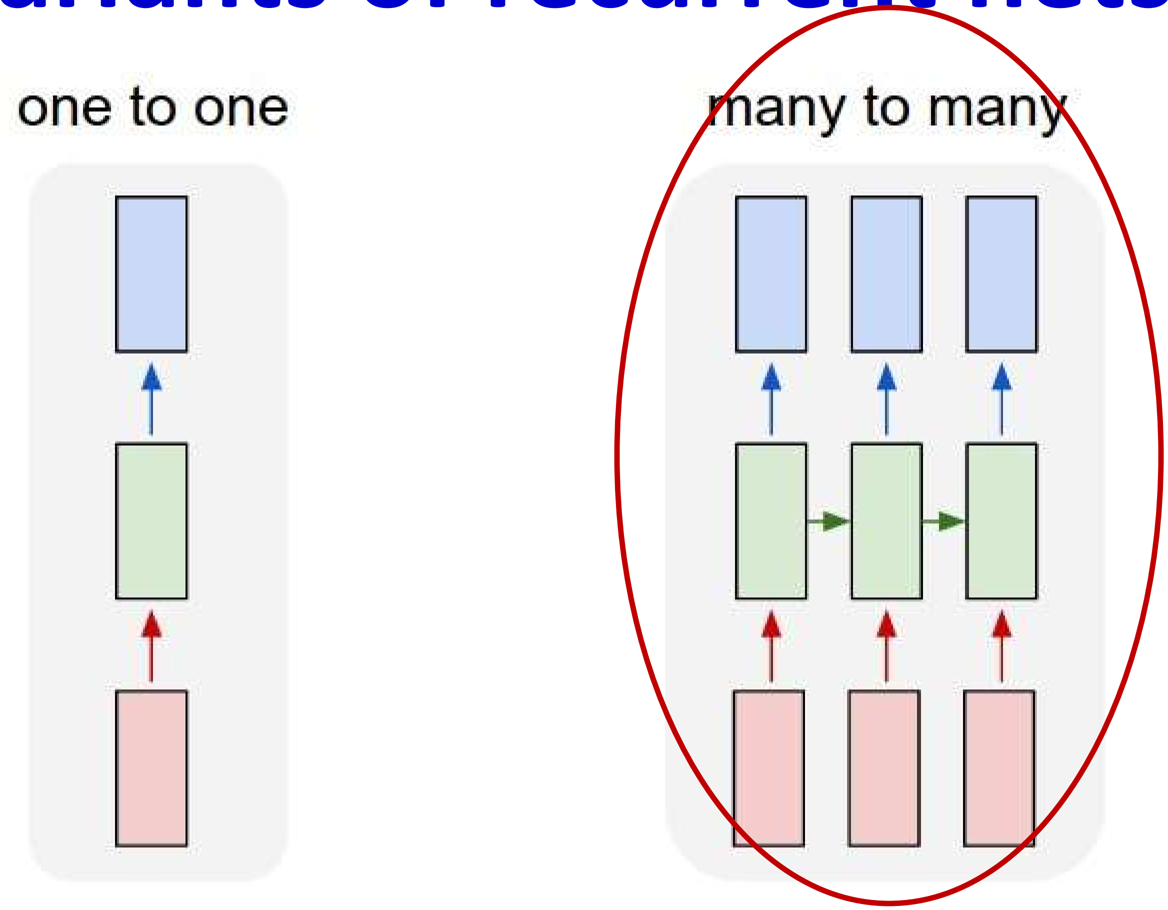
$$Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \sum_t Div\left(Y_{target}(t), Y(t)\right)$$

$$\nabla_{Y(t)} Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \nabla_{Y(t)} Div\left(Y_{target}(t), Y(t)\right)$$

  – This is further backpropagated to update weights etc

Typical Divergence for classification: $Div\left(Y_{target}(t), Y(t)\right) = KL(Y_{target}(t), Y(t))$

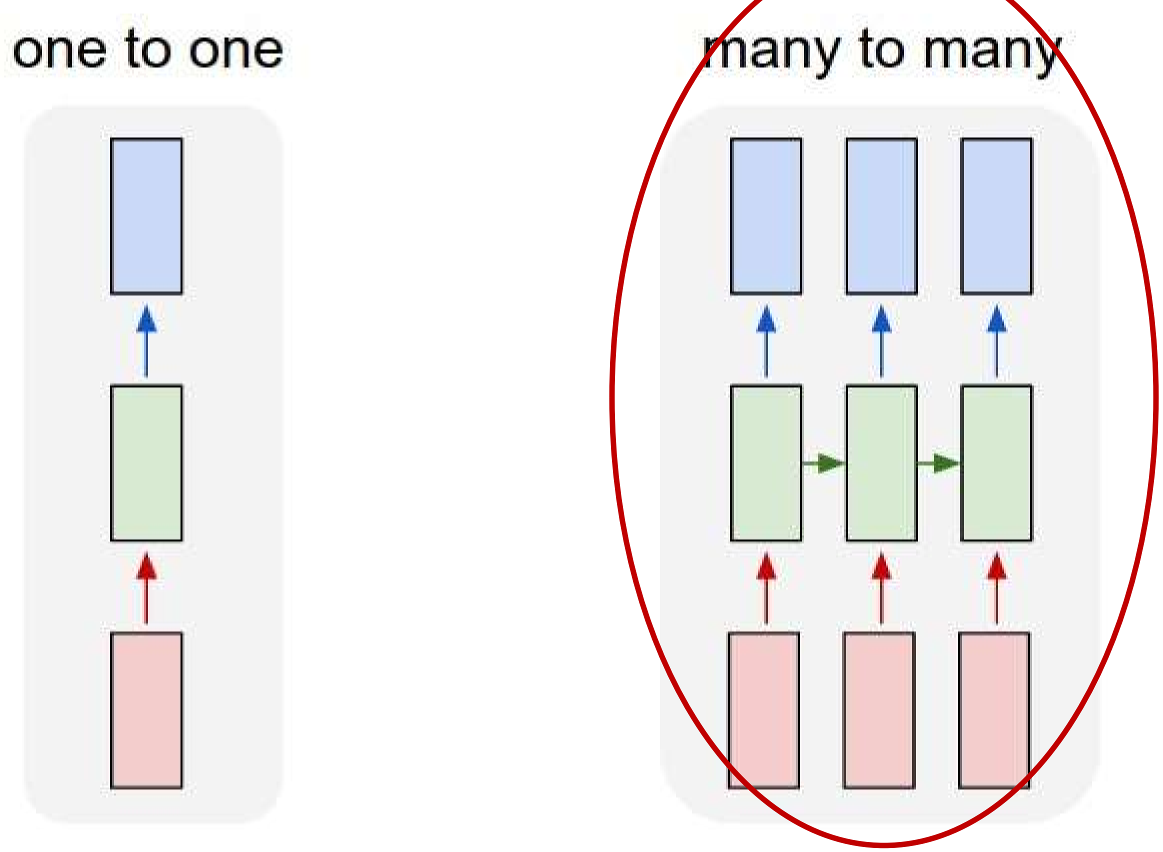
Variants of recurrent nets



Images from Karpathy

- Conventional MLP
- Time-synchronous outputs
  - E.g. part of speech tagging

Variants of recurrent nets

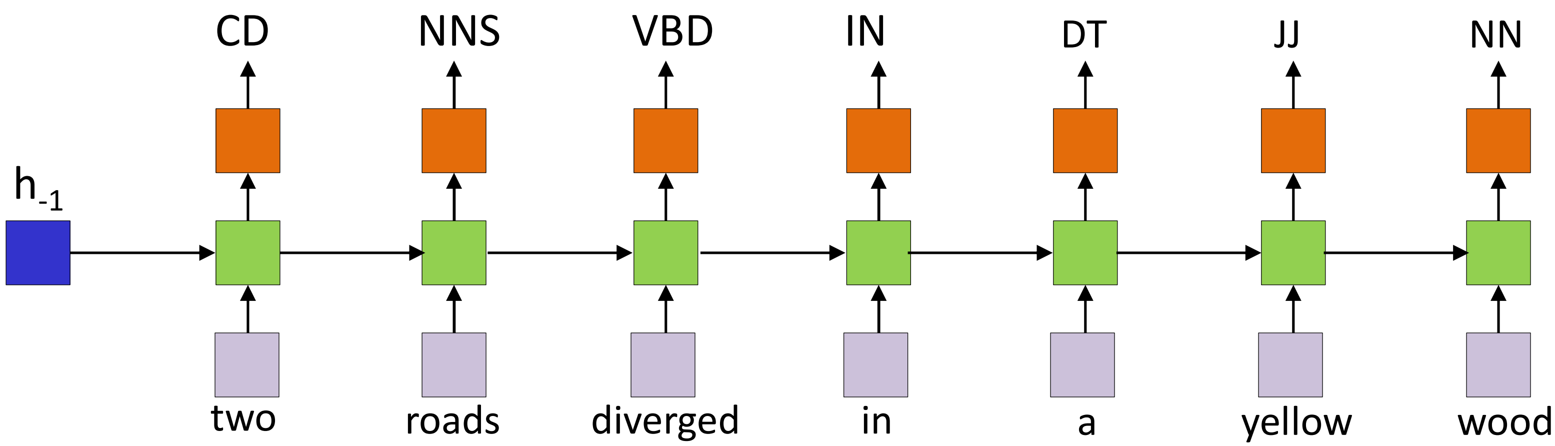


Images from Karpathy

With a brief detour into modelling language

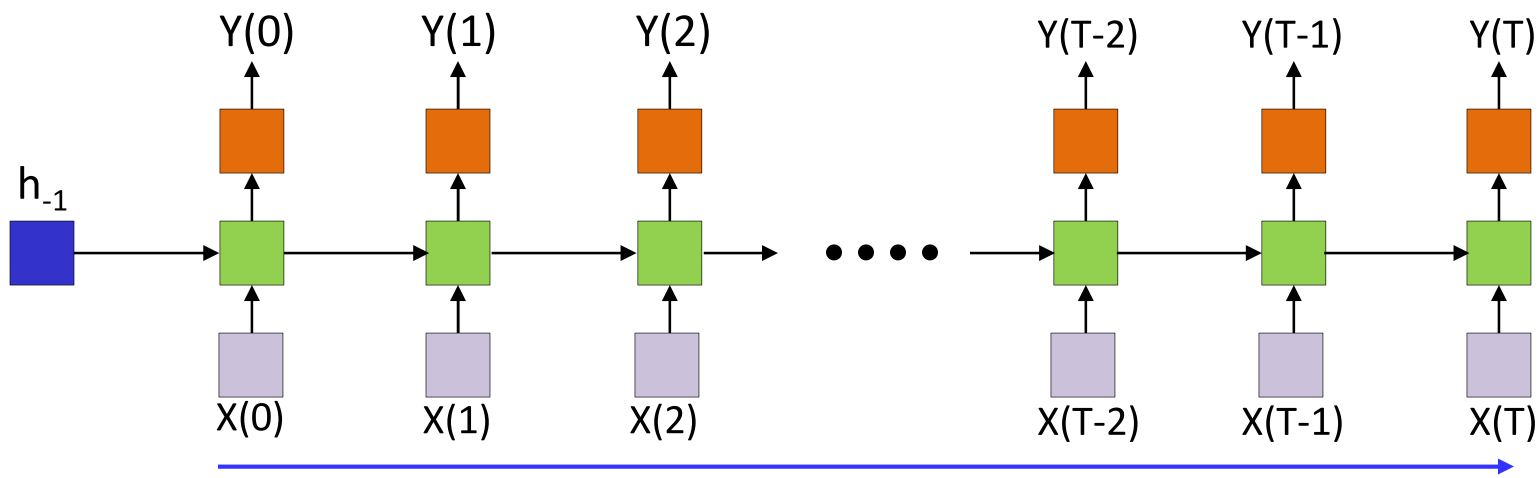
- Time-synchronous outputs
  - E.g. part of speech tagging

Time synchronous network



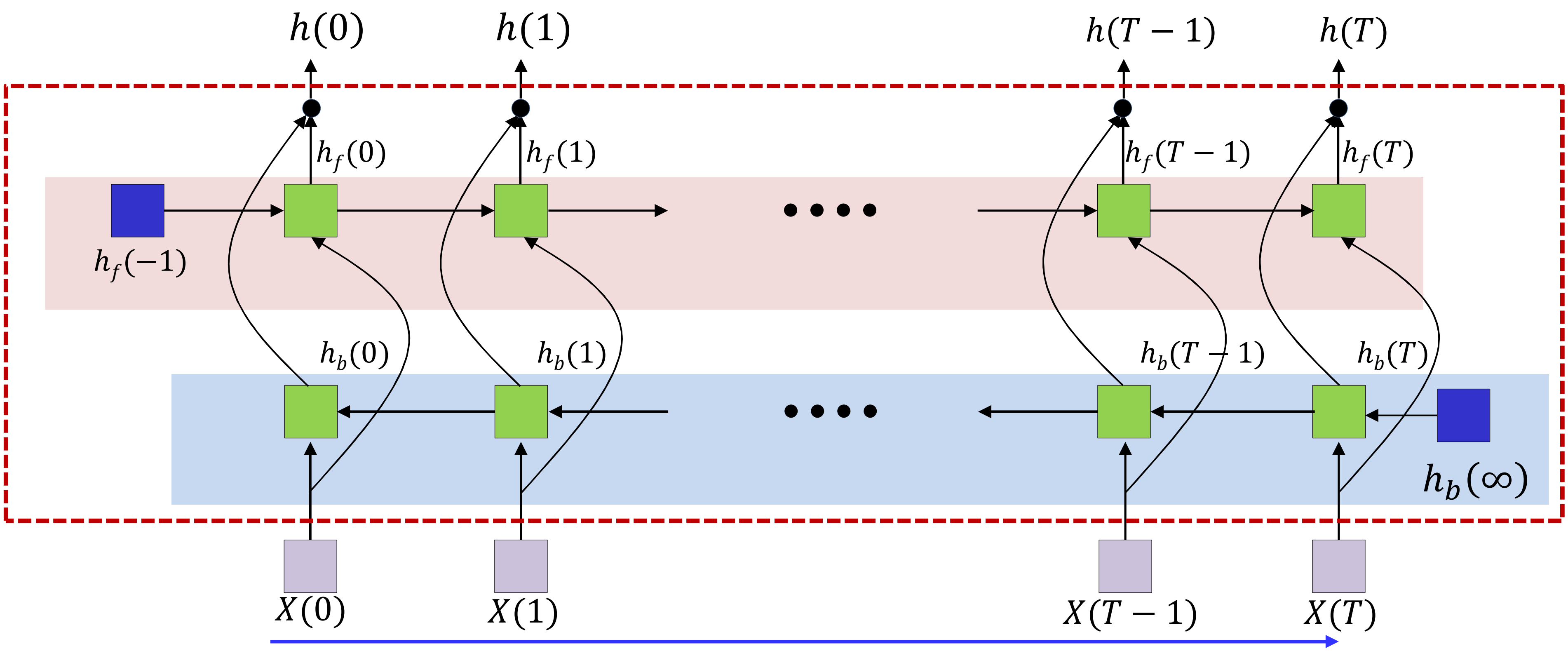
- Network produces one output for each input
  - With one-to-one correspondence
  - E.g. Assigning grammar tags to words
    - May require a bidirectional network to consider both past and future words in the sentence

Time-synchronous networks: Inference



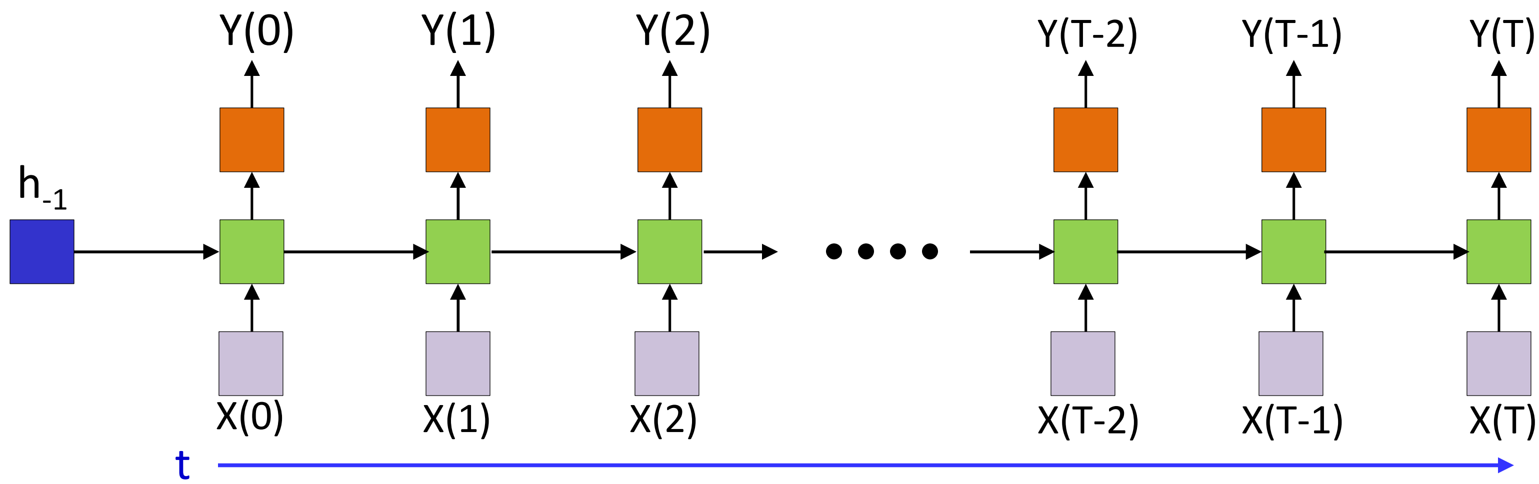
- One sided network: Process input left to right and produce output after each input

Time-synchronous networks: Inference



- **For bidirectional networks:**
  - Process input left to right using forward net
  - Process it right to left using backward net
  - The combined outputs are time-synchronous, one per input time, and are passed up to the next layer
- Rest of the lecture(s) will not specifically consider bidirectional nets, but the discussion generalizes

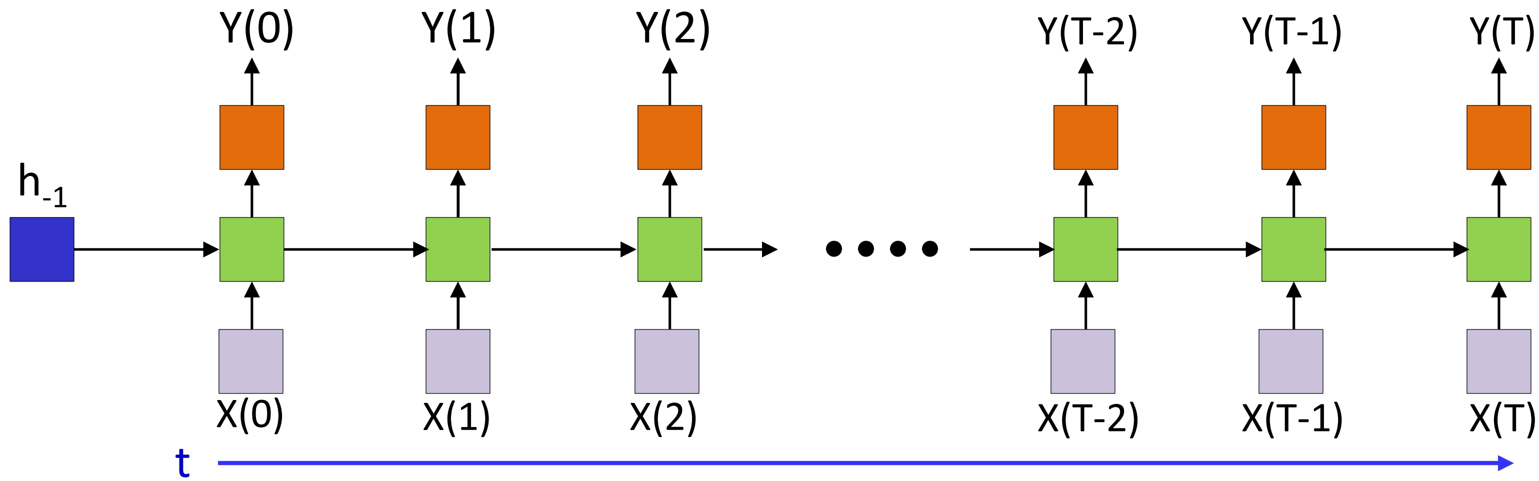
How do we *train* the network



- Back propagation through time (BPTT)

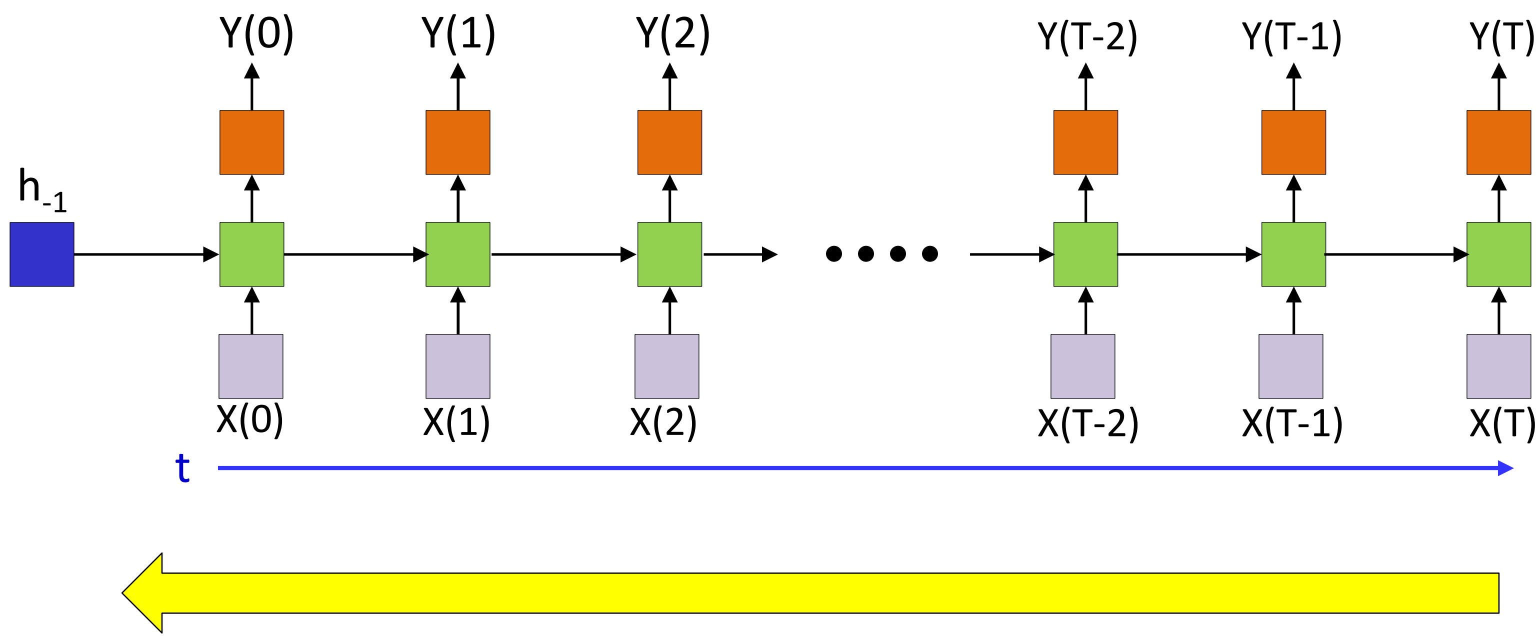
- Given a collection of *sequence* training instances comprising input sequences and output sequences of equal length, with one-to-one correspondence
  - $(\mathbf{X}_i, \mathbf{D}_i)$, where
  - $\mathbf{X}_i = X_{i,0}, \ldots, X_{i,T}$
  - $\mathbf{D}_i = D_{i,0}, \ldots, D_{i,T}$

Training: Forward pass



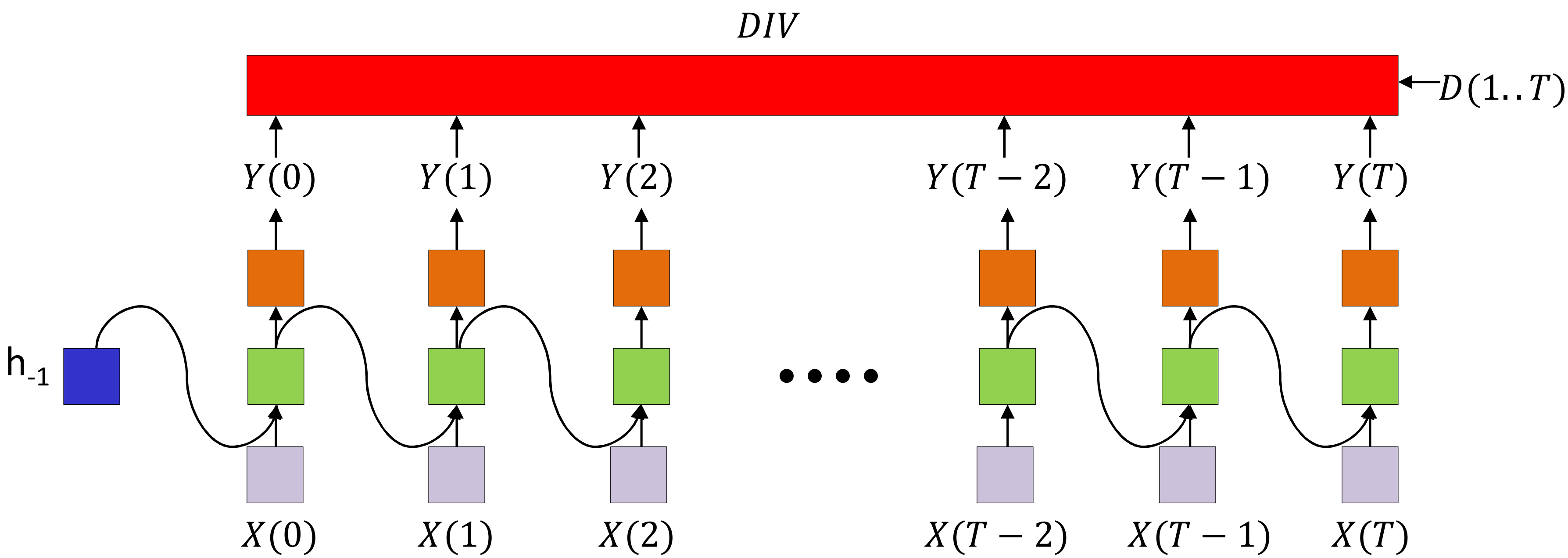
- For each training input:
- Forward pass: pass the entire data sequence through the network, generate outputs

Training: Computing gradients



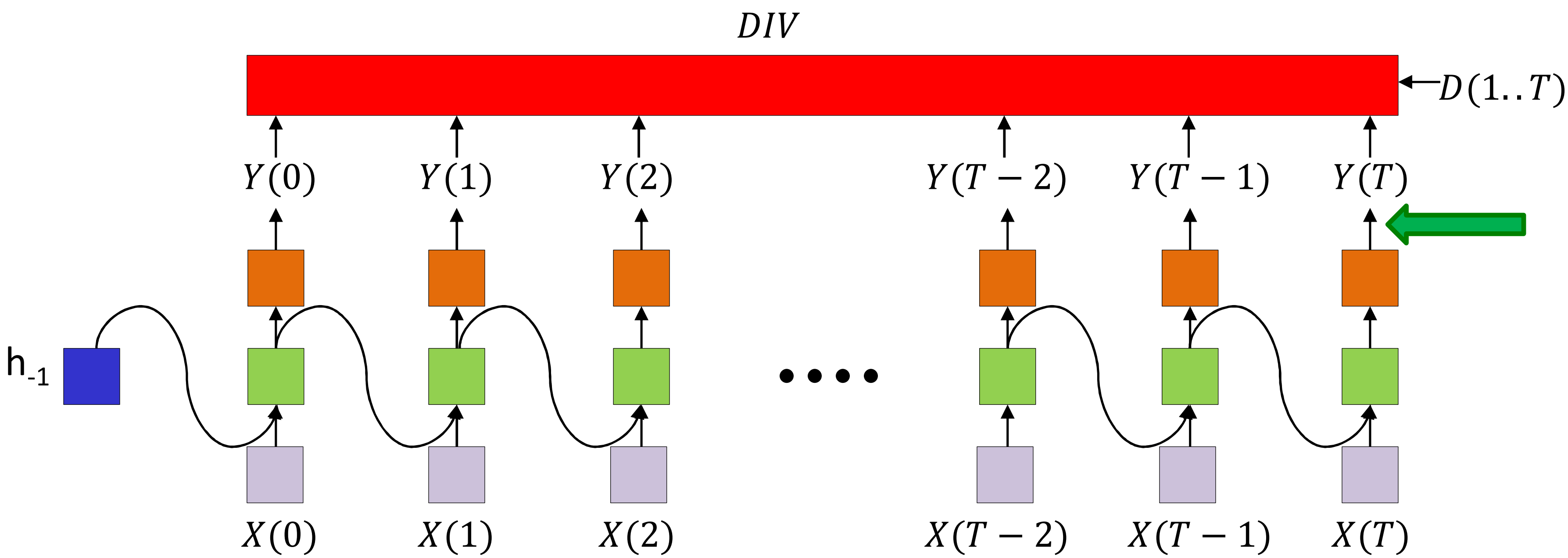
- For each training input:
- Backward pass: Compute divergence gradients via backpropagation
  - *Back Propagation Through Time*

Back Propagation Through Time



- The divergence computed is between the *sequence of outputs* by the network and the *desired sequence of outputs*
- This is *not* just the sum of the divergences at individual times
  - Unless we explicitly define it that way

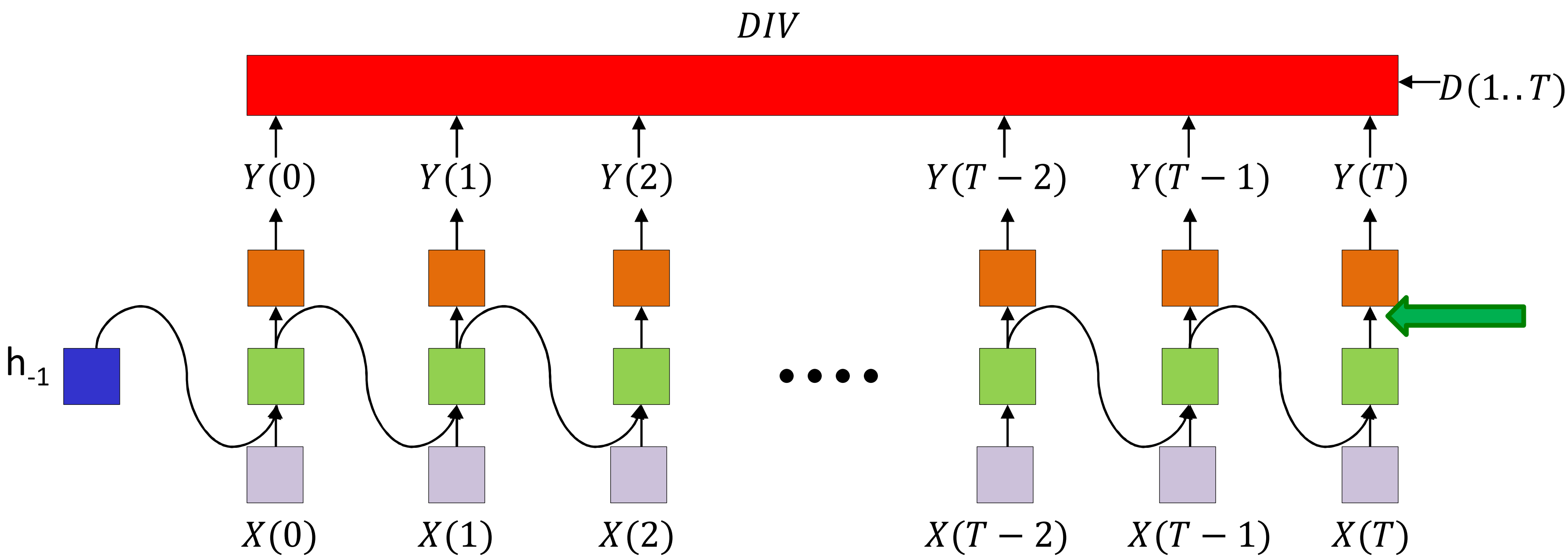
Back Propagation Through Time



First step of backprop: Compute $\nabla_{Y(t)} DIV$ for all t

The rest of backprop continues from there

Back Propagation Through Time

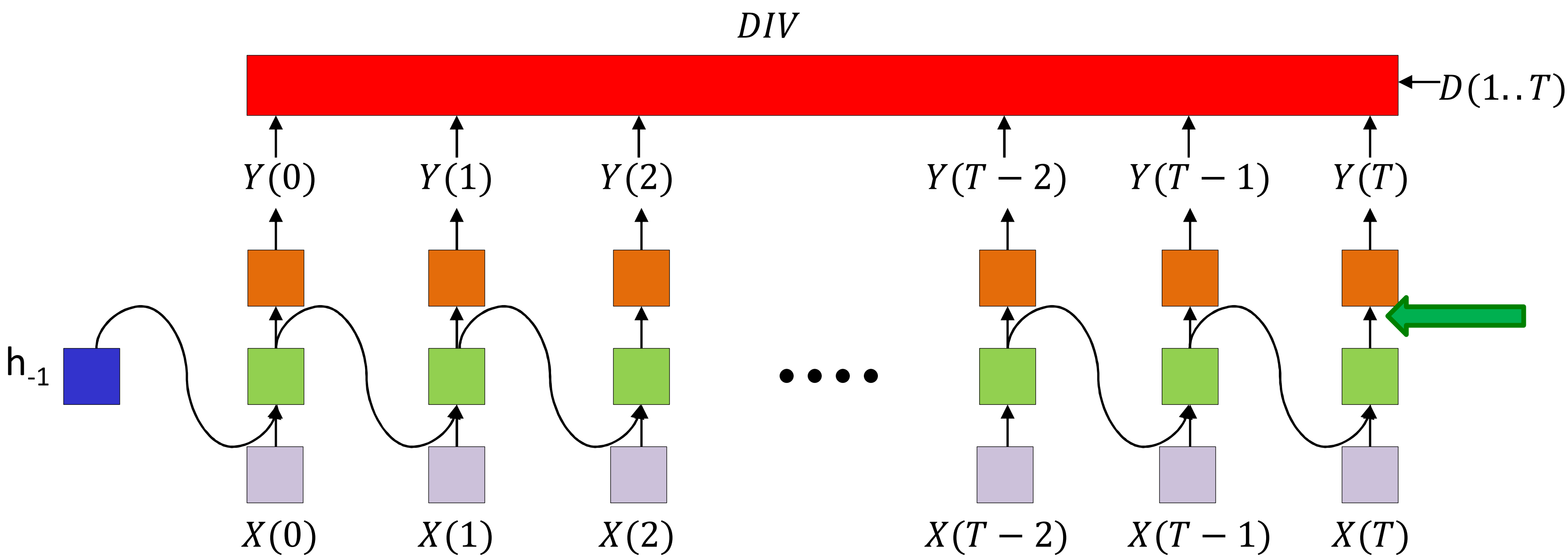


First step of backprop: Compute $\nabla_{Y(t)} DIV$ for all t

$$\boxed{\nabla_{Z^{(1)}(t)} DIV = \nabla_{Y(t)} DIV \; \nabla_{Z(t)} Y(t)}$$

And so on!

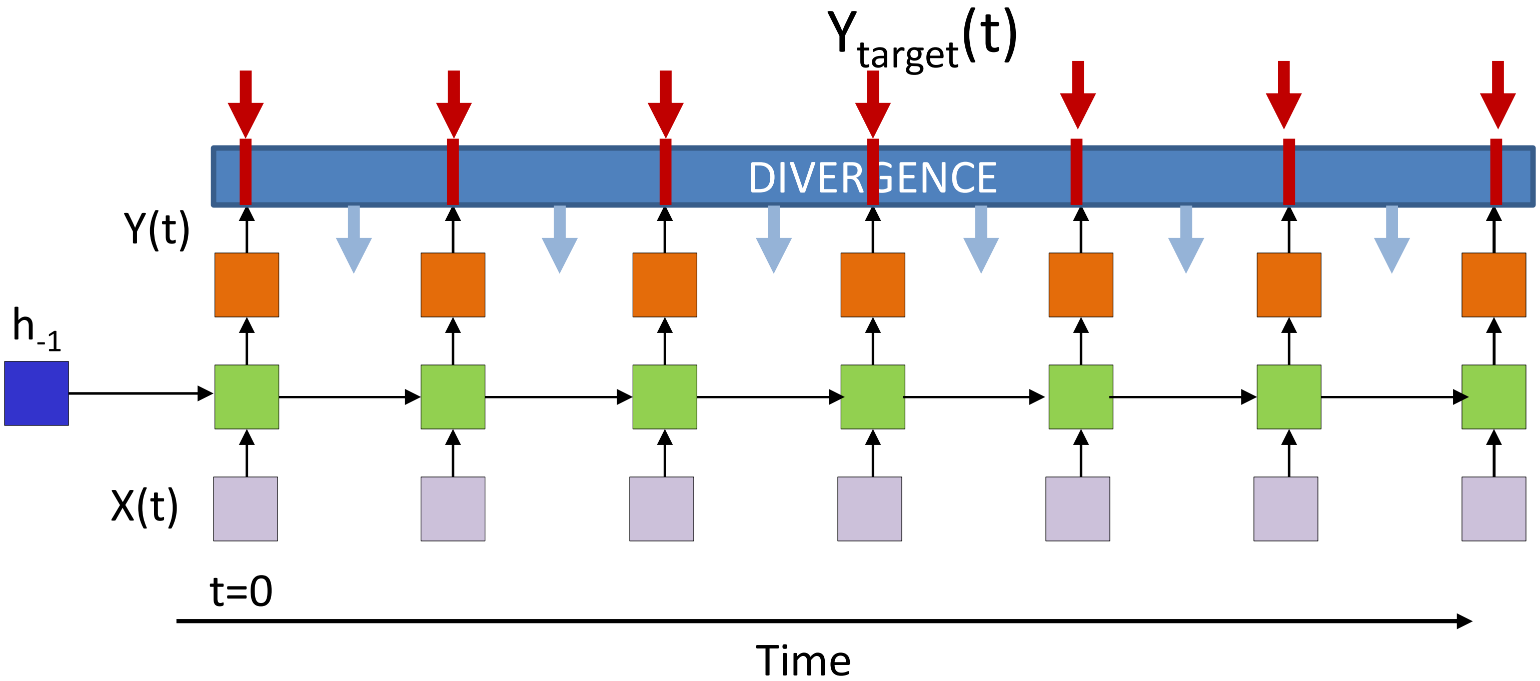
Back Propagation Through Time



First step of backprop: Compute $\nabla_{Y(t)} DIV$ for all t

- The key component is the computation of this derivative!!
- This depends on the definition of "DIV"

Time-synchronous recurrence

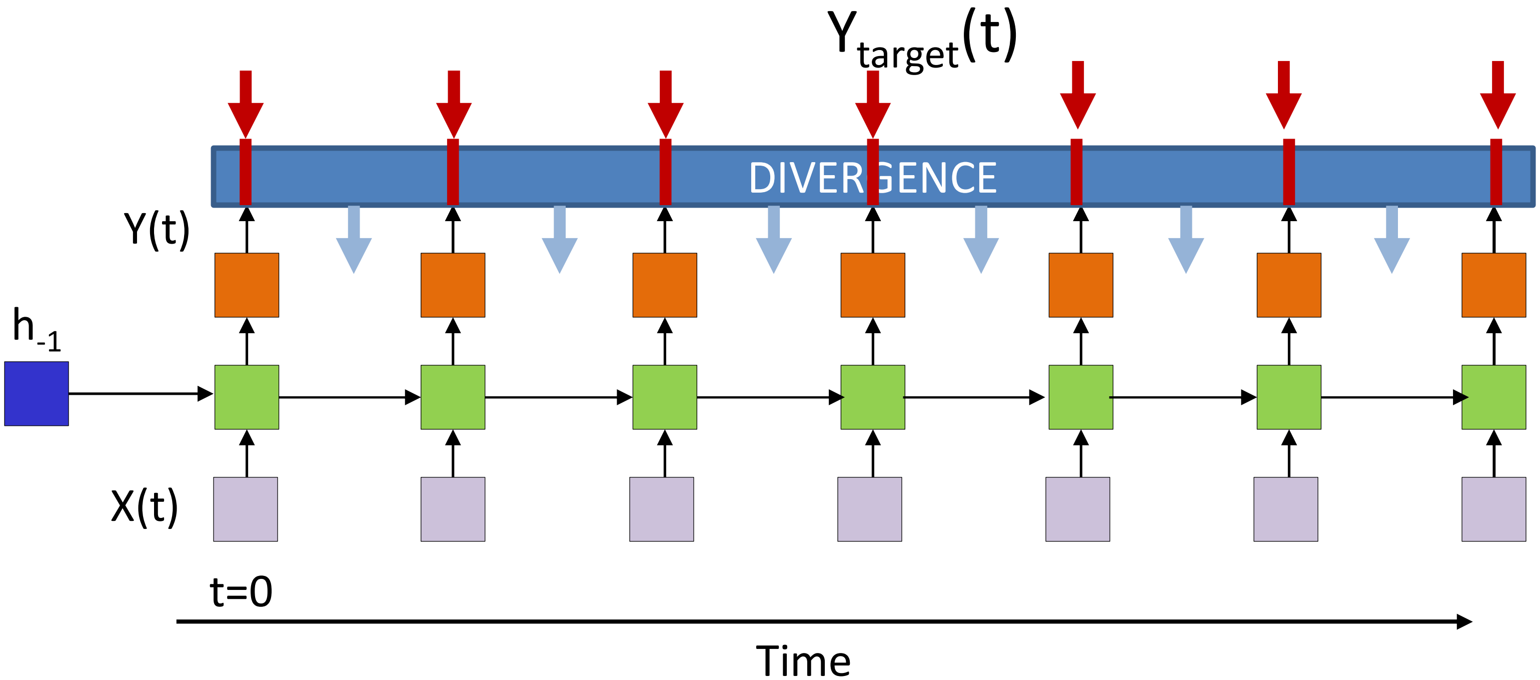


- Usual assumption: ***Sequence divergence is the sum of the divergence at individual instants***

$$Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \sum_t Div\left(Y_{target}(t), Y(t)\right)$$

$$\nabla_{Y(t)} Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \nabla_{Y(t)} Div\left(Y_{target}(t), Y(t)\right)$$

Time-synchronous recurrence



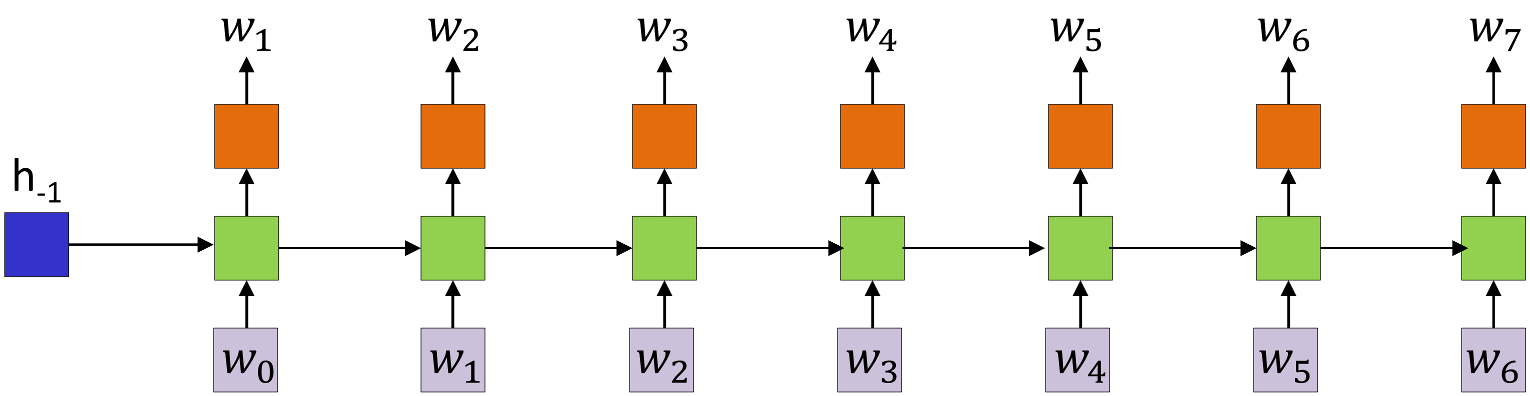
- Usual assumption: ***Sequence divergence is the sum of the divergence at individual instants***

$$Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \sum_t Div\left(Y_{target}(t), Y(t)\right)$$

$$\nabla_{Y(t)} Div\left(Y_{target}(1 \dots T), Y(1 \dots T)\right) = \nabla_{Y(t)} Div\left(Y_{target}(t), Y(t)\right)$$

Typical Divergence for classification: $Div\left(Y_{target}(t), Y(t)\right) = KL(Y_{target}(t), Y(t))$

Simple recurrence example: Text Modelling



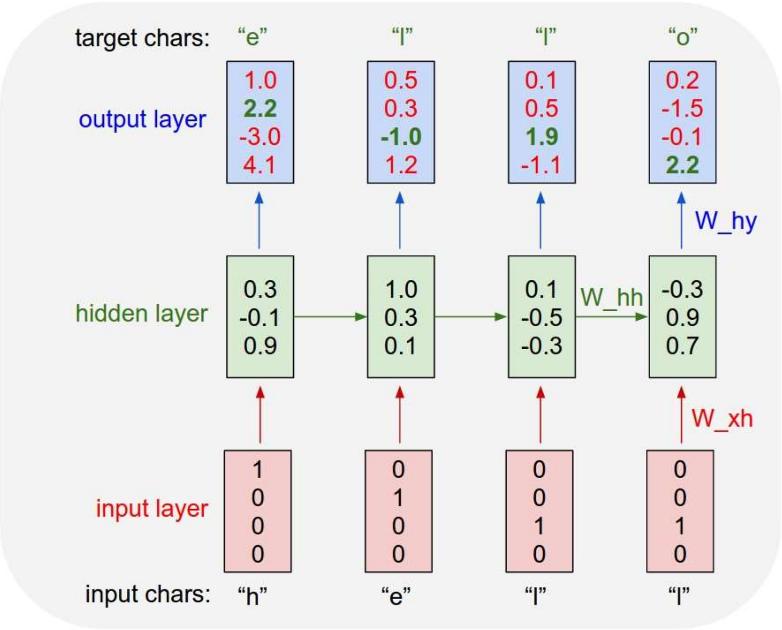
- Learn a model that can predict the next character given a sequence of characters
    - L I N C O L ?
  - Or, at a higher level, words
    - TO BE OR NOT TO ???
- After observing inputs $w_0 \ldots w_k$ it predicts w_{k+1}

Simple recurrence example: Text Modelling

Figure from Andrej Karpathy.

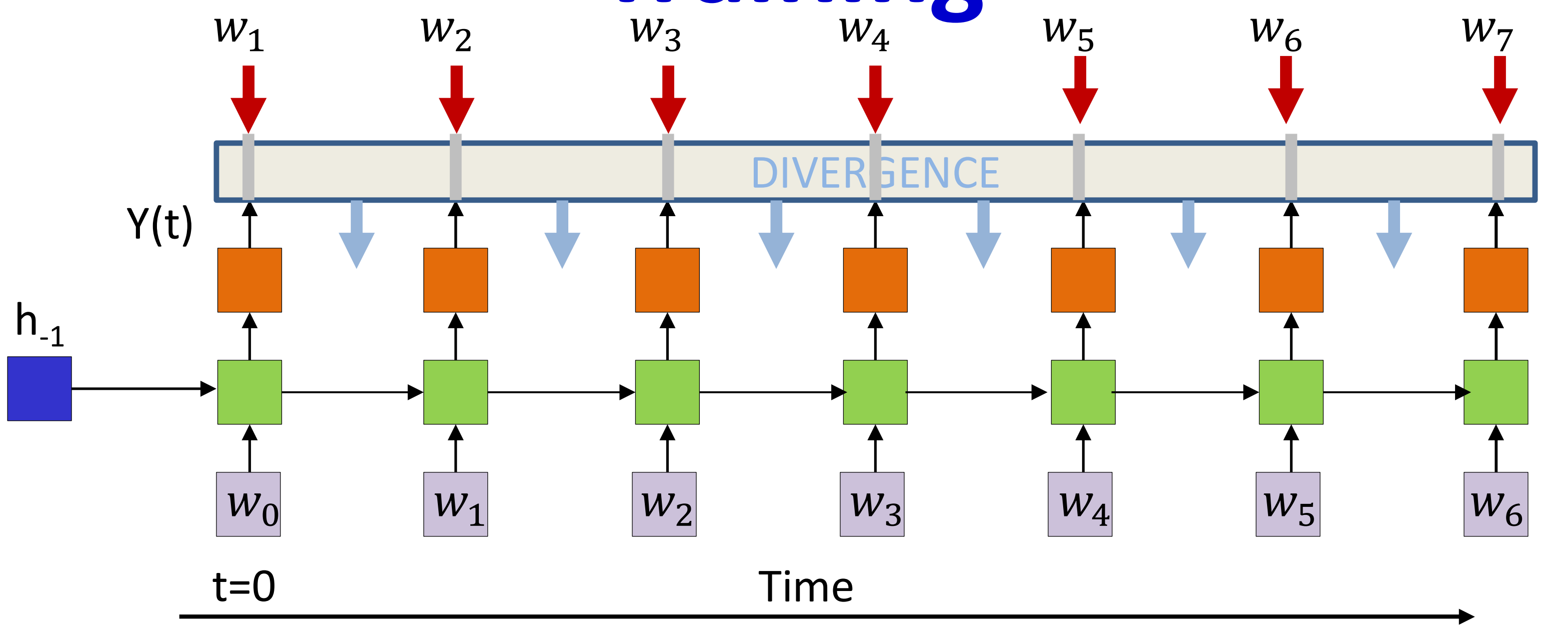
Input: Sequence of characters (presented as one-hot vectors).

Target output after observing “h e l l” is “o”



- Input presented as one-hot vectors
  - Actually “embeddings” of one-hot vectors
- Output: probability distribution over characters
  - Must ideally peak at the target character

Training



- Input: symbols as one-hot vectors
  - Dimensionality of the vector is the size of the "vocabulary"
- Output: Probability distribution over symbols

$$Y(t,i) = P(V_i | w_0 \dots w_{t-1})$$

  - V_i is the i-th symbol in the vocabulary
- Divergence

The probability assigned to the correct next word

$$Div\big(Y_{target}(1 \dots T), Y(1 \dots T)\big) = \sum_t KL\big(Y_{target}(t), Y(t)\big) = -\sum_t \log Y(t, w_{t+1})$$

Brief detour: Language models

- Modelling language using time-synchronous nets
- More generally language models and embeddings..

Language modelling using RNNs

Four score and seven years ???

A B R A H A M L I N C O L ??

- Problem: Given a sequence of words (or characters) predict the next one

Language modelling: Representing words

- Represent words as one-hot vectors
  - Pre-specify a vocabulary of N words in fixed (e.g. lexical) order
    - *E.g. [A AARDVARK AARON ABACK ABACUS… ZZYP]*
  - Represent each word by an N-dimensional vector with N-1 zeros and a single 1 (in the position of the word in the ordered list of words)
    - E.g. “AARDVARK” → [0 1 0 0 0 …]
    - E.g. “AARON” → [0 0 1 0 0 0 …]

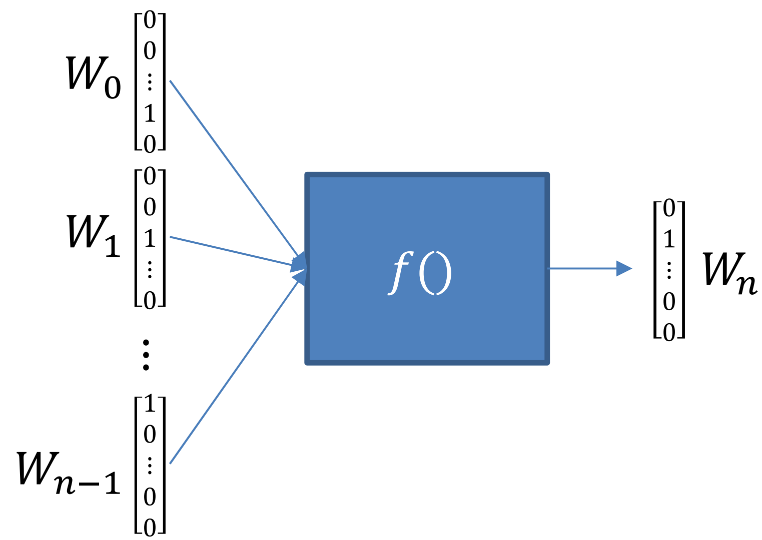
- Characters can be similarly represented
  - English will require about 100 characters, to include both cases, special characters such as commas, hyphens, apostrophes, etc., and the space character

Predicting words

Four score and seven years ???

$$W_n = f(W_0, \ldots, W_{n-1})$$

Nx1 one-hot vectors



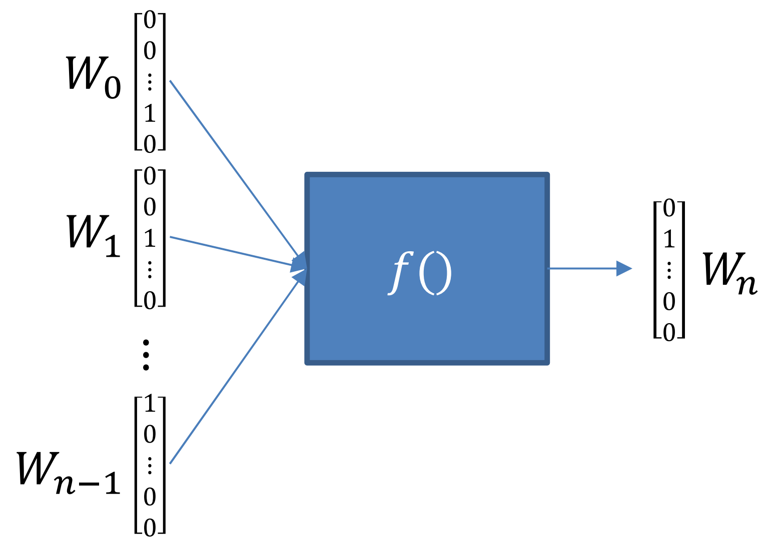
- Given one-hot representations of $W_0 \ldots W_{n-1}$, predict W_n

Predicting words

Four score and seven years ???

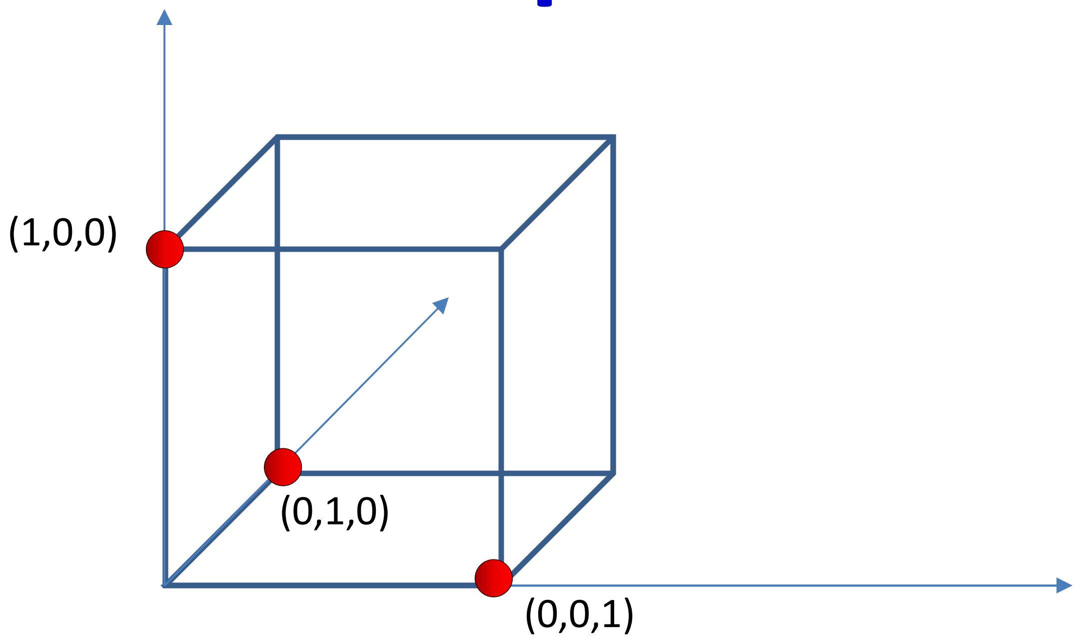
$$W_n = f(W_0, \ldots, W_{n-1})$$

Nx1 one-hot vectors



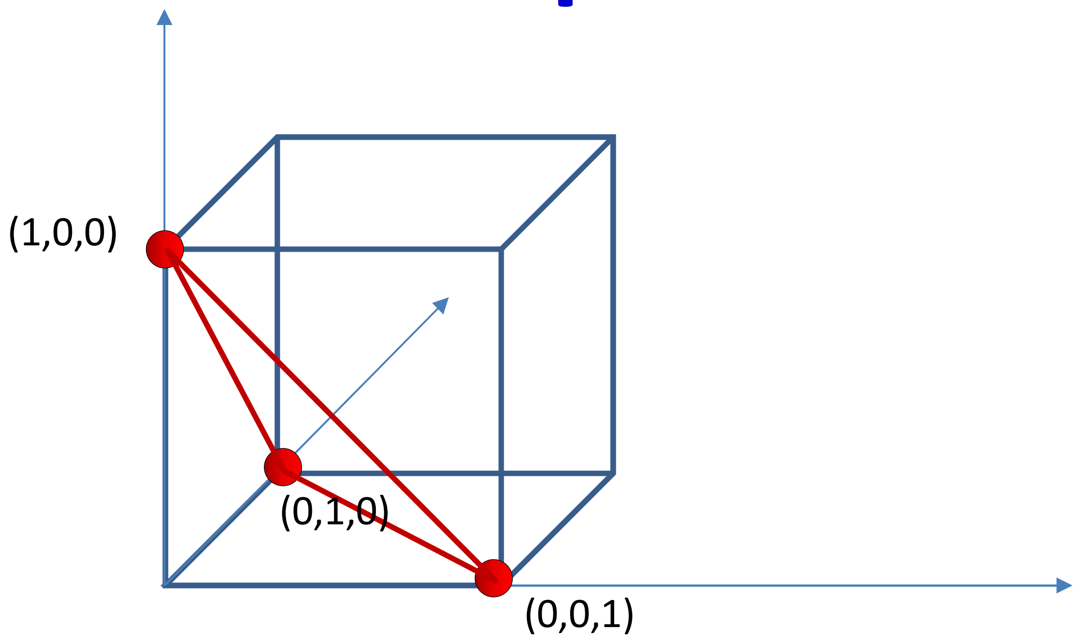
- Given one-hot representations of $W_0 \ldots W_{n-1}$, predict W_n
- **Dimensionality problem:** All inputs $W_0 \ldots W_{n-1}$ are both very high-dimensional and very sparse

The one-hot representation



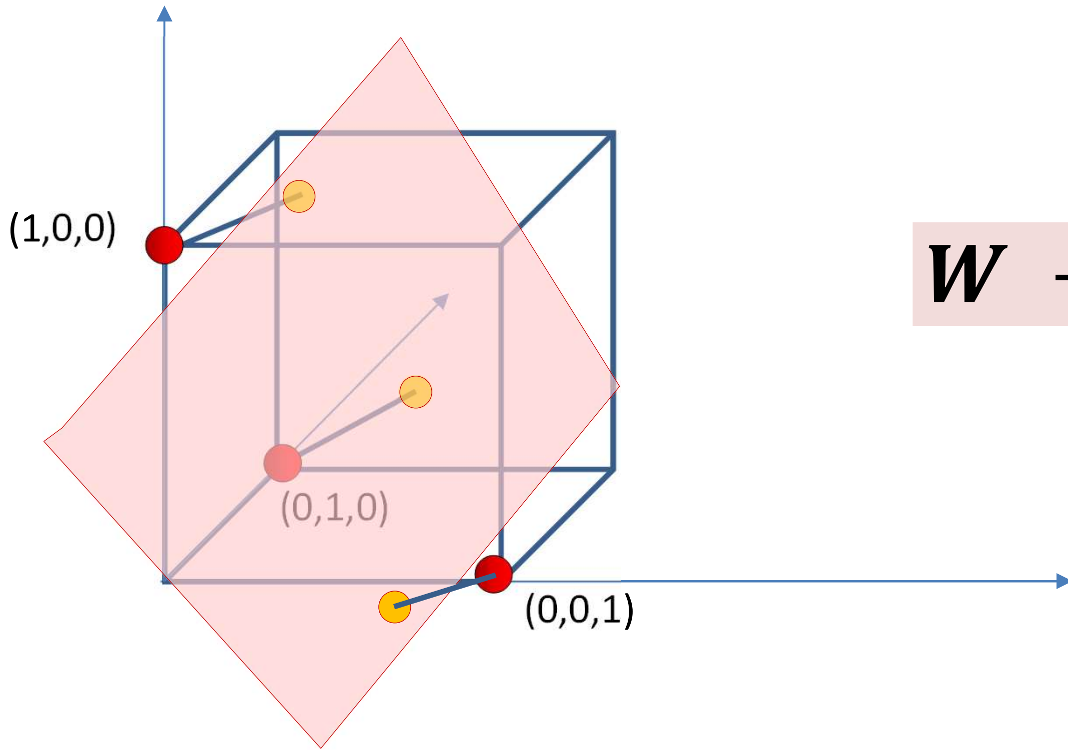
- The one hot representation uses only N corners of the 2^N corners of a unit cube
  - Actual volume of space used = 0
    - $(1, \varepsilon, \delta)$ has no meaning except for $\varepsilon = \delta = 0$
  - Density of points: $\mathcal{O}\left(\frac{N}{r^N}\right)$
- This is a tremendously inefficient use of dimensions

Why one-hot representation



- The one-hot representation makes no assumptions about the relative importance of words
  - All word vectors are the same length
- It makes no assumptions about the relationships between words
  - The distance between every pair of words is the same

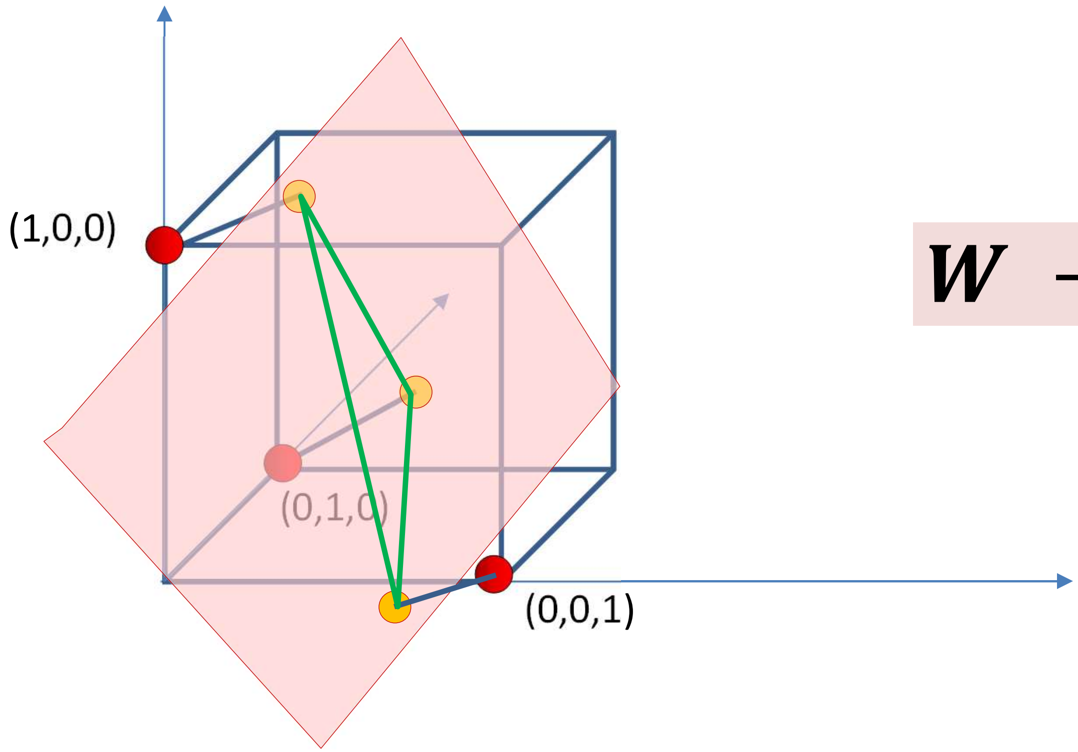
Solution to dimensionality problem



$$\boldsymbol{W} \rightarrow \boldsymbol{PW}$$

- Project the points onto a lower-dimensional subspace
  - Or more generally, a linear transform into a lower-dimensional subspace
  - The volume used is still 0, but density can go up by many orders of magnitude
    - Density of points: $\mathcal{O}\left(\frac{N}{r^M}\right)$

Solution to dimensionality problem



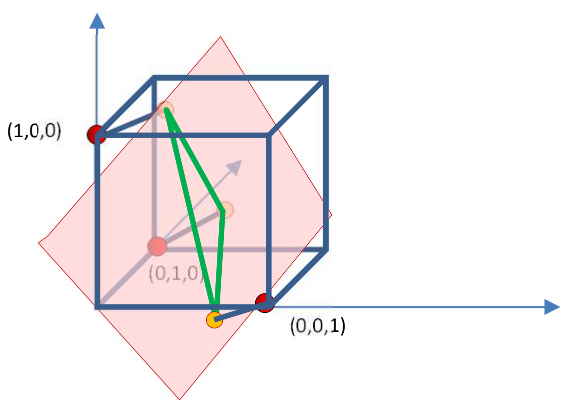
$$\boldsymbol{W} \rightarrow \boldsymbol{PW}$$

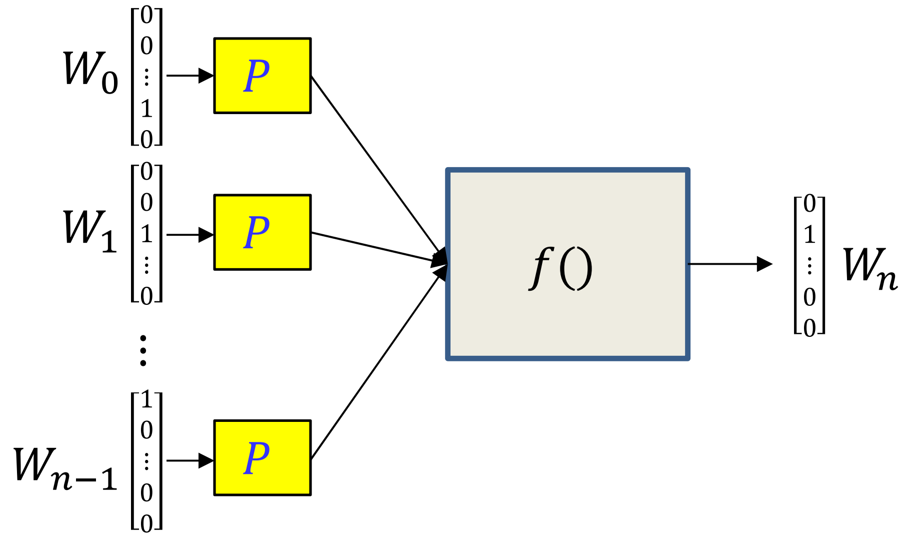
- Project the points onto a lower-dimensional subspace
  - Or more generally, a linear transform into a lower-dimensional subspace
  - The volume used is still 0, but density can go up by many orders of magnitude
    - Density of points: $\mathcal{O}\left(\frac{N}{r^M}\right)$
  - If properly learned, the distances between projected points will capture semantic relations between the words

The *Projected* word vectors

Four score and seven years ???

$$W_n = f(PW_0, PW_2, \ldots, PW_{n-1})$$

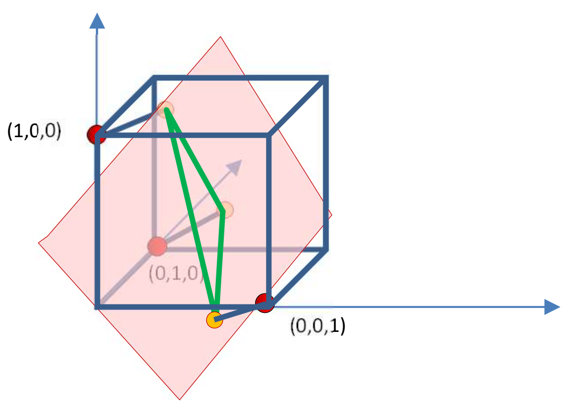


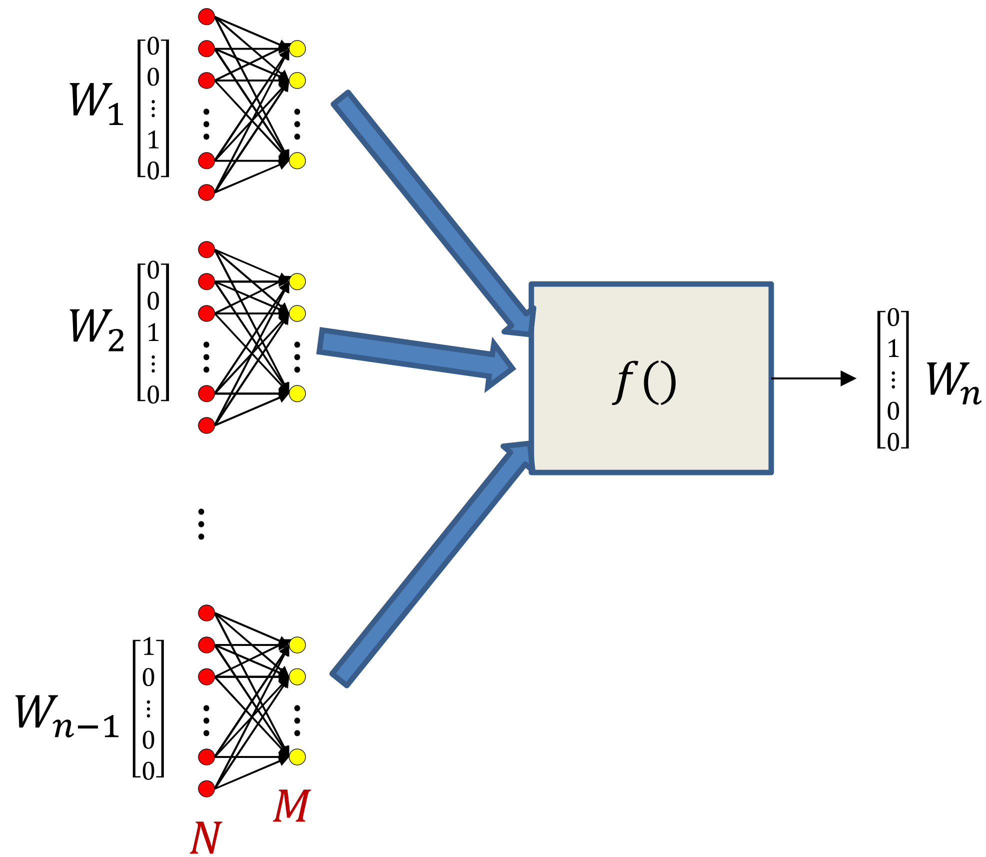


- *Project* the N-dimensional one-hot word vectors into a lower-dimensional space
  - Replace every one-hot vector W_i by PW_i
  - P is an $M \times N$ matrix
  - PW_i is now an M-dimensional vector
  - *Learn P* using an appropriate objective
    - Distances in the projected space will reflect relationships imposed by the objective

"Projection"

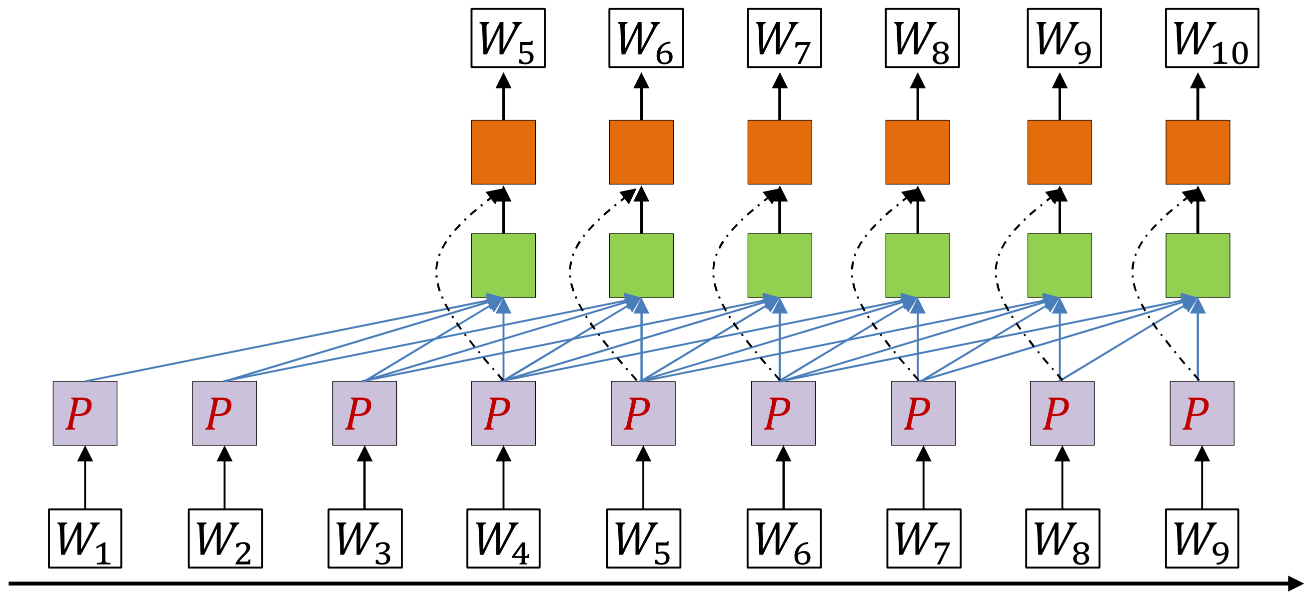
$$W_n = f(PW_1, PW_2, \ldots, PW_{n-1})$$





- P is a simple linear transform
- A single transform can be implemented as a layer of M neurons with linear activation
- The transforms that apply to the individual inputs are all M-neuron linear-activation subnets with tied weights

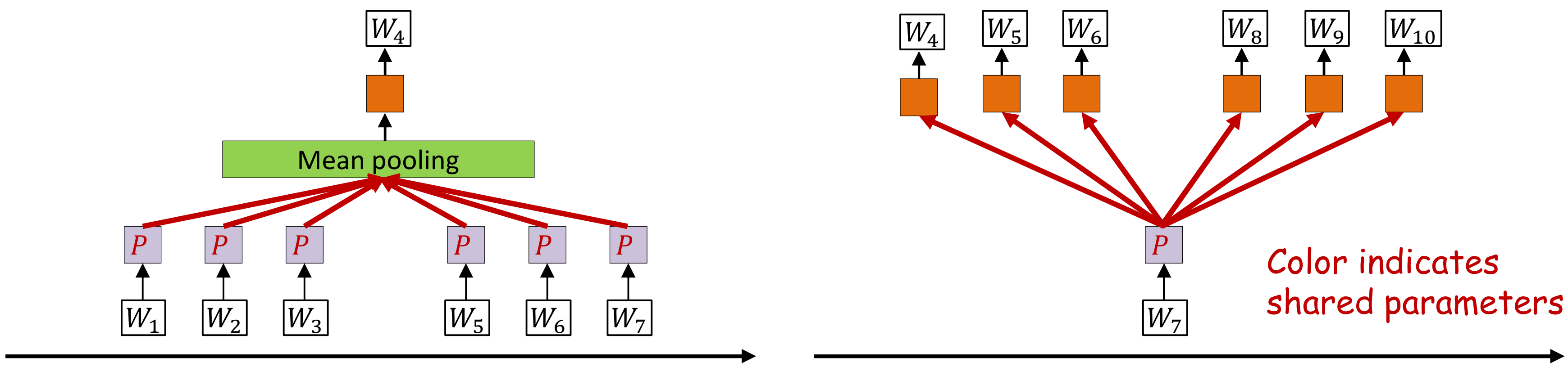
Predicting words: The TDNN model



- Predict each word based on the past N words
  - "A neural probabilistic language model", Bengio et al. 2003
  - Hidden layer has Tanh() activation, output is softmax

- One of the outcomes of learning this model is that we also learn low-dimensional representations PW of words

Alternative models to learn projections



- Soft bag of words: Predict word based on words in immediate context
  - Without considering specific position
- Skip-grams: Predict adjacent words based on current word
- More on these in a future recitation?

Embeddings: Examples

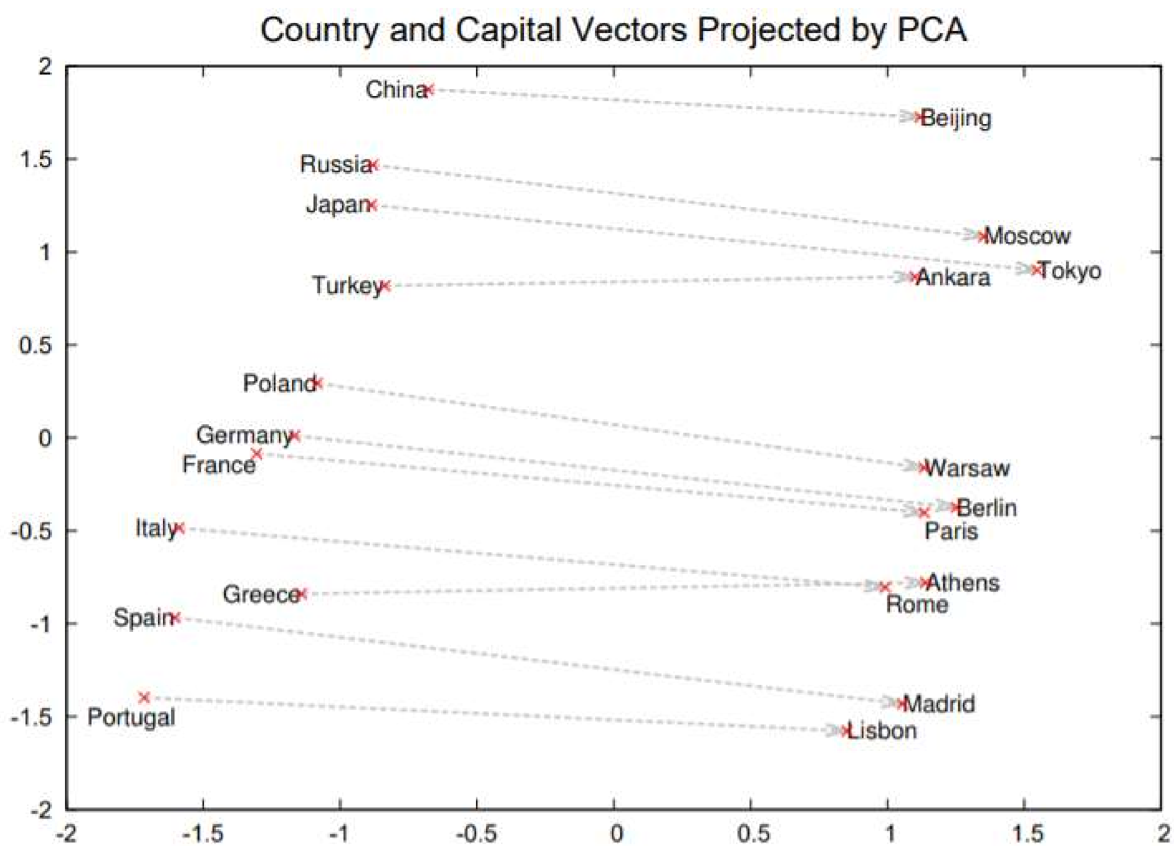
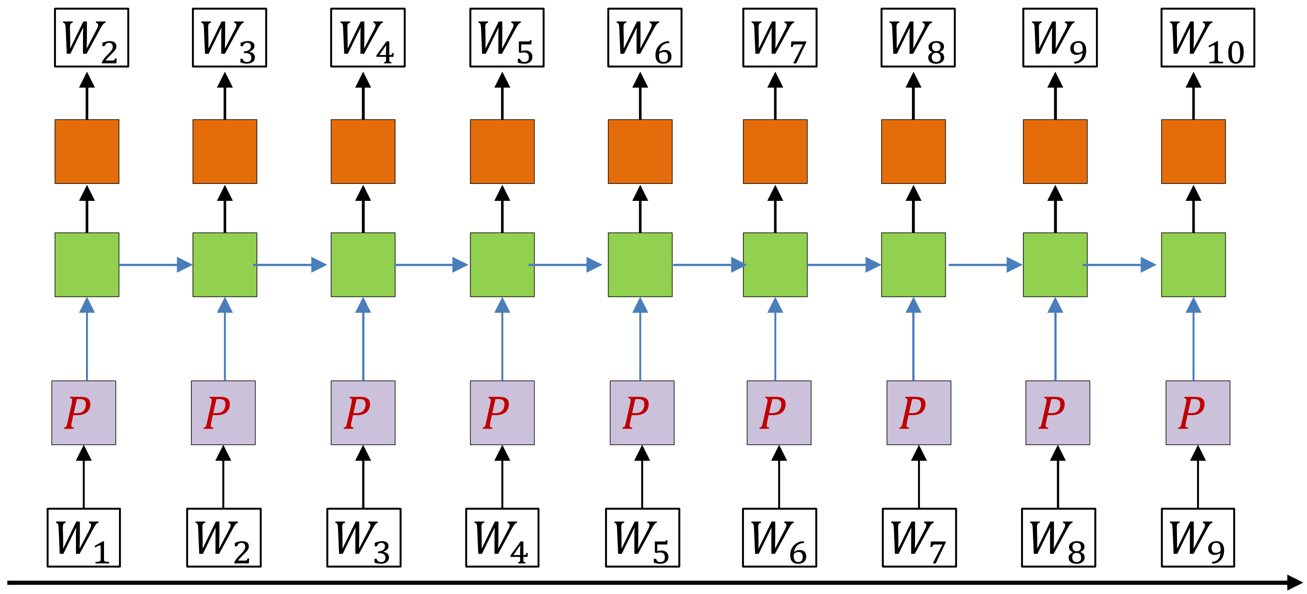


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

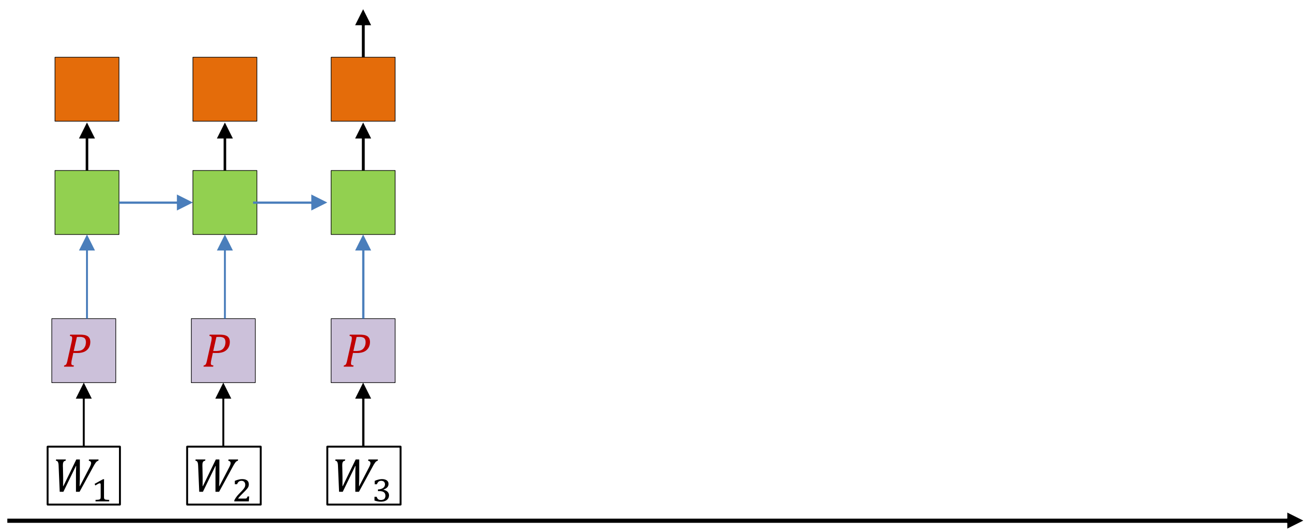
- From Mikolov et al., 2013, "Distributed Representations of Words and Phrases and their Compositionality"

Modelling language



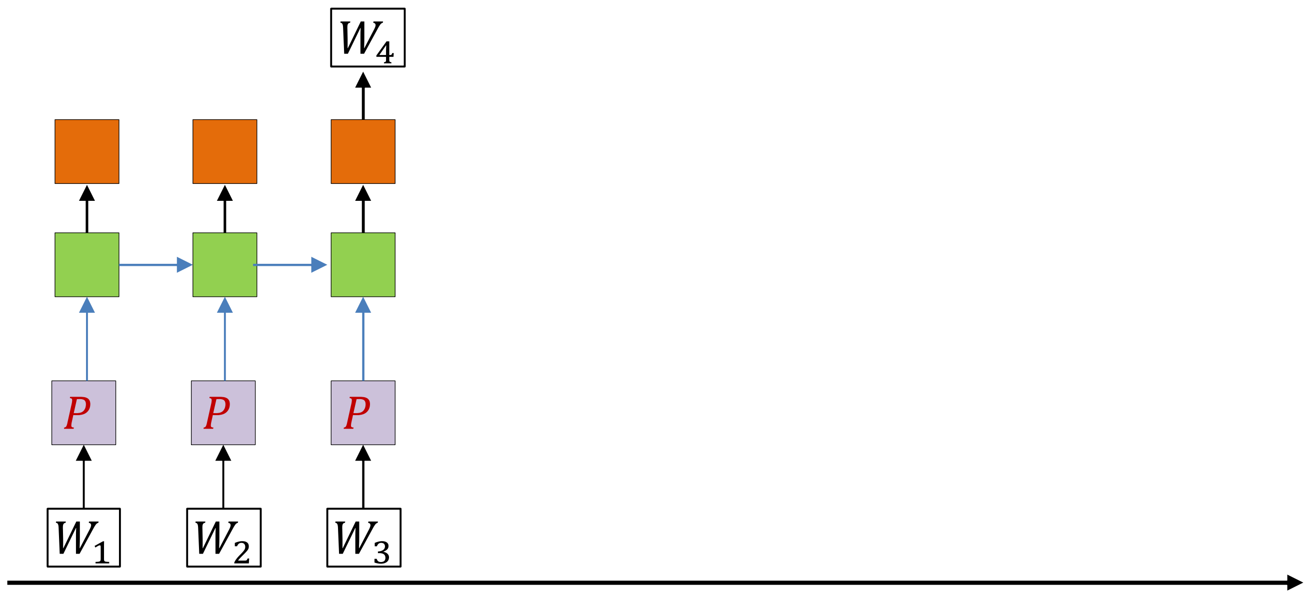
- The hidden units are (one or more layers of) LSTM units
- Trained via backpropagation from a lot of text
  - No explicit labels in the training data: at each time the next word is the label.

Generating Language: Synthesis



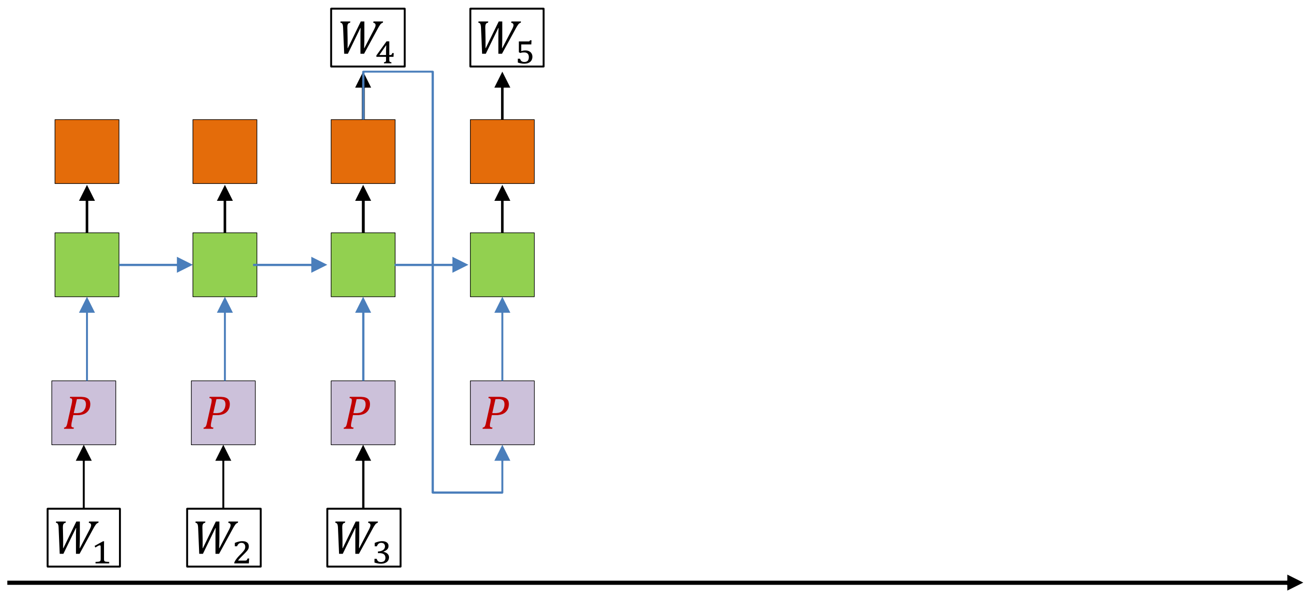
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector

Generating Language: Synthesis



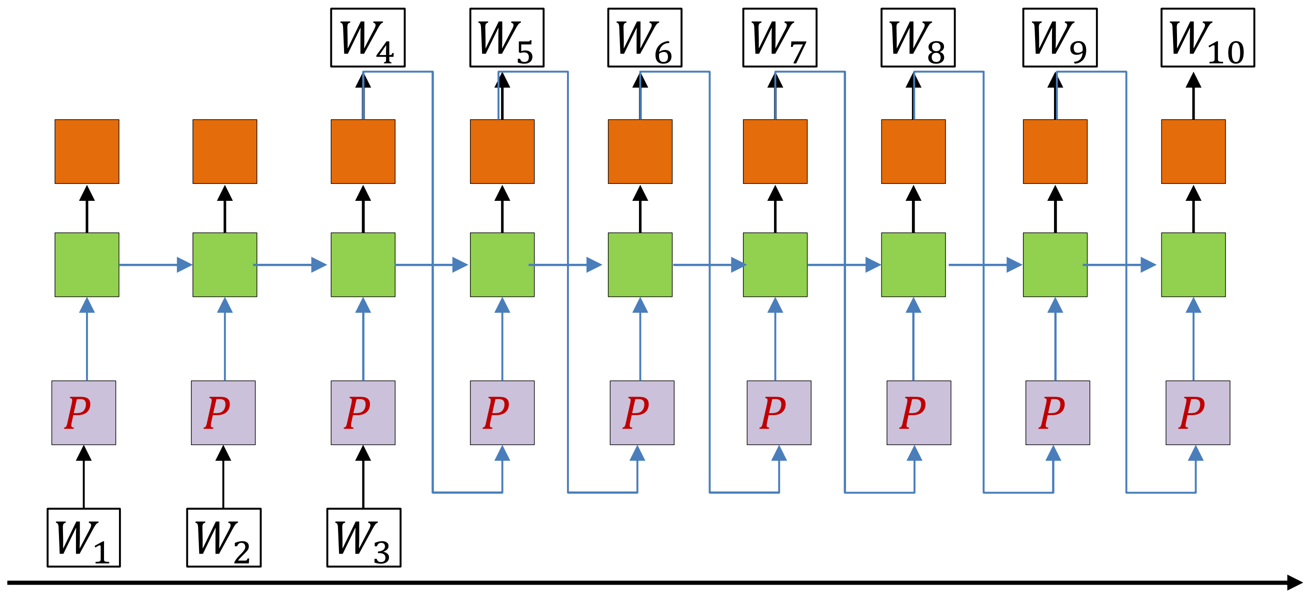
- On trained model : Provide the first few words
  - One-hot vectors
- After the last input word, the network generates a probability distribution over words
  - Outputs an N-valued probability distribution rather than a one-hot vector
- Draw a word from the distribution
  - And set it as the next word in the series

Generating Language: Synthesis



- Feed the drawn word as the next word in the series
  - And draw the next word from the output probability distribution

Generating Language: Synthesis



- Feed the drawn word as the next word in the series
  - And draw the next word from the output probability distribution
- Continue this process until we terminate generation
  - In some cases, e.g. generating programs, there may be a natural termination

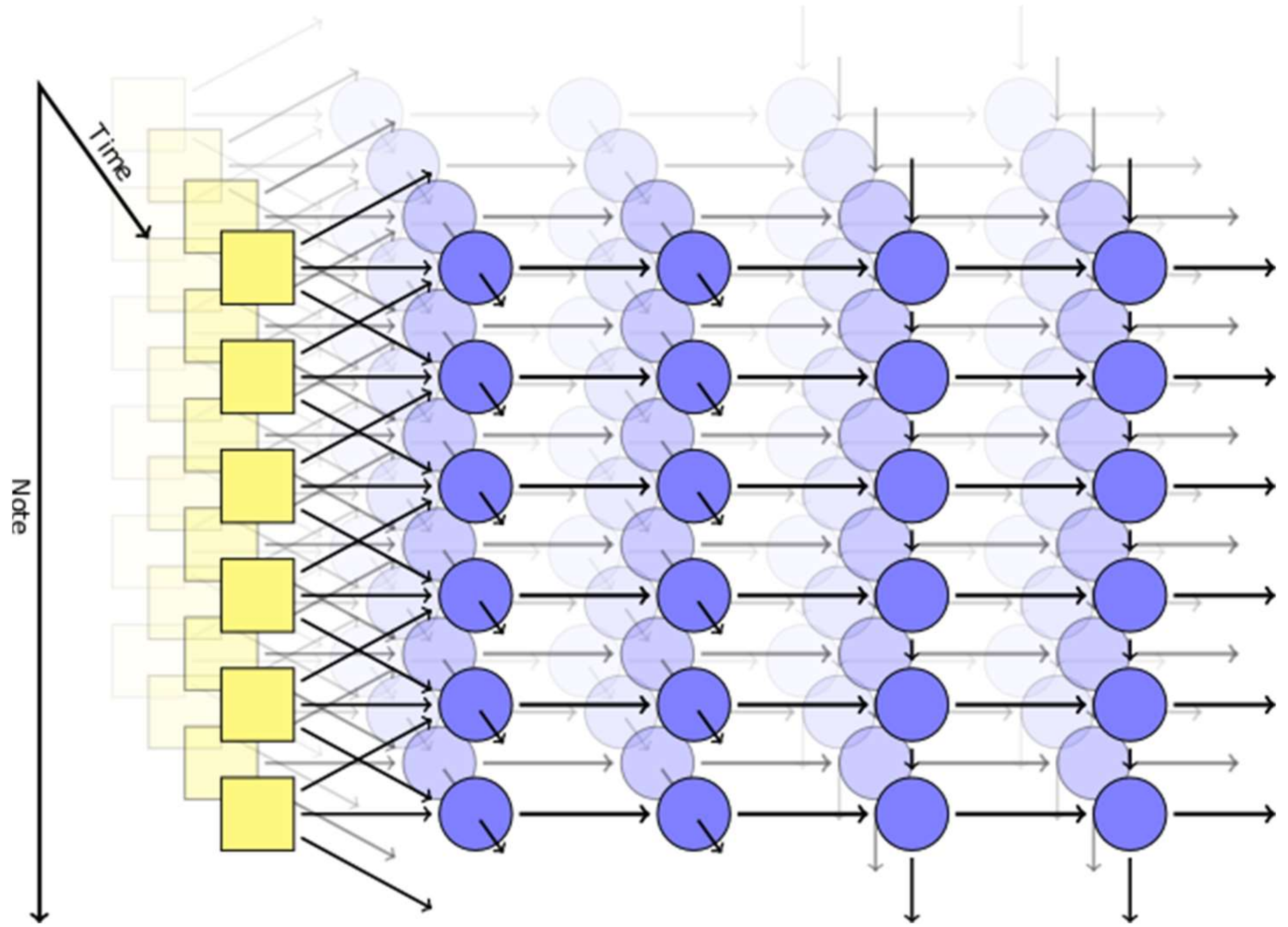
Which open source project?

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECON
  return segtable;
}
```

Trained on linux source code

Actually uses a *character-level* model (predicts character sequences)

Composing music with RNN

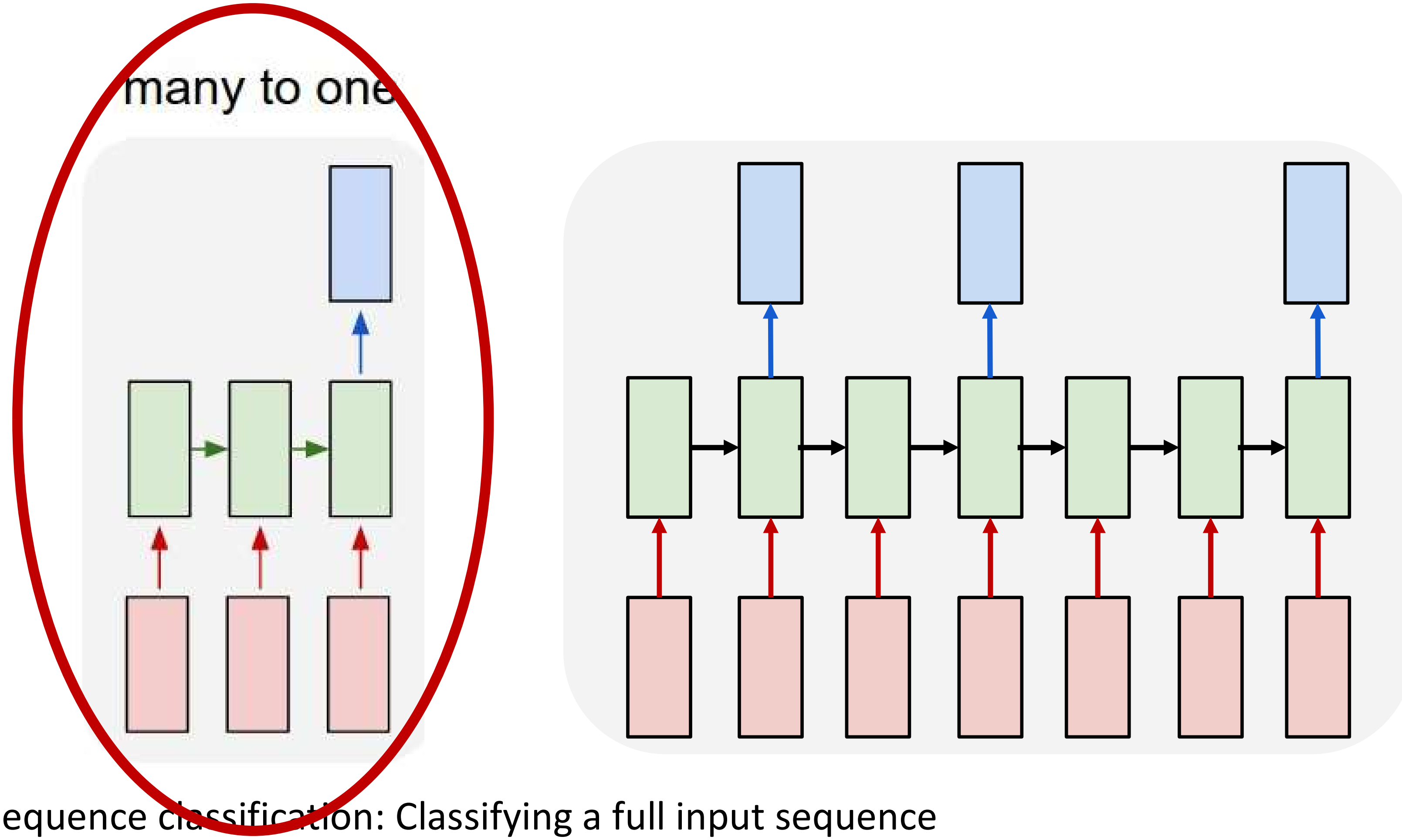


http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/

Returning to our problem

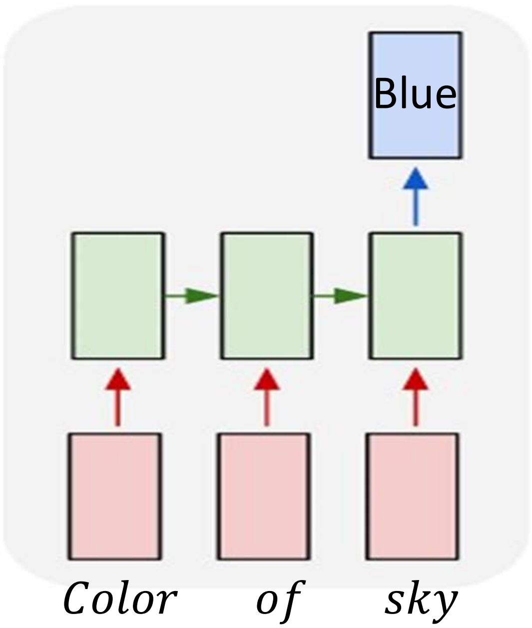
- Divergences are harder to define in other scenarios..

Variants of recurrent nets



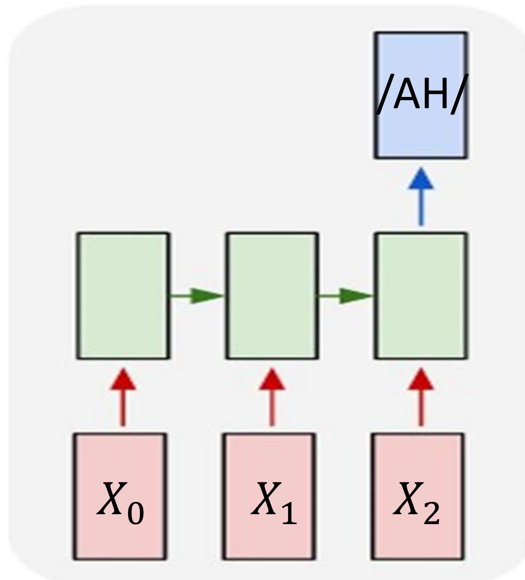
- Sequence classification: Classifying a full input sequence
  - E.g phoneme recognition
- Order synchronous , time asynchronous sequence-to-sequence generation
  - E.g. speech recognition
  - Exact location of output is unknown a priori

Example..



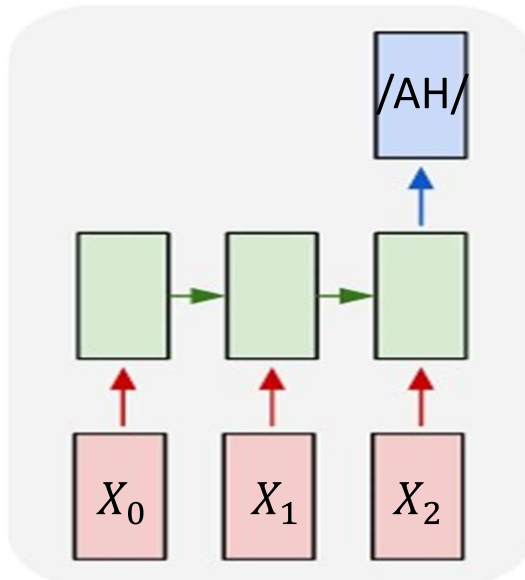
- Question answering
- Input : Sequence of words
- Output: Answer at the end of the question

Example..



- Speech recognition
- Input : Sequence of feature vectors (e.g. Mel spectra)
- Output: Phoneme ID at the end of the sequence
  - Represented as an N-dimensional output probability vector, where N is the number of phonemes

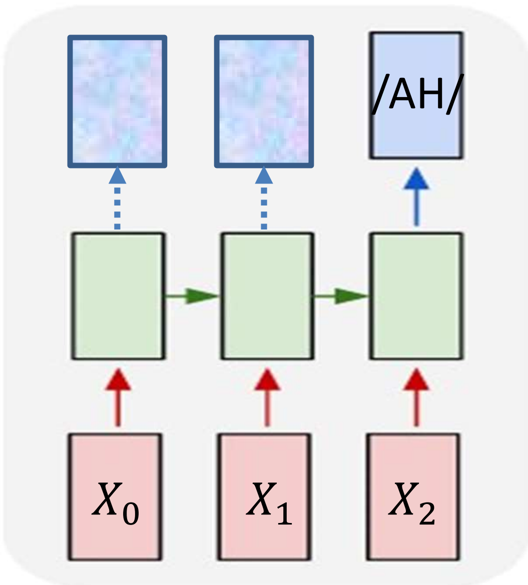
Inference: Forward pass



- Exact input sequence provided
  - Output generated when the last vector is processed
    - Output is a probability distribution over phonemes

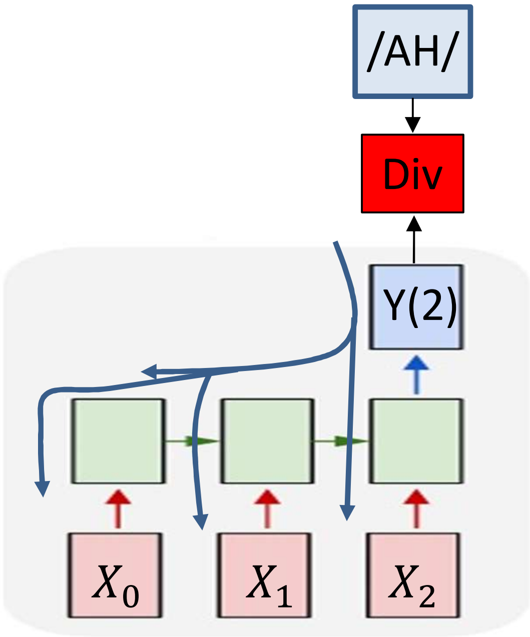
- But what about at *intermediate stages?*

Forward pass



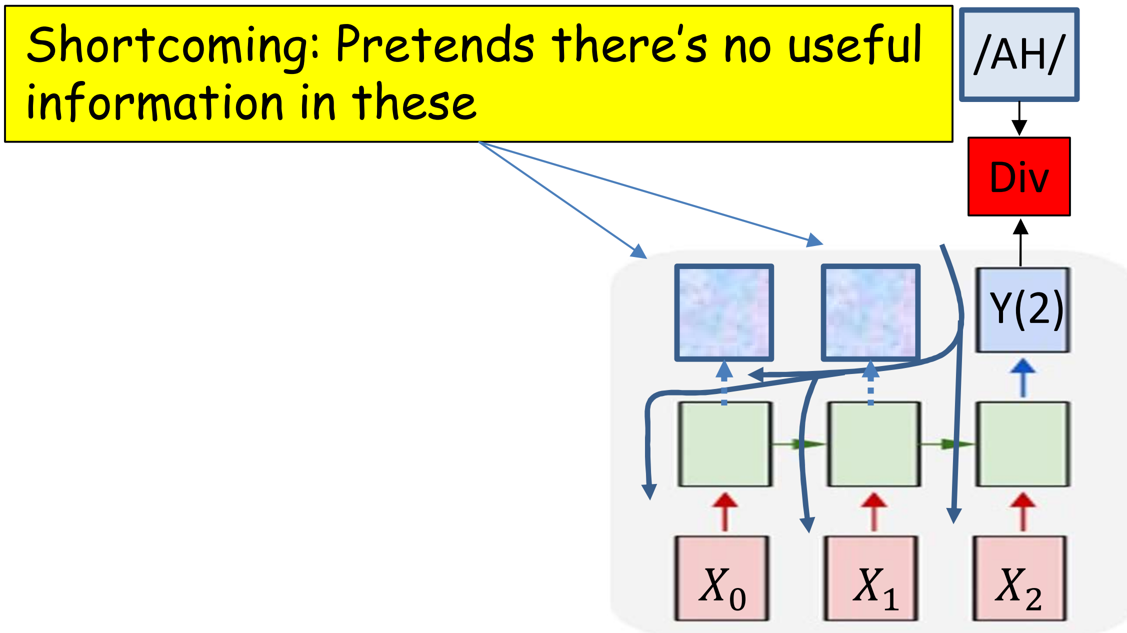
- Exact input sequence provided
  - Output generated when the last vector is processed
    - Output is a probability distribution over phonemes
- Outputs are actually produced for *every* input
  - We only *read* it at the end of the sequence

Training



- The Divergence is only defined at the final input
  - $DIV(Y_{target}, Y) = KL(Y(T), Phoneme)$
- This divergence must propagate through the net to update all parameters

Training

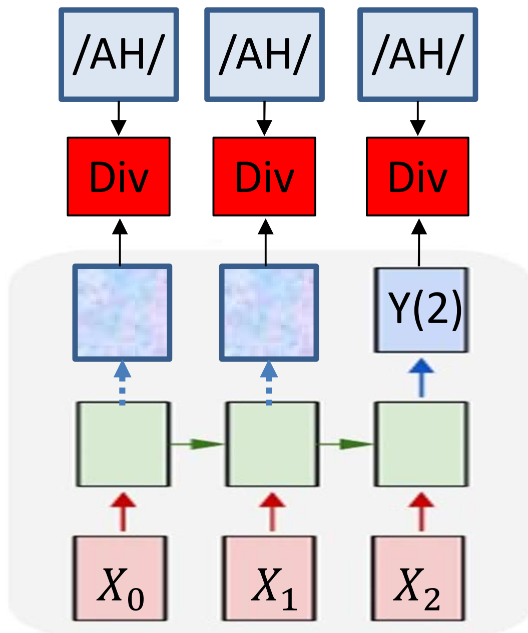


- The Divergence is only defined at the final input
  - $DIV(Y_{target}, Y) = Xent(Y(T), Phoneme)$
- This divergence must propagate through the net to update all parameters

Training

Fix: Use these outputs too.

These too must ideally point to the correct phoneme



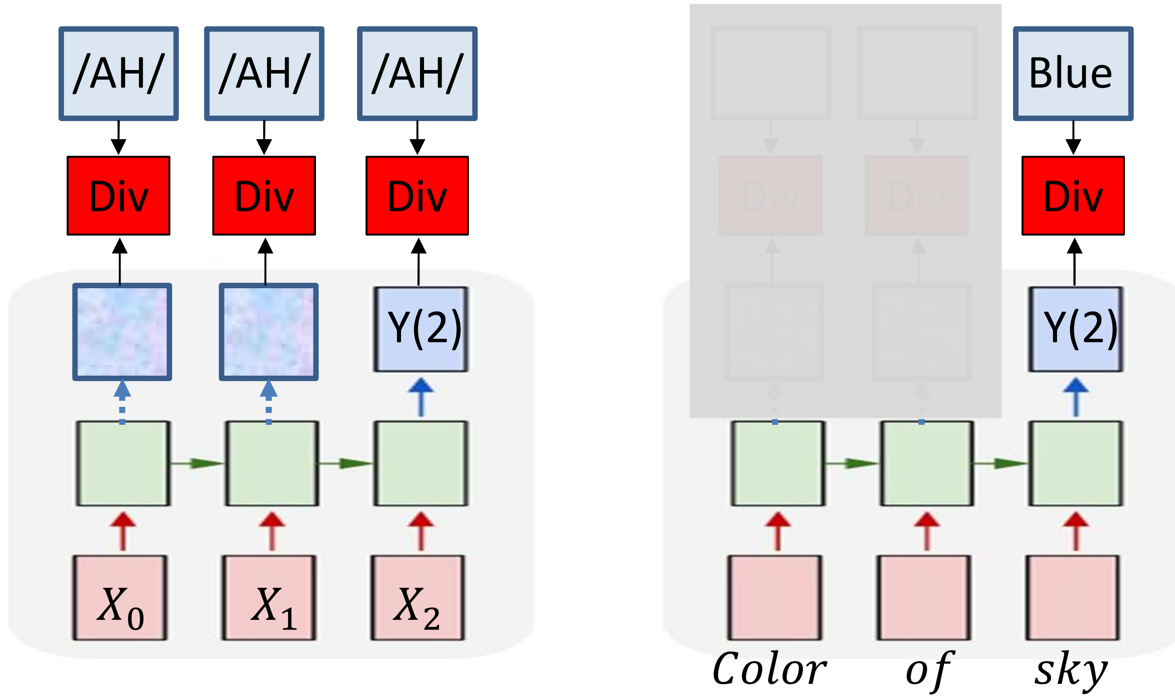
- Exploiting the untagged inputs: assume the same output for the entire input
- Define the divergence everywhere

$$DIV\left(Y_{target}, Y\right) = \sum_t w_t Xent(Y(t), Phoneme)$$

Training

Fix: Use these outputs too.

These too must ideally point to the correct phoneme

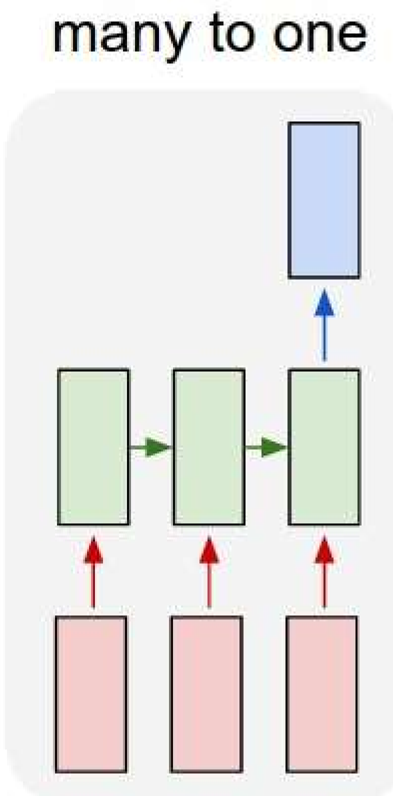


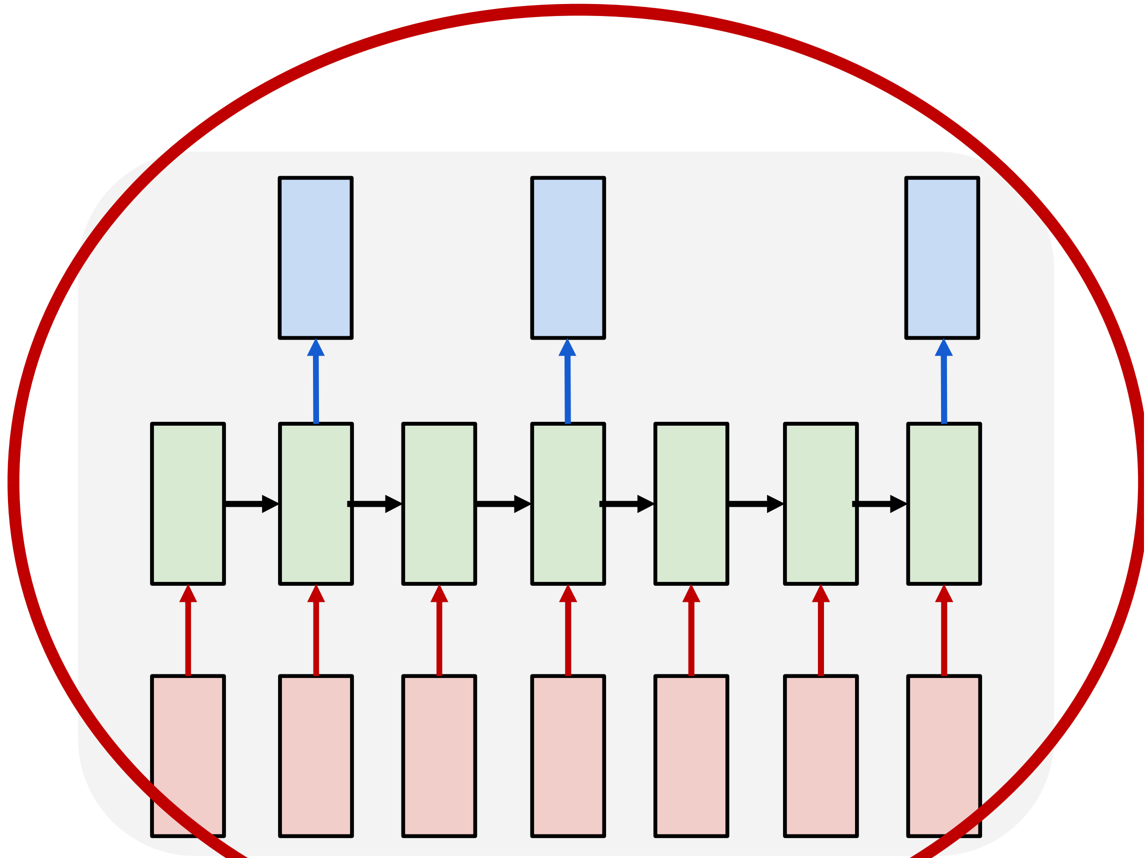
- Define the divergence everywhere

$$DIV\left(Y_{target}, Y\right) = \sum_t w_t Xent(Y(t), Phoneme)$$

- Typical weighting scheme for speech: all are equally important
- Problem like question answering: answer only expected after the question ends
  - Only w_T is high, other weights are 0 or low

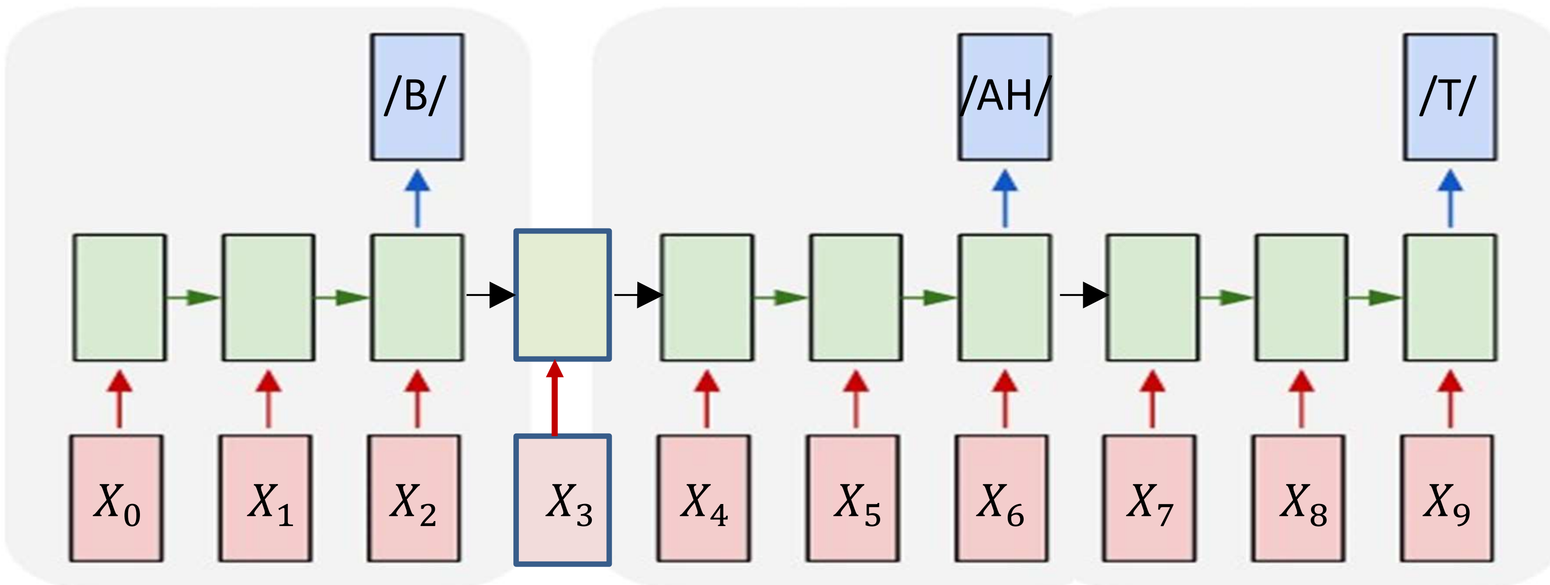
Variants on recurrent nets





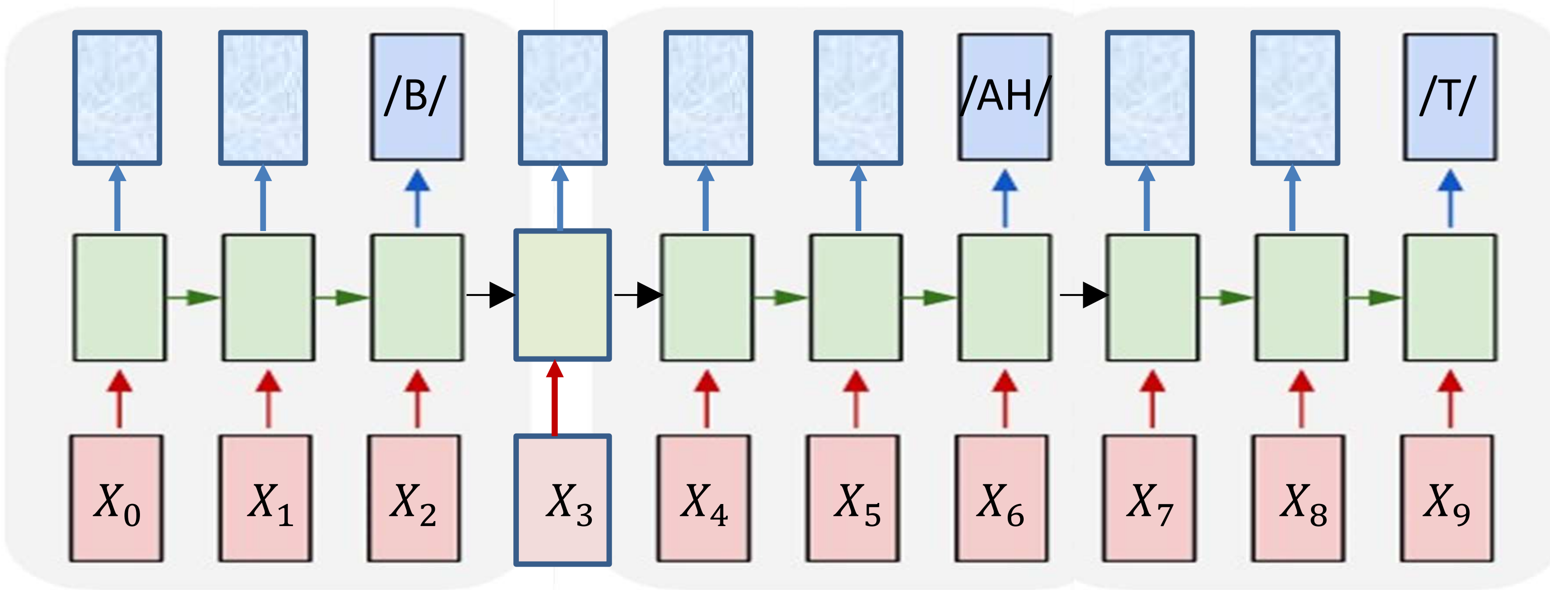
- Sequence classification: Classifying a full input sequence
  - E.g phoneme recognition
- Order synchronous , time asynchronous sequence-to-sequence generation
  - E.g. speech recognition
  - Exact location of output is unknown a priori

A more complex problem



- Objective: Given a sequence of inputs, asynchronously output a sequence of symbols
  - This is just a simple concatenation of many copies of the simple "output at the end of the input sequence" model we just saw
- But this simple extension complicates matters..

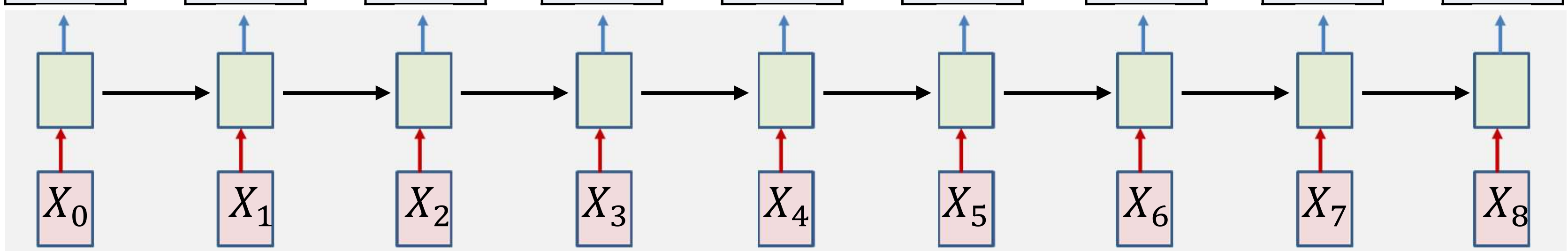
The *sequence-to-sequence* problem



- How do we know *when* to output symbols
  - In fact, the network produces outputs at *every* time
  - *Which* of these are the *real* outputs
    - Outputs that represent the definitive occurrence of a symbol

The actual output of the network

/AH/	y_0^{AH}	y_1^{AH}	y_2^{AH}	y_3^{AH}	y_4^{AH}	y_5^{AH}	y_6^{AH}	y_7^{AH}	y_8^{AH}
/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}
/D/	y_0^{D}	y_1^{D}	y_2^{D}	y_3^{D}	y_4^{D}	y_5^{D}	y_6^{D}	y_7^{D}	y_8^{D}
/EH/	y_0^{EH}	y_1^{EH}	y_2^{EH}	y_3^{EH}	y_4^{EH}	y_5^{EH}	y_6^{EH}	y_7^{EH}	y_8^{EH}
/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}
/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}
/G/	y_0^{G}	y_1^{G}	y_2^{G}	y_3^{G}	y_4^{G}	y_5^{G}	y_6^{G}	y_7^{G}	y_8^{G}



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

Recap: The output of a network

- Any neural network with a softmax (or logistic) output is actually outputting an estimate of the *a posteriori* probability of the classes given the output

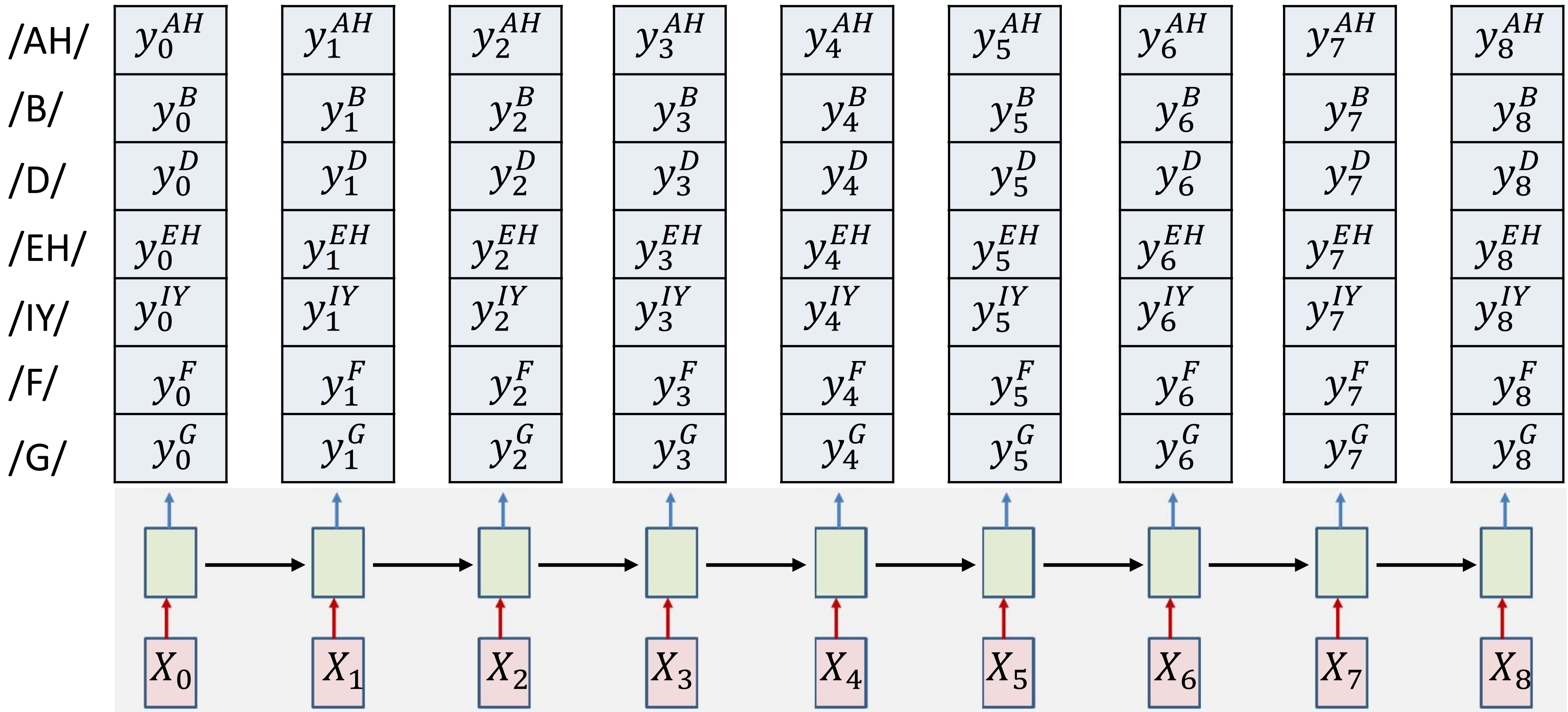
$$[P(c_1|X), P(c_2|X), \ldots, P(c_K|X)]$$

- Selecting the class with the highest probability results in *maximum a posteriori probability* classification

$$Class = \underset{i}{\operatorname{argmax}} P(Y_i|X)$$

- We use the same principle here

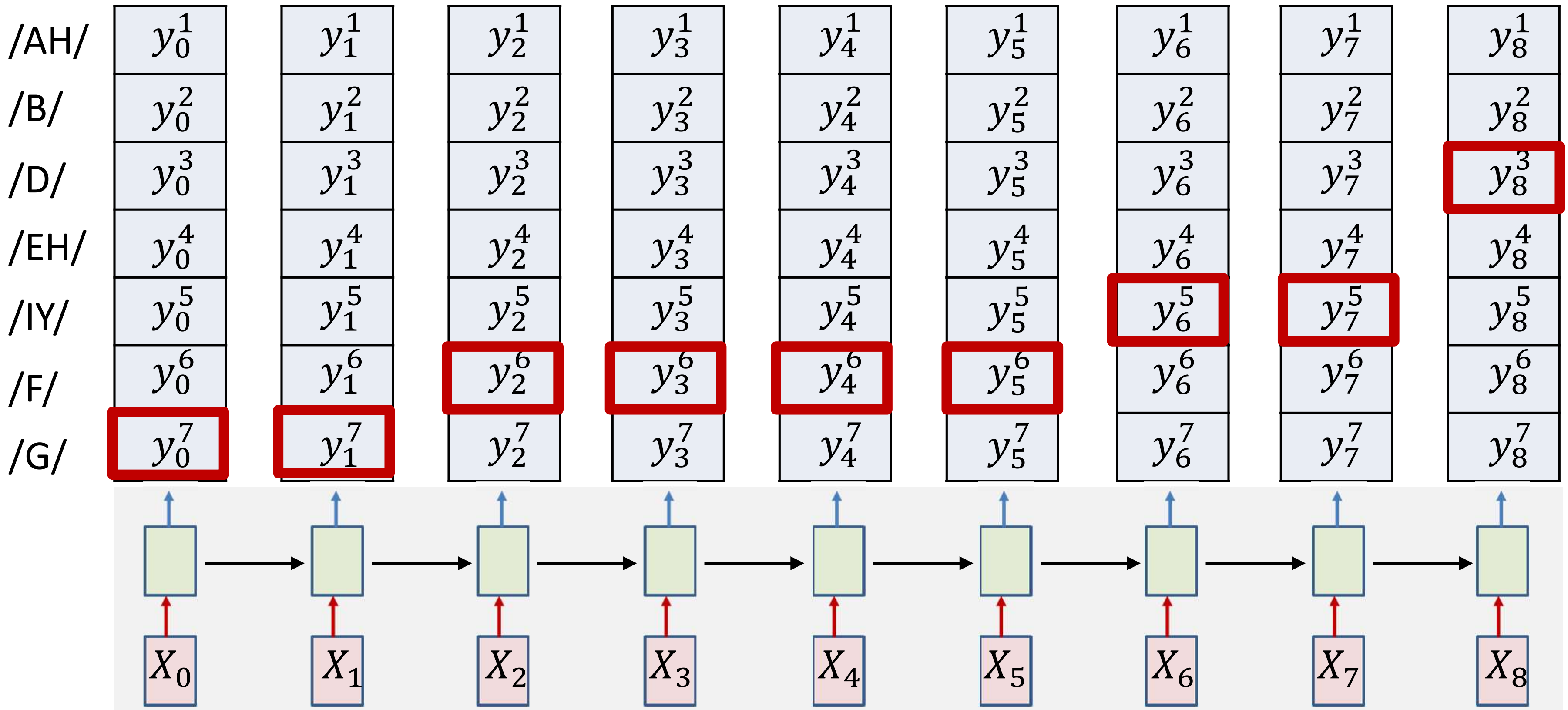
Overall objective



- Find most likely symbol sequence given inputs

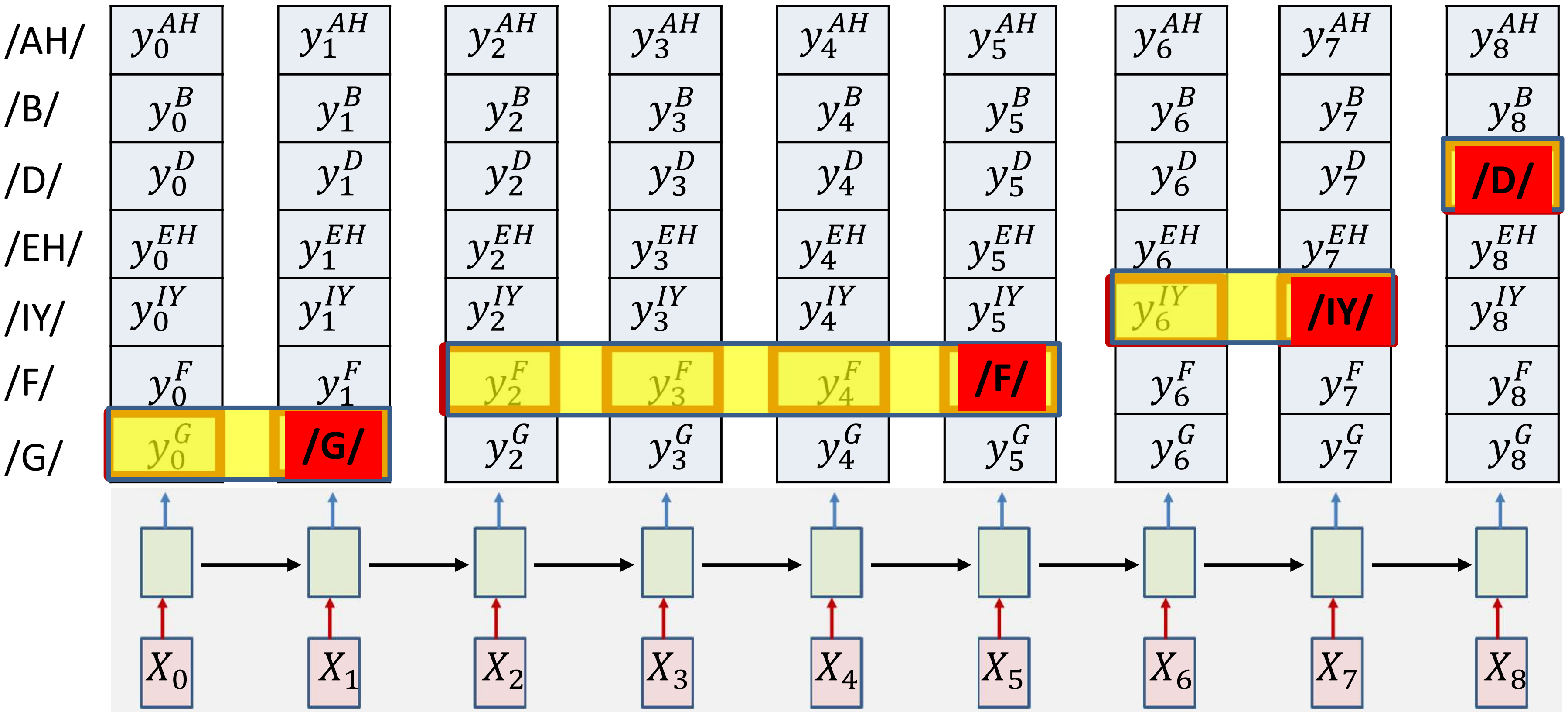
$$S_0 \ldots S_{K-1} = \underset{S'_0 \ldots S'_{K-1}}{\operatorname{argmax}} \; prob(S'_0 \ldots S'_{K-1} | X_0 \ldots X_{N-1})$$

Finding the best output



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

Finding the best output



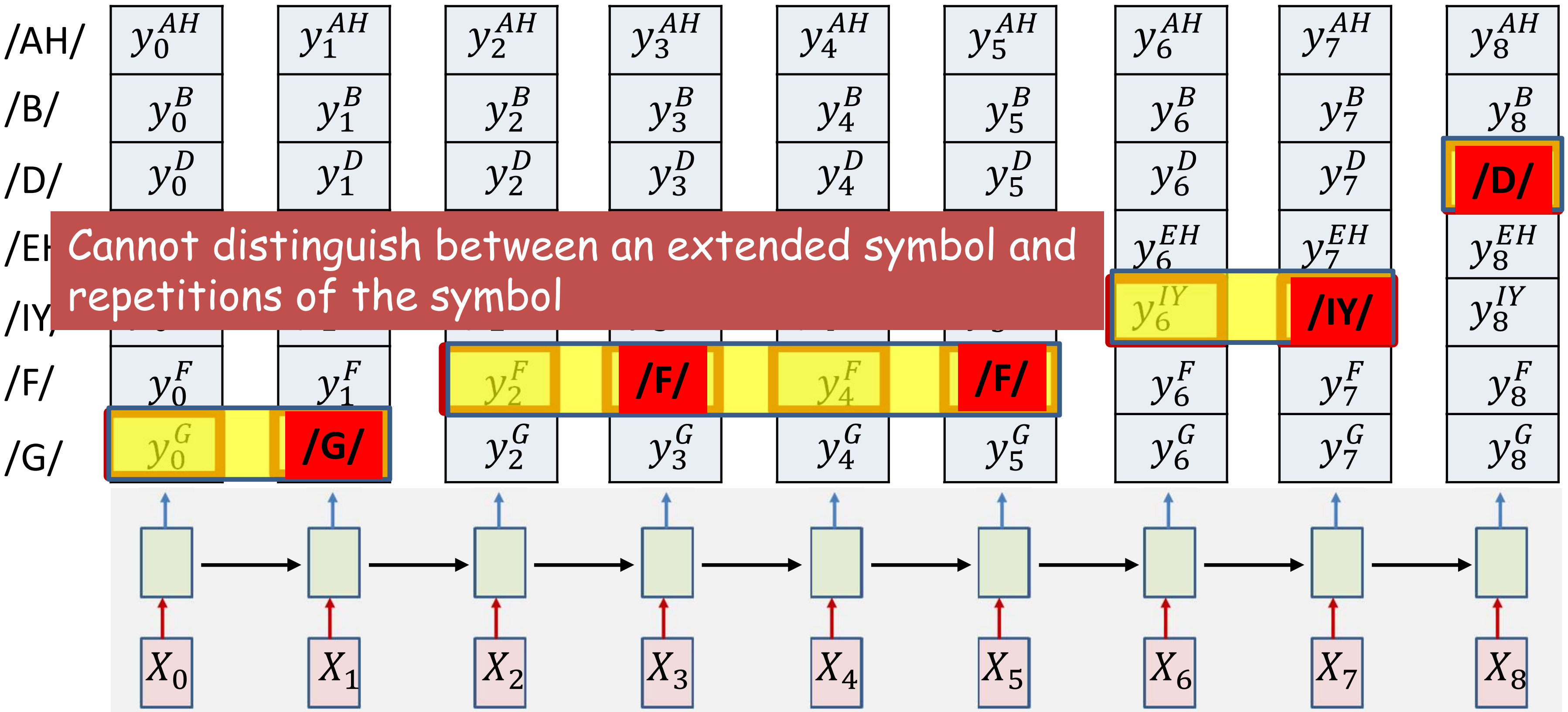
- Option 1: Simply select the most probable symbol at each time
  - *Merge* adjacent repeated symbols, and place the actual emission of the symbol in the final instant

Simple pseudocode

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

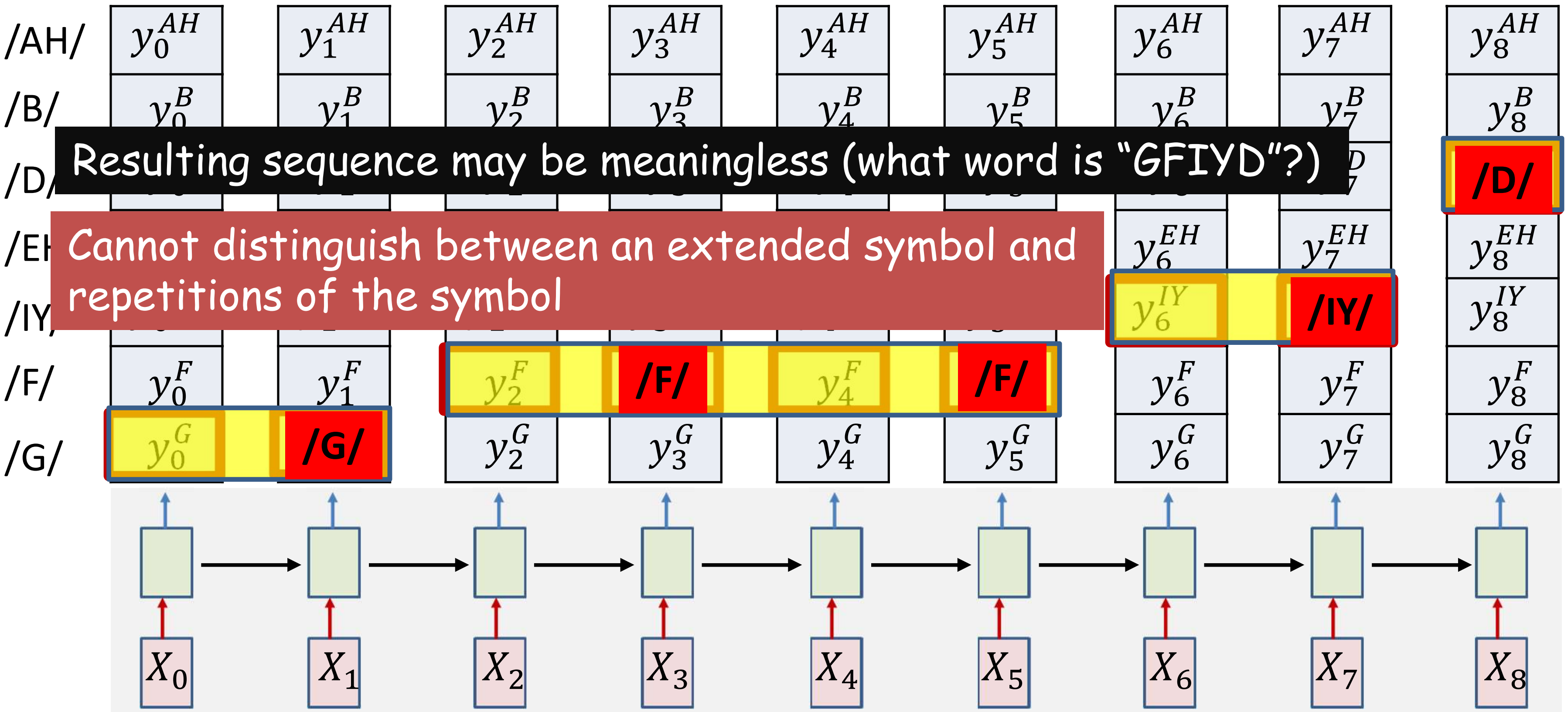
```
n = 1
best(1)= argmax_i(y(1,i))
for t = 1:T
    best(t)= argmax_i(y(t,i))
    if (best(t) != best(t-1))
        out(n) = best(t-1)
        time(n) = t-1
        n = n+1
```

Finding the best output



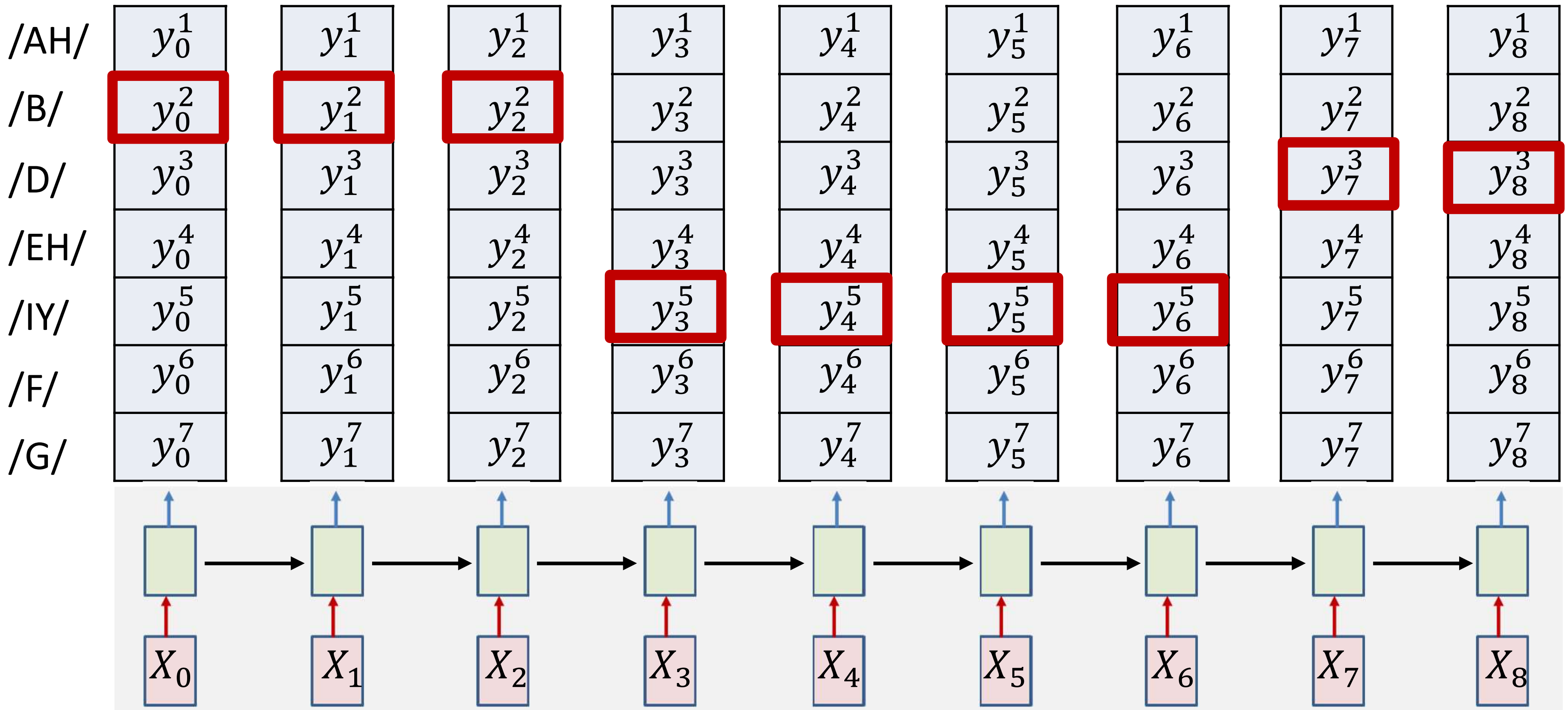
- Option 1: Simply select the most probable symbol at each time
  - *Merge* adjacent repeated symbols, and place the actual emission of the symbol in the final instant

Finding the best output



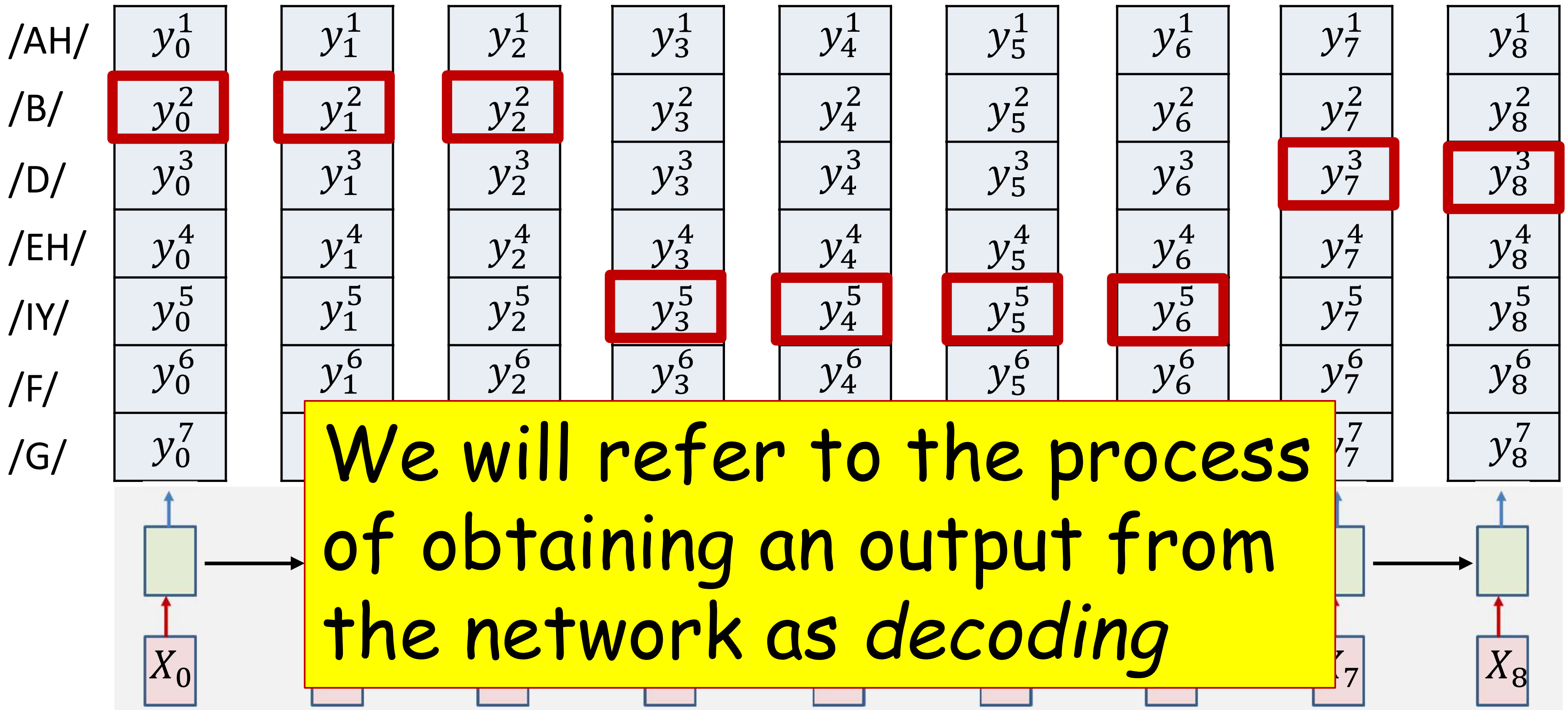
- Option 1: Simply select the most probable symbol at each time
  - *Merge* adjacent repeated symbols, and place the actual emission of the symbol in the final instant

Finding the best output



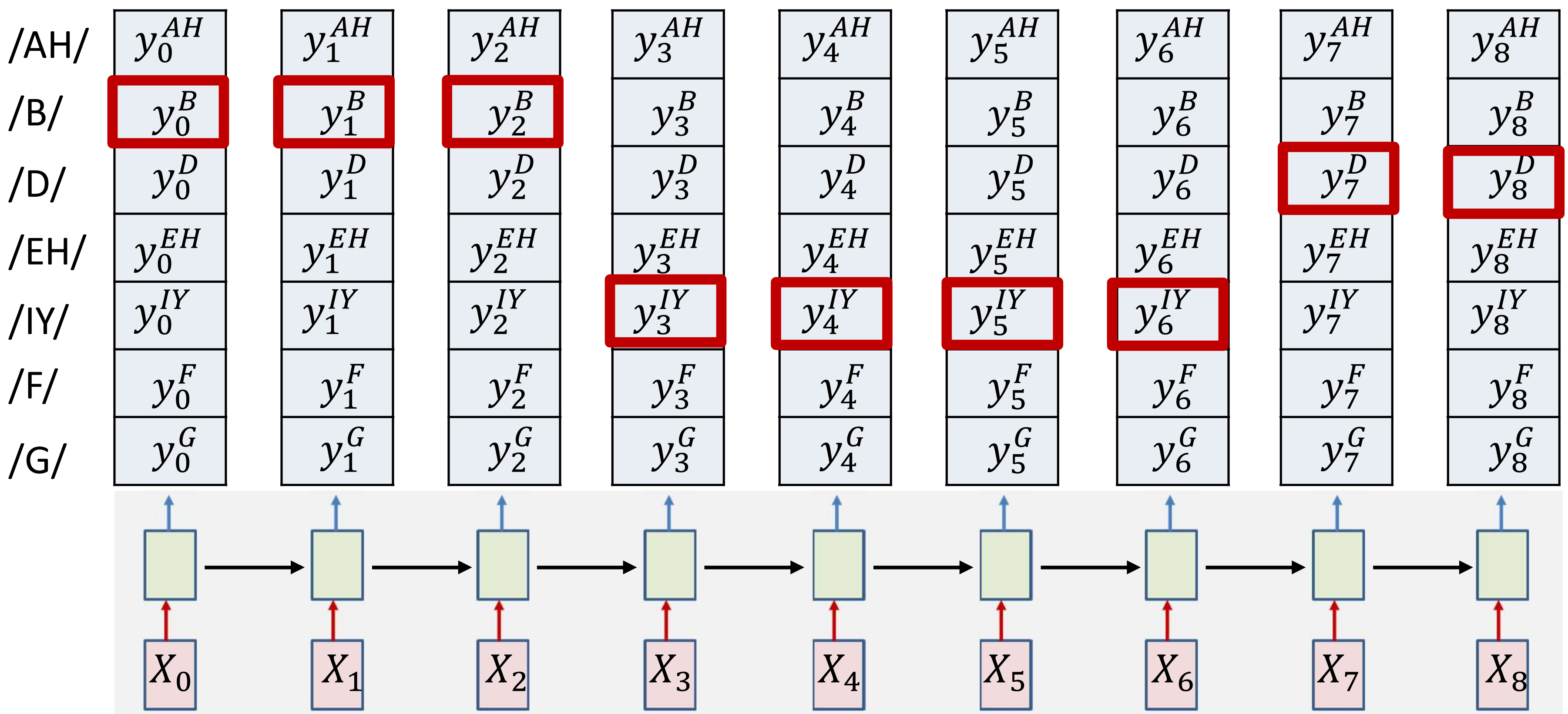
- Option 2: Impose external constraints on what sequences are allowed
  - *E.g.* only allow sequences corresponding to dictionary words
  - *E.g. Sub-symbol* units (like in HW1 – what were they?)
  - *E.g.* using special “separating” symbols to separate repetitions

Finding the best output



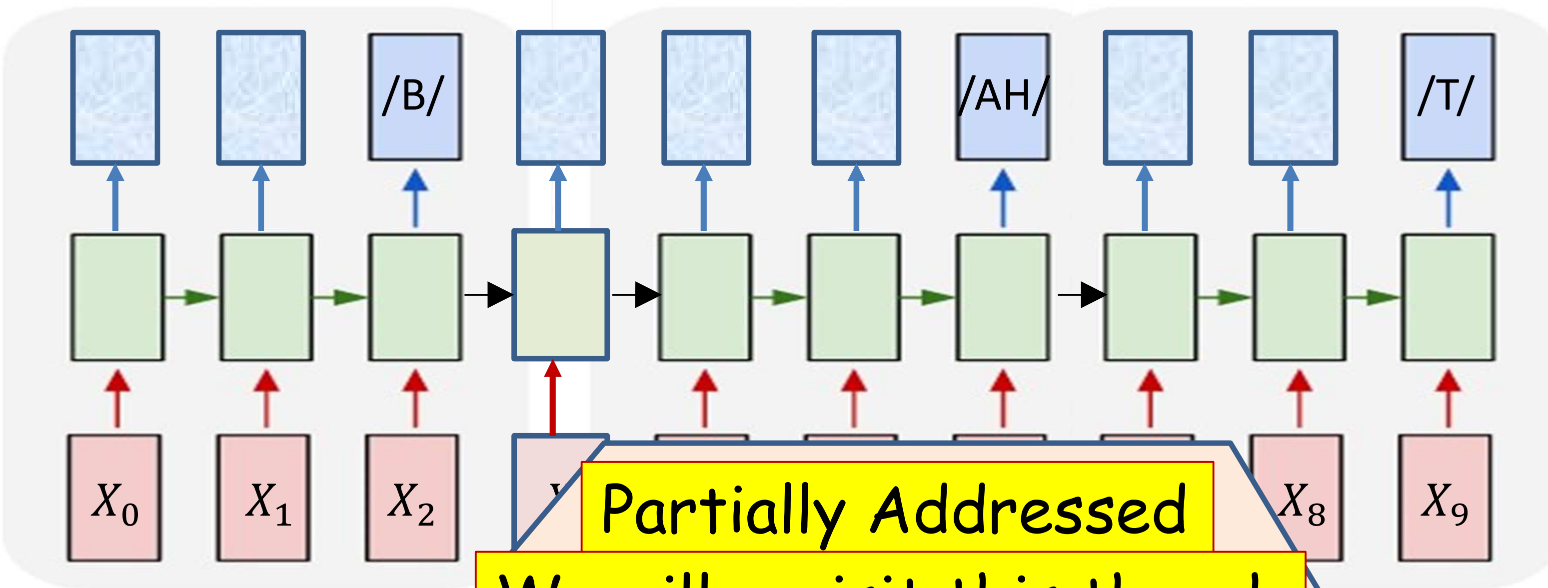
- Option 2: Impose external constraints on what sequences are allowed
  - *E.g.* only allow sequences corresponding to dictionary words
  - *E.g. Sub-symbol* units (like in HW1 – what were they?)
  - *E.g.* using special "separating" symbols to separate repetitions

Decoding



- 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
    - The "merging" heuristics do not guarantee optimal order-synchronous sequences
  - We will return to this topic later

The *sequence-to-sequence* problem



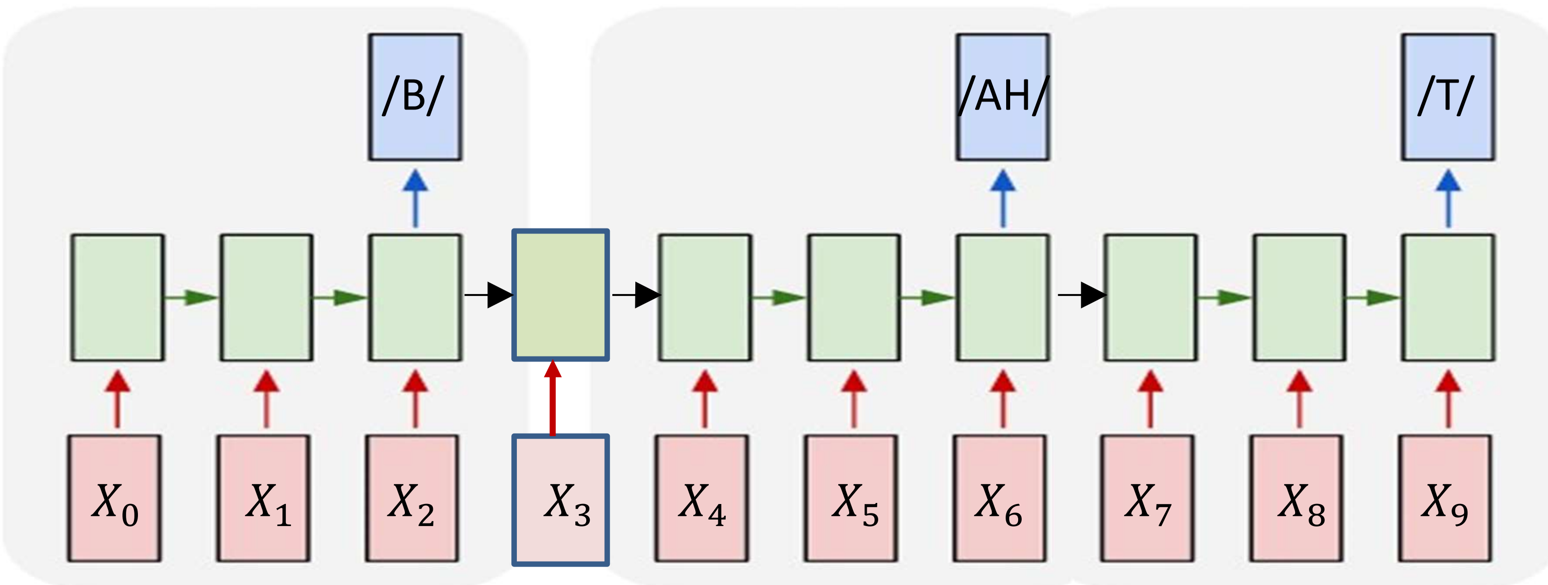
Partially Addressed

We will revisit this though

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

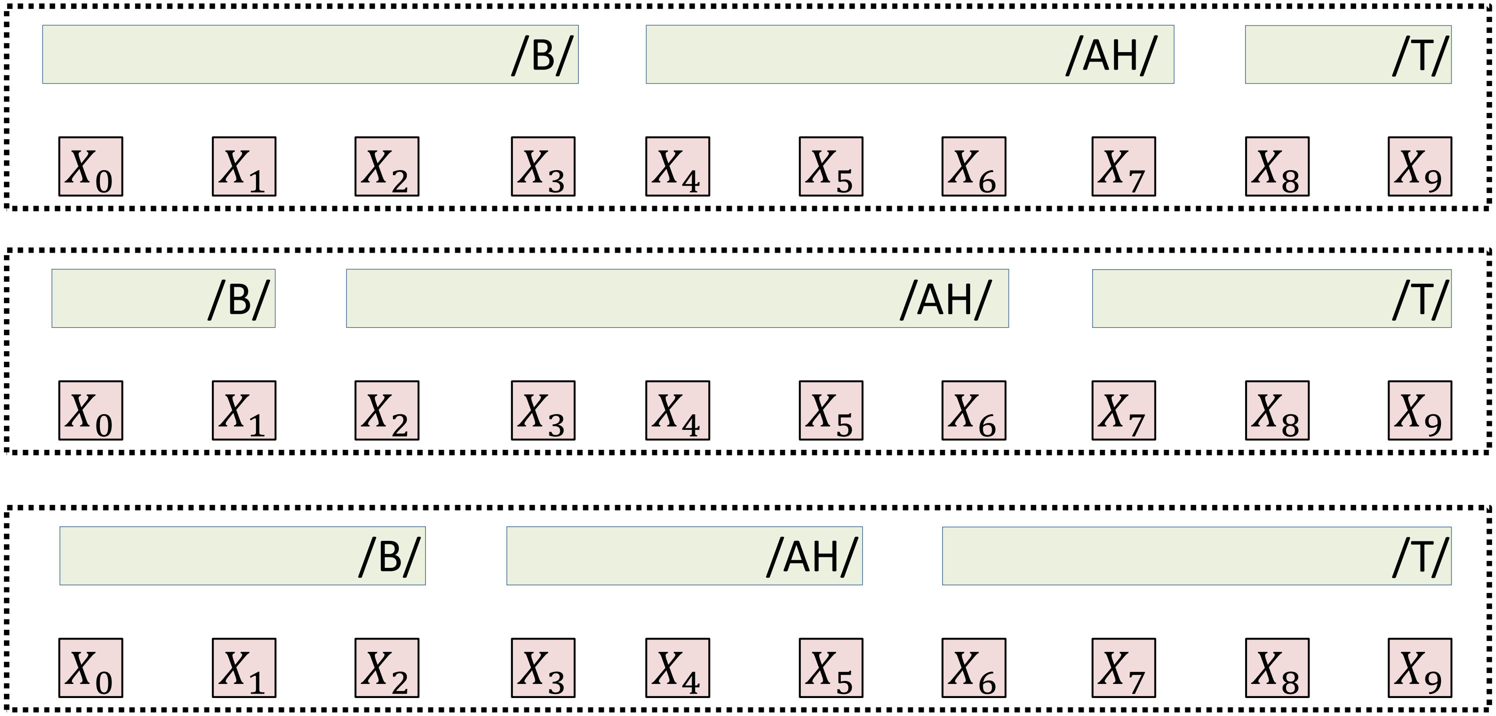
- How do we *train* these models?

Training



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

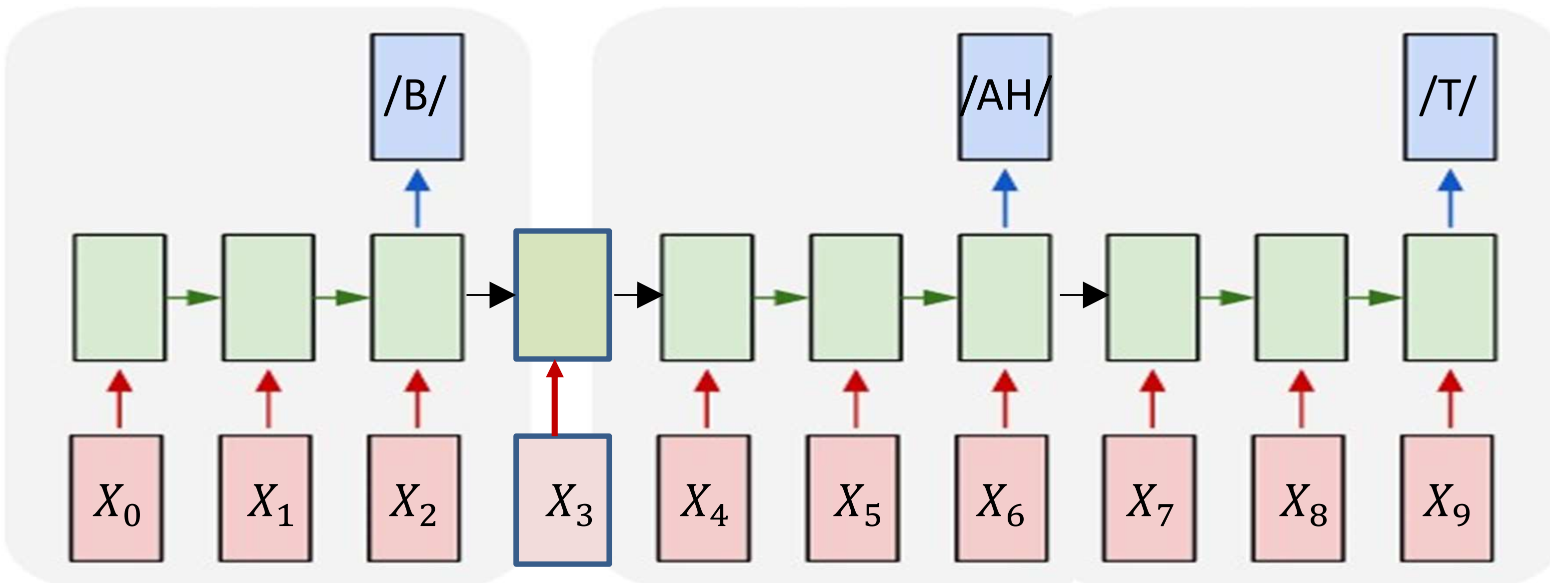
The "alignment" of labels



- The time-stamps of the output symbols give us the "alignment" of the output sequence to the input sequence
  - Which portion of the input aligns to what symbol

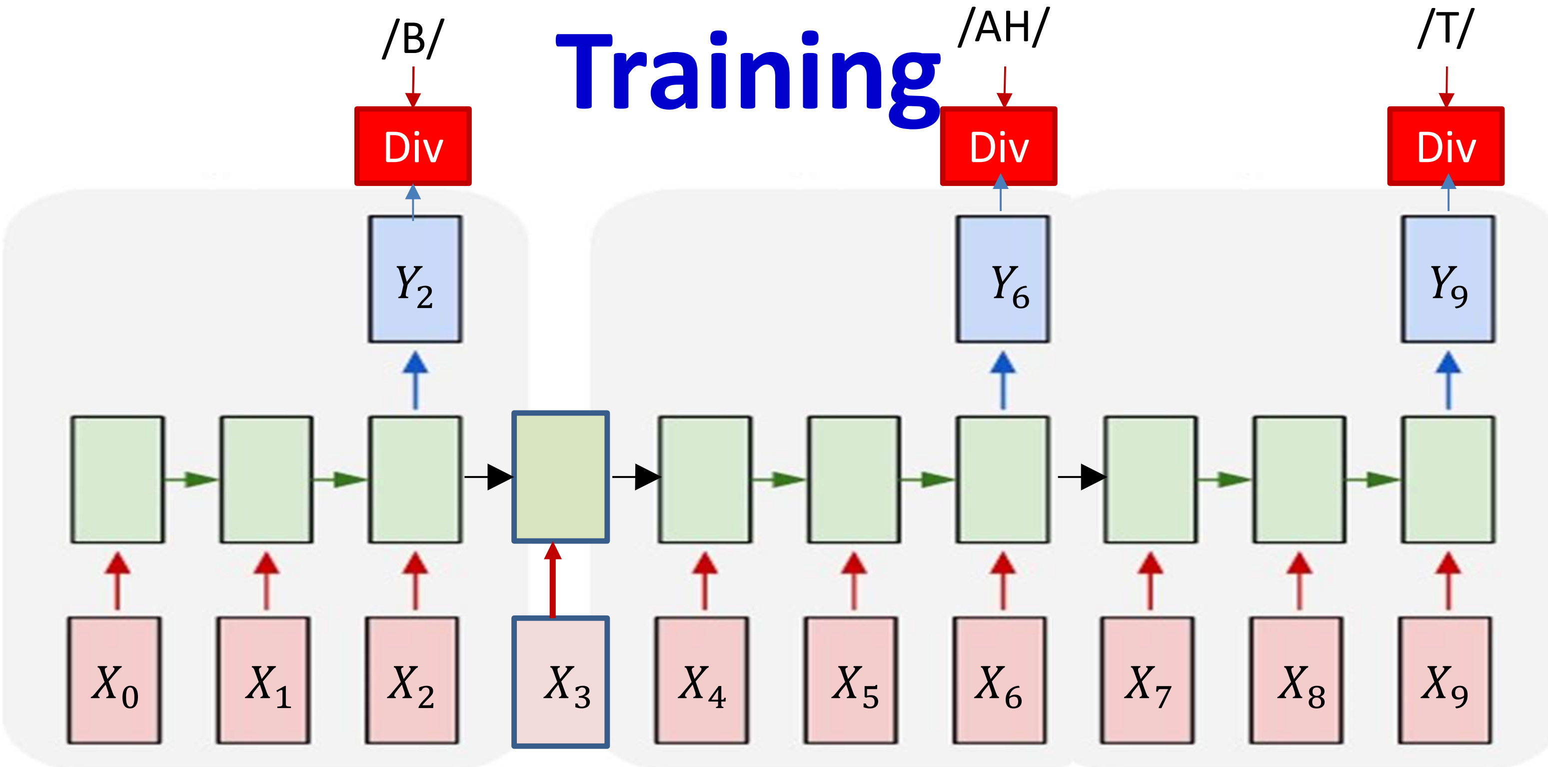
- Simply knowing the output sequence does not provide us the alignment
  - This is extra information

Training with alignment

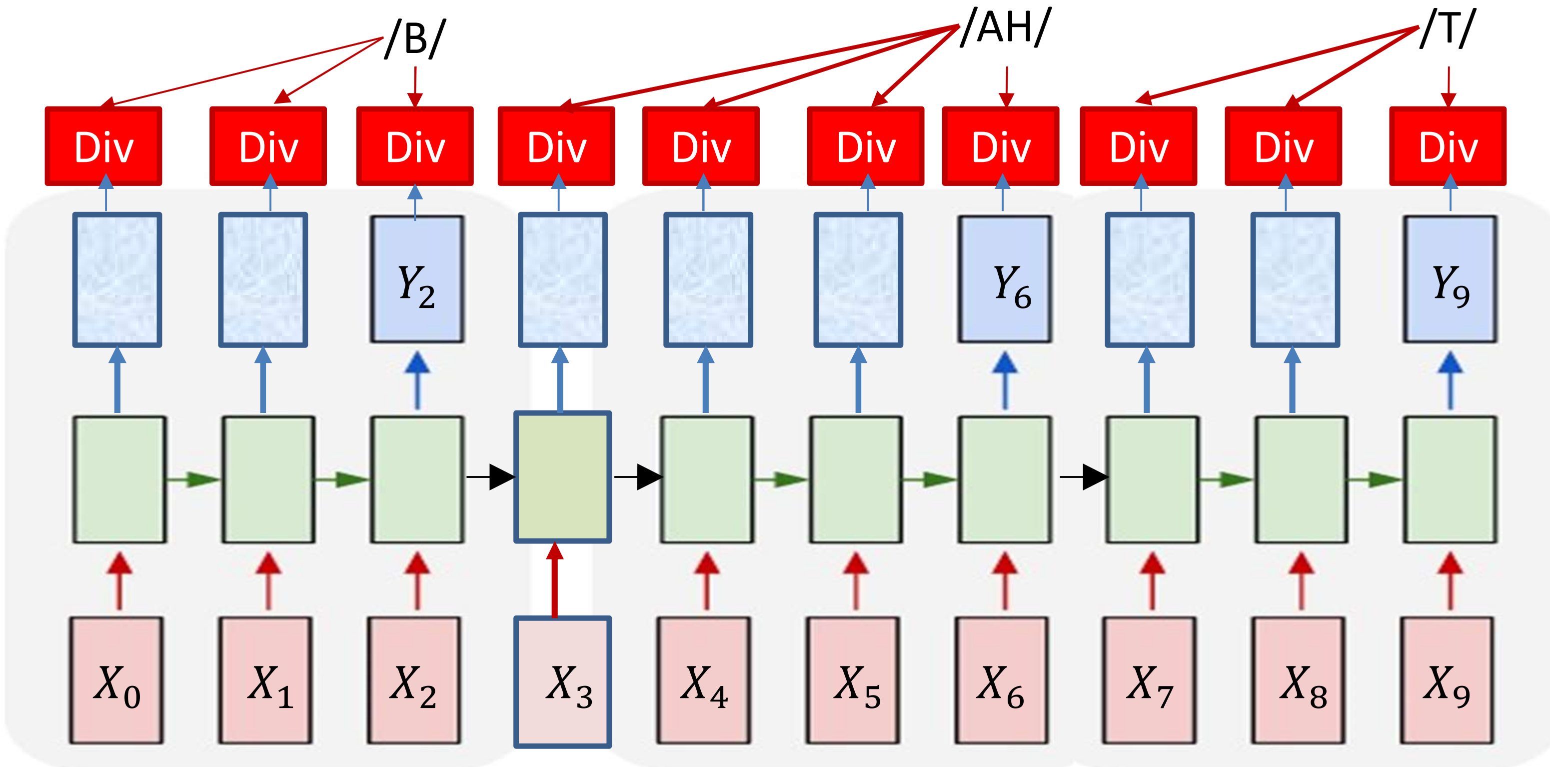


- Training data: input sequence + output sequence
  - Output sequence length <= input sequence length
- Given the *alignment* of the output to the input
  - The phoneme /B/ ends at X_2, /AH/ at X_6, /T/ at X_9

Training



- Either just define Divergence as:
  $$DIV = KL(Y_2, B) + KL(Y_6, AH) + KL(Y_9, T)$$
- Or..



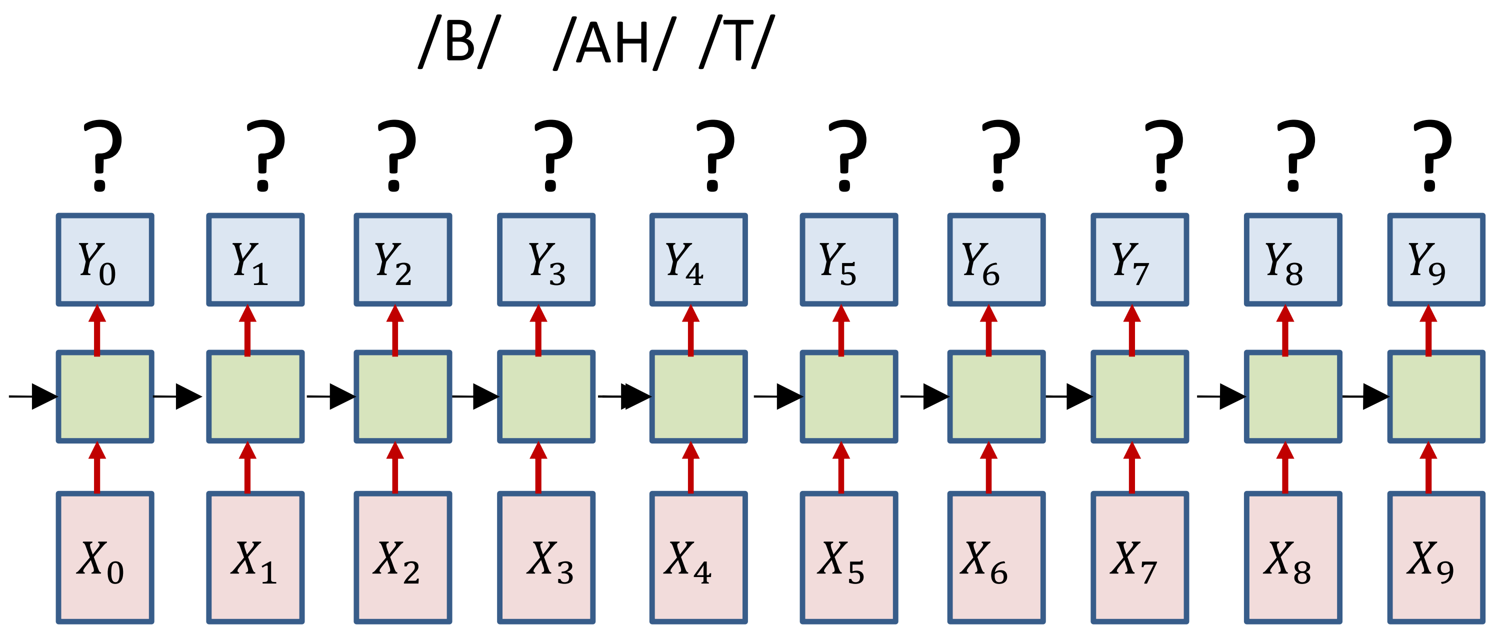
- Either just define Divergence as:

$$DIV = Xent(Y_2, B) + Xent(Y_6, AH) + Xent(Y_9, T)$$

- Or *repeat the symbols over their duration*

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

Problem: No timing information provided



- Only the sequence of output symbols is provided for the training data
  - But no indication of which one occurs where

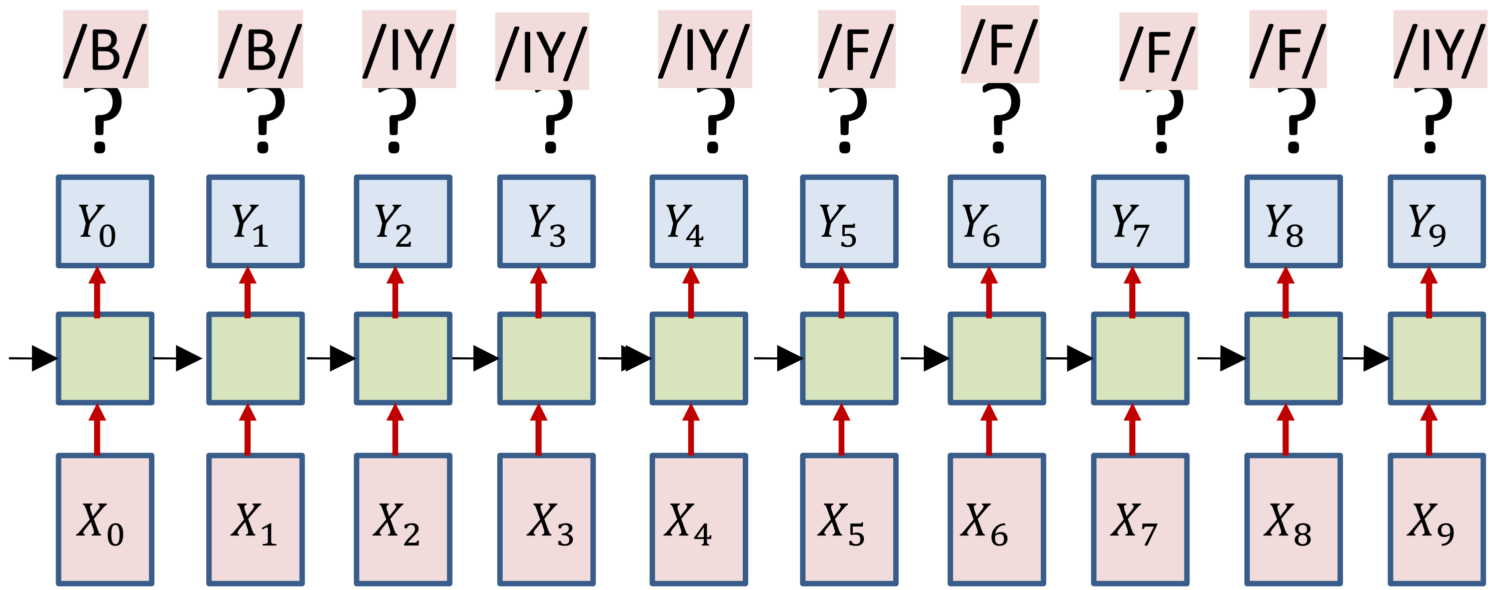
- How do we compute the divergence?
  - And how do we compute its gradient w.r.t. Y_t

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

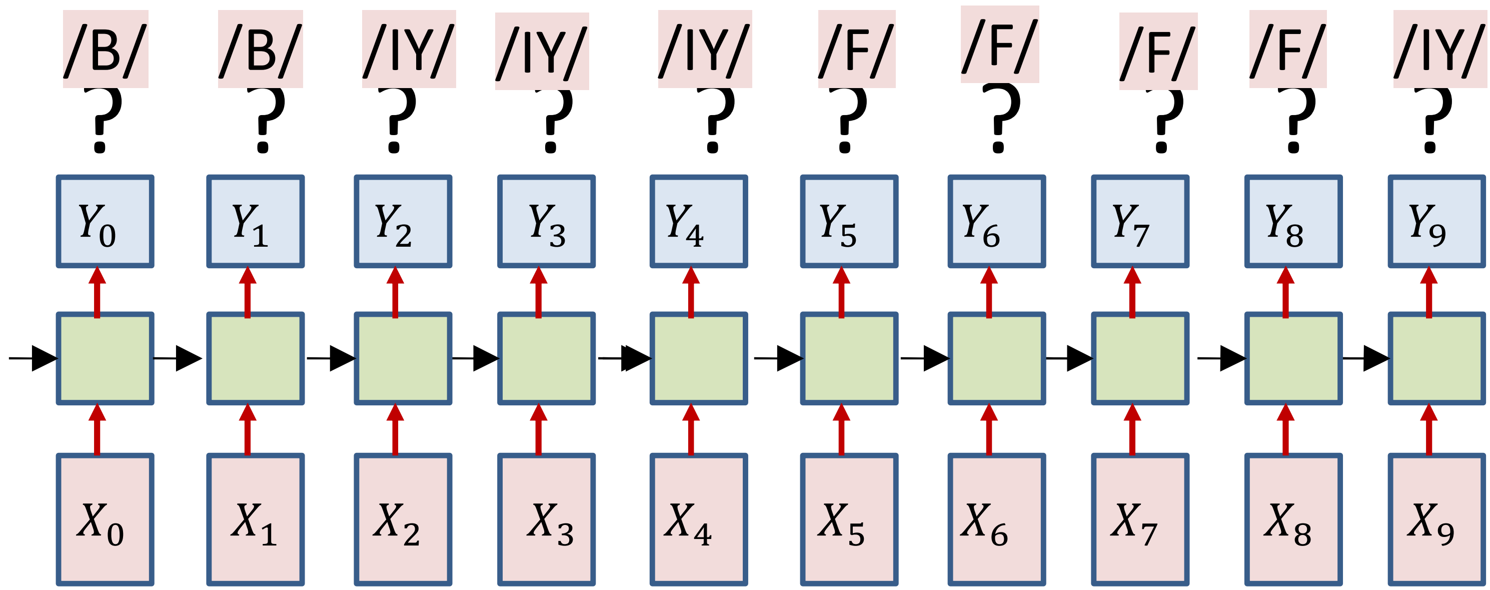
Solution 1: *Guess the alignment*



- Guess an initial alignment and iteratively refine it as the model improves

- 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

Solution 1: *Guess the alignment*



- Guess an initial alignment and iteratively refine it as the model improves

- 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

Characterizing the alignment

/B/	/B/	/B/	**/B/**	/AH/	/AH/	/AH/	**/AH/**	/T/	**/T/**
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9

/B/	**/B/**	/AH/	/AH/	/AH/	/AH/	**/AH/**	/T/	/T/	**/T/**
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9

/B/	/B/	**/B/**	/AH/	/AH/	**/AH/**	/T/	/T/	/T/	**/T/**
X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9

- An alignment can be represented as a repetition of symbols
  - Examples show different alignments of /B/ /AH/ /T/ to $X_0 \ldots X_9$

Estimating an alignment

- Given:
  - The unaligned K-length symbol sequence $S = S_0 \dots S_{K-1}$ (e.g. /B/ /IY/ /F/ /IY/)
  - An N-length input ($N \geq K$)
  - And a (trained) recurrent network

- Find:
  - An N-length expansion $s_0 \dots s_{N-1}$ comprising the symbols in S in strict order
    - e.g. $S_0 S_1 S_1 S_2 S_3 S_3 \dots S_{K-1}$
      - i.e.$s_0 = S_0, s_2 = S_1, s_3 = S_1, s_4 = S_2, s_5 = S_3, \dots s_{N-1} = S_{K-1}$
    - E.g. /B/ /B/ /IY/ /IY/ /IY/ /F/ /F/ /F/ /F/ /IY/ ..

- Outcome: an *alignment* of the target symbol sequence $S_0 \dots S_{K-1}$ to the input $X_0 \dots X_{N-1}$

Estimating an alignment

- Alignment problem:

- Find

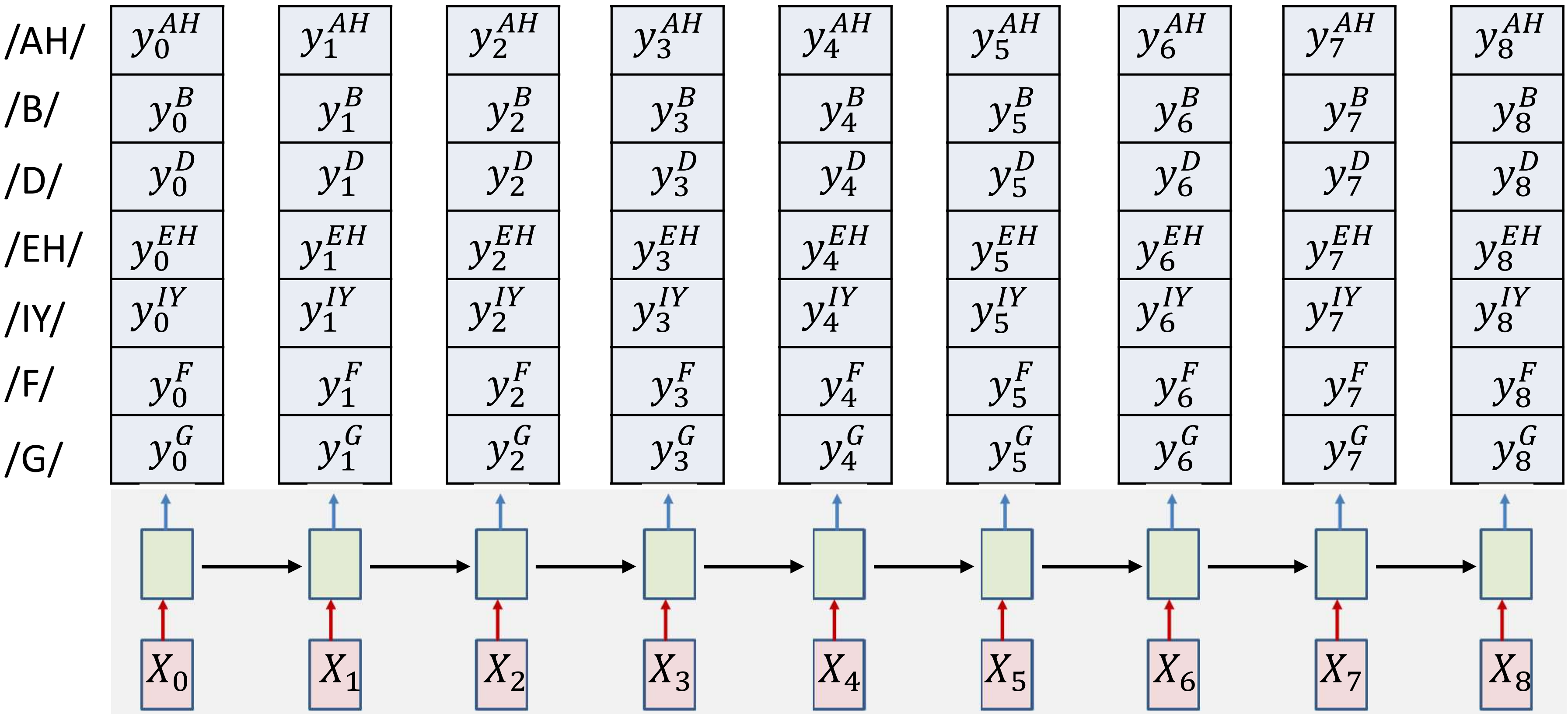
$$argmax\ P(s_0, s_1, \ldots, s_{N-1} | S_0, S_1, \ldots, S_K, X_0, X_1, \ldots, X_{N-1})$$

  - Such that

$$compress(s_0, s_1, \ldots, s_{N-1}) \equiv S_0, S_1, \ldots, S_K$$

- $compress()$ is the operation of compressing repetitions into one

Recall: The actual output of the network



- At each time the network outputs a probability for *each* output symbol

Recall: unconstrained decoding

/AH/	$\boxed{y_0^{AH}}$	$\boxed{y_1^{AH}}$	$\boxed{y_2^{AH}}$	y_3^{AH}	y_4^{AH}	$\boxed{y_5^{AH}}$	y_6^{AH}	y_7^{AH}	y_8^{AH}
/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}
/D/	y_0^{D}	y_1^{D}	y_2^{D}	$\boxed{y_3^{D}}$	$\boxed{y_4^{D}}$	y_5^{D}	y_6^{D}	y_7^{D}	y_8^{D}
/EH/	y_0^{EH}	y_1^{EH}	y_2^{EH}	y_3^{EH}	y_4^{EH}	y_5^{EH}	y_6^{EH}	y_7^{EH}	y_8^{EH}
/IY/	y_0^{IY}	y_1^{IY}	y_2^{IY}	y_3^{IY}	y_4^{IY}	y_5^{IY}	y_6^{IY}	$\boxed{y_7^{IY}}$	$\boxed{y_8^{IY}}$
/F/	y_0^{F}	y_1^{F}	y_2^{F}	y_3^{F}	y_4^{F}	y_5^{F}	$\boxed{y_6^{F}}$	y_7^{F}	y_8^{F}
/G/	y_0^{G}	y_1^{G}	y_2^{G}	y_3^{G}	y_4^{G}	y_5^{G}	y_6^{G}	y_7^{G}	y_8^{G}

- We find the most likely sequence of symbols
  - (Conditioned on input $X_0 \dots X_{N-1}$)
- This may not correspond to an expansion of the desired symbol sequence
  - E.g. the unconstrained decode may be /AH//AH//AH//D//D//AH//F//IY//IY/
    - Contracts to /AH/ /D/ /AH/ /F/ /IY/
  - Whereas we want an expansion of /B//IY//F//IY/

Constraining the alignment: Try 1

[redacted]									
/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
[redacted]									
/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}
/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
[redacted]									

- Block out all rows that do not include symbols from the target sequence
  - E.g. Block out rows that are not /B/ /IY/ or /F/

Blocking out unnecessary outputs

Compute the entire output (for all symbols)

Copy the output values for the target symbols into the secondary reduced structure

Constraining the alignment: Try 1

/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}
/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

- Only decode on reduced grid
  - We are now assured that only the appropriate symbols will be hypothesized

Constraining the alignment: Try 1

/B/	$\boxed{y_0^B}$	$\boxed{y_1^B}$	y_2^B	y_3^B	$\boxed{y_4^B}$	$\boxed{y_5^B}$	y_6^B	y_7^B	y_8^B
/IY/	y_0^{IY}	y_1^{IY}	$\boxed{y_2^{IY}}$	y_3^{IY}	y_4^{IY}	y_5^{IY}	y_6^{IY}	y_7^{IY}	$\boxed{y_8^{IY}}$
/F/	y_0^F	y_1^F	y_2^F	$\boxed{y_3^F}$	y_4^F	y_5^F	$\boxed{y_6^F}$	$\boxed{y_7^F}$	y_8^F

- Only decode on reduced grid
  - We are now assured that only the appropriate symbols will be hypothesized
- Problem: This still doesn't assure that the decode sequence correctly expands the target symbol sequence
  - E.g. the above decode is not an expansion of /B//IY//F//IY/
- Still needs additional constraints

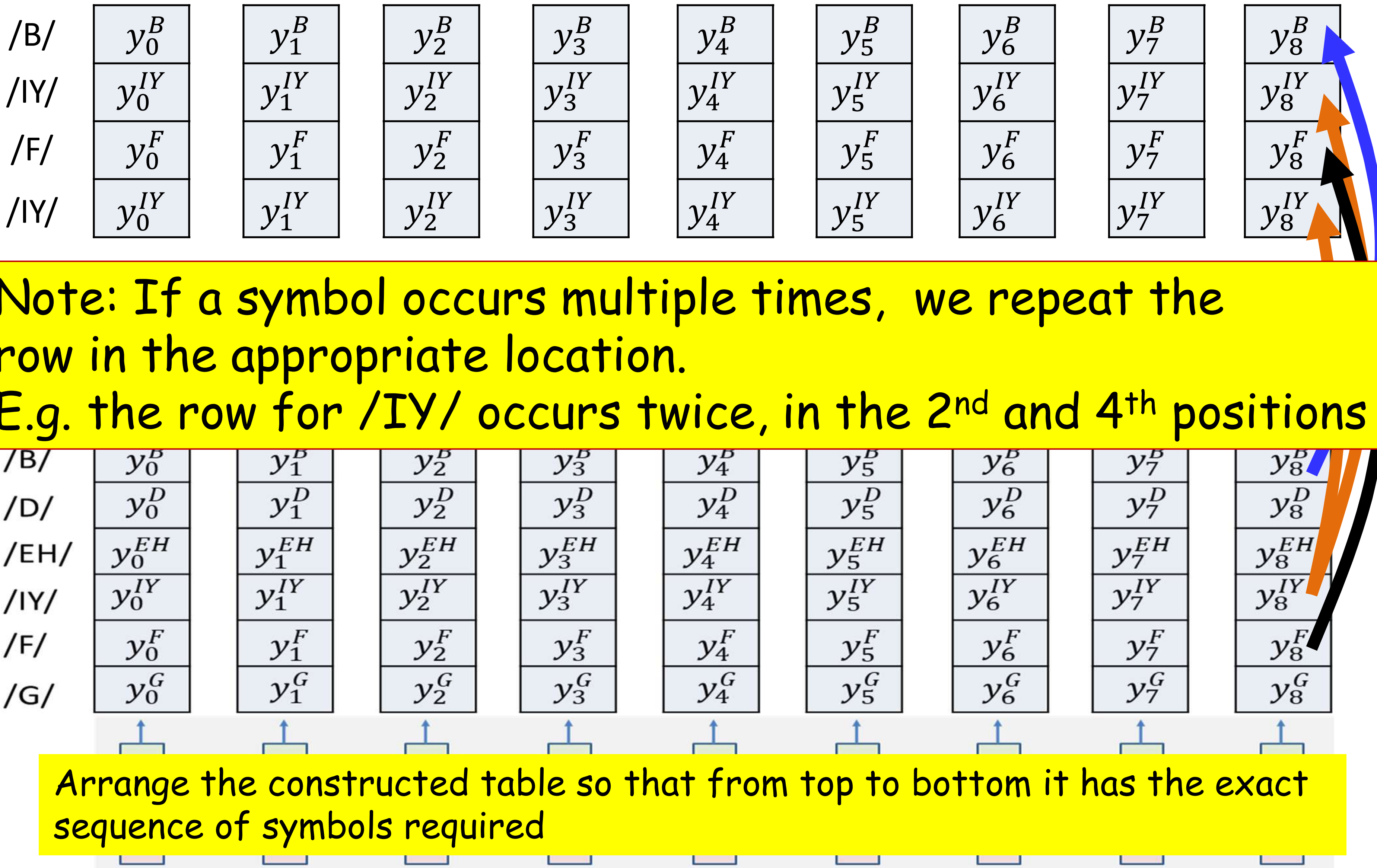
Try 2: Explicitly arrange the constructed table

/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}
/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
/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}

/AH/	y_0^{AH}	y_1^{AH}	y_2^{AH}	y_3^{AH}	y_4^{AH}	y_5^{AH}	y_6^{AH}	y_7^{AH}	y_8^{AH}
/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
/D/	y_0^D	y_1^D	y_2^D	y_3^D	y_4^D	y_5^D	y_6^D	y_7^D	y_8^D
/EH/	y_0^{EH}	y_1^{EH}	y_2^{EH}	y_3^{EH}	y_4^{EH}	y_5^{EH}	y_6^{EH}	y_7^{EH}	y_8^{EH}
/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}
/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
/G/	y_0^G	y_1^G	y_2^G	y_3^G	y_4^G	y_5^G	y_6^G	y_7^G	y_8^G

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

Try 2: Explicitly arrange the constructed table



Note: If a symbol occurs multiple times, we repeat the row in the appropriate location.
E.g. the row for /IY/ occurs twice, in the 2nd and 4th positions

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

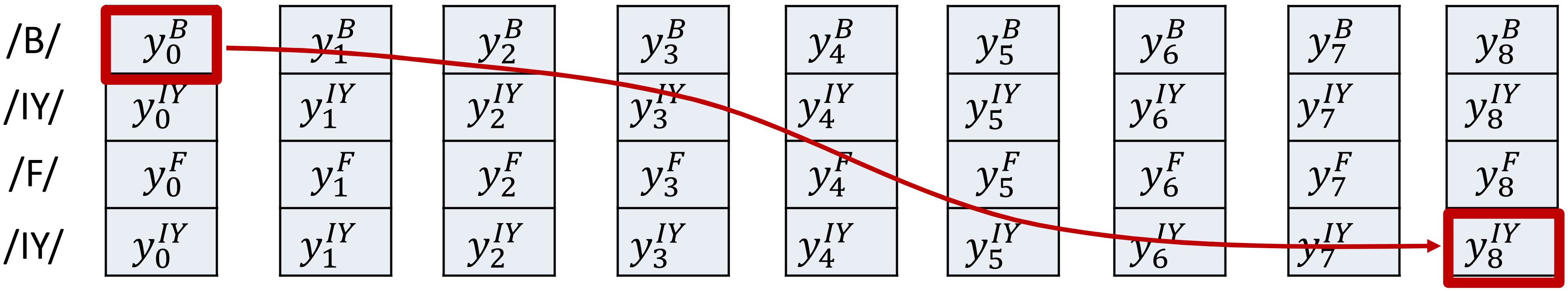
Composing the graph

```
#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))
```

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

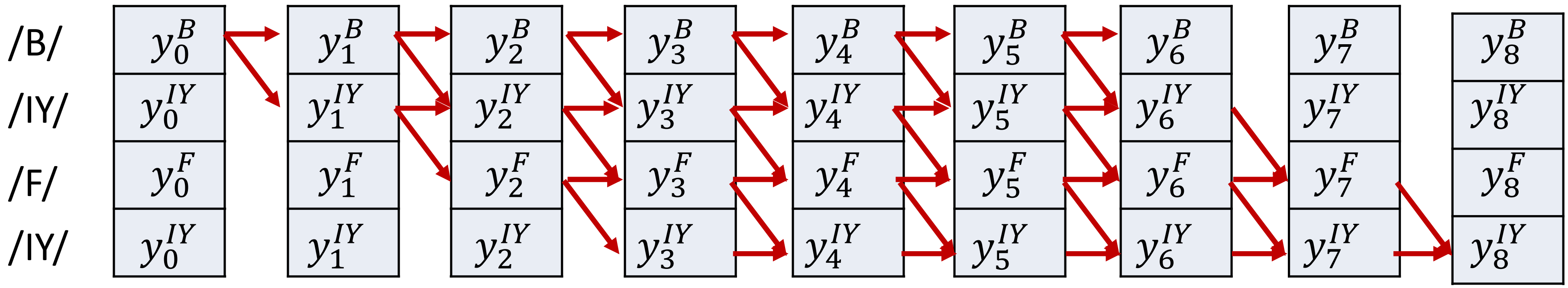
/B/	y_0^B		y_1^B		y_2^B		y_3^B		y_4^B		y_5^B		y_6^B		y_7^B		y_8^B
/IY/	y_0^{IY}		y_1^{IY}		y_2^{IY}		y_3^{IY}		y_4^{IY}		y_5^{IY}		y_6^{IY}		y_7^{IY}		y_8^{IY}
/F/	y_0^F		y_1^F		y_2^F		y_3^F		y_4^F		y_5^F		y_6^F		y_7^F		y_8^F
/IY/	y_0^{IY}		y_1^{IY}		y_2^{IY}		y_3^{IY}		y_4^{IY}		y_5^{IY}		y_6^{IY}	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

Explicitly constrain alignment



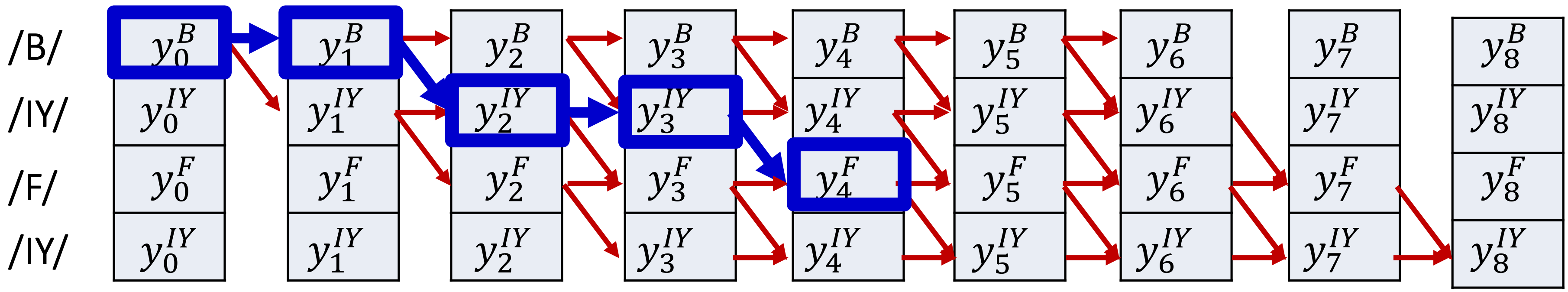
- Constrain that the first symbol in the decode *must* be the top left block
- The last symbol *must* be the bottom right
- The rest of the symbols must follow a sequence that *monotonically* travels down from top left to bottom right
  - I.e. symbol chosen at any time is at the same level or at the next level to the symbol at the previous time
- This guarantees that the sequence *is* an expansion of the target sequence
  - /B/ /IY/ /F/ /IY/ in this case

Explicitly constrain alignment



- Compose a graph such that every path in the graph from source to sink represents a valid alignment
  - Which maps on to the target symbol sequence (/B//IY//F//IY/)
- Edge scores are 1
- Node scores are the probabilities assigned to the symbols by the neural network

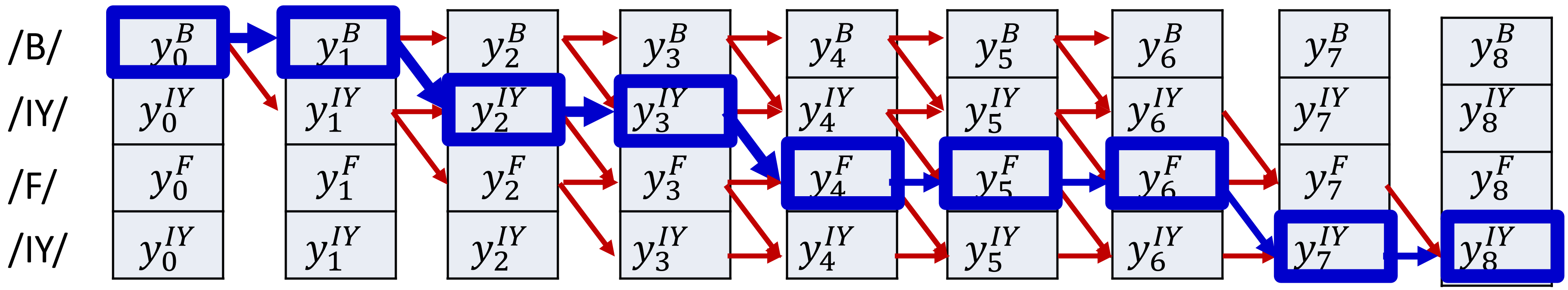
Path Score (probability)



- Compose a graph such that every path in the graph from source to sink represents a valid alignment
  - Which maps on to the target symbol sequence (/B//IY//F//IY/)
- Edge scores are 1
- Node scores are the probabilities assigned to the symbols by the neural network
- **The "score" of a path is the product of the probabilities of all nodes along the path**
- **E.g. the probability of the marked path is**

$$Scr(Path) = y_0^B y_1^B y_2^{IY} y_3^{IY} y_4^F$$

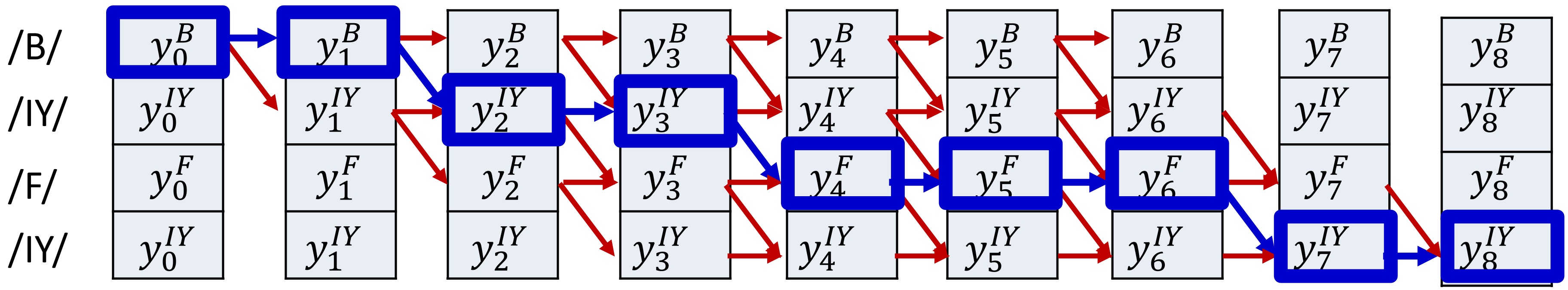
Path Score (probability)



- Compose a graph such that every path in the graph from source to sink represents a valid alignment
  - Which maps on to the target symbol sequence (/B//IY//F//IY/)
- Edge scores are 1
- Node scores are the probabilities assigned to the symbols by the neural network
- **The "score" of a path is the product of the probabilities of all nodes along the path**

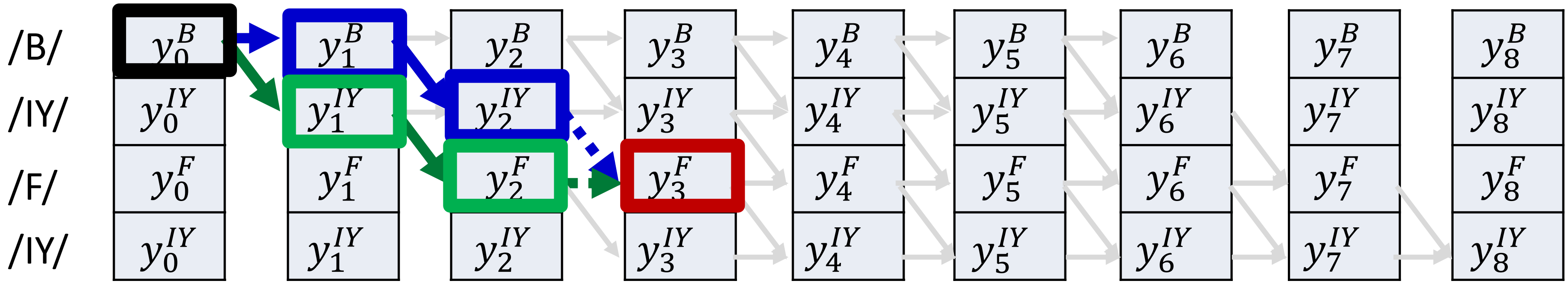
Figure shows a typical end-to-end path. There are an exponential number of such paths. Challenge: Find the path with the highest score (probability)

Explicitly constrain alignment



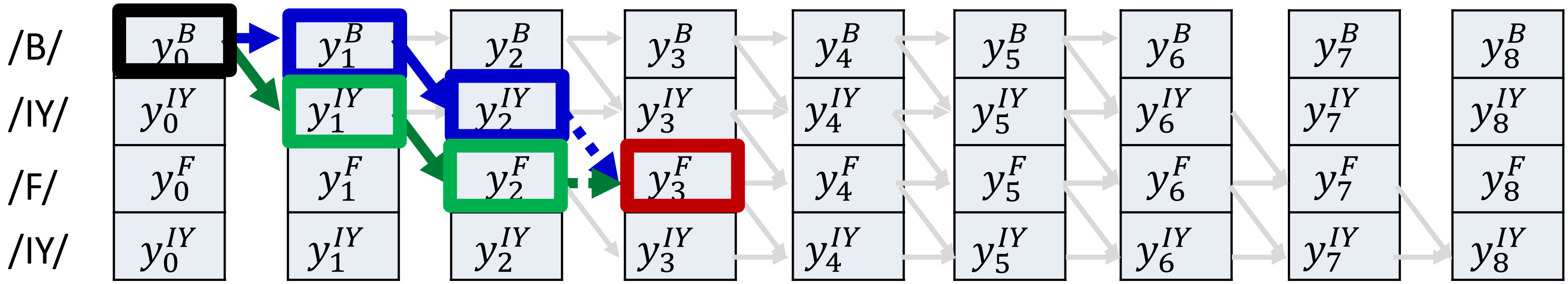
- Find the ***most probable path*** from source to sink using any dynamic programming algorithm
  - E.g. The Viterbi algorithm

Viterbi algorithm: Basic idea



- The best path to any node *must* be an extension of the best path to one of its parent nodes
  - Any other path would necessarily have a lower probability
- The best parent is simply the parent with the best-scoring best path

Viterbi algorithm: Basic idea



$$BestPath(y_0^B \rightarrow y_3^F) = BestPath(y_0^B \rightarrow y_2^{IY})y_3^F$$

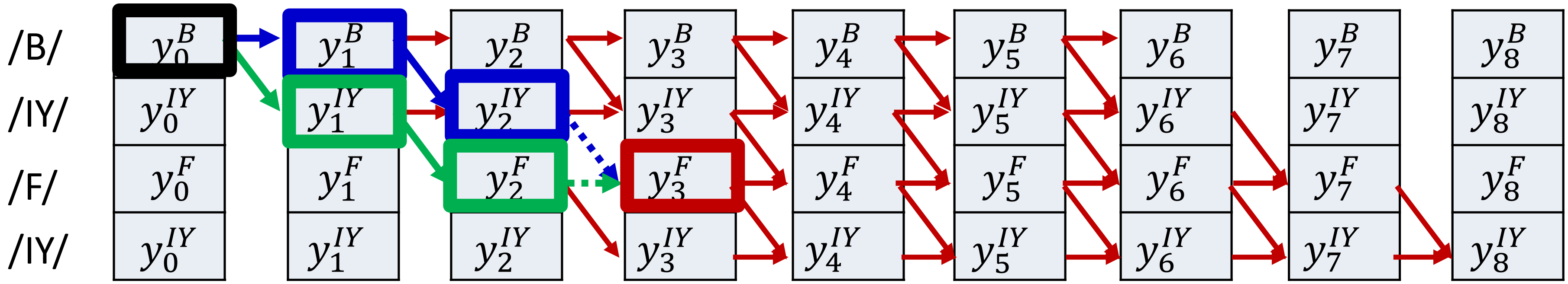
$$or \quad BestPath(y_0^B \rightarrow y_2^F)y_3^F$$

$$BestPath(y_0^B \rightarrow y_3^F) = BestPath(y_0^B \rightarrow BestParent)y_3^F$$

- The best parent is simply the parent with the best-scoring best path

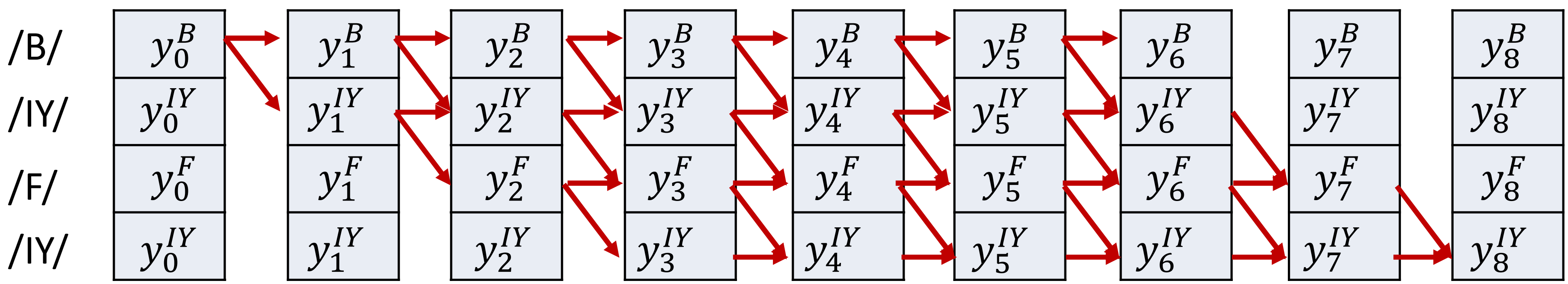
$$BestParent$$
$$= argmax_{Parent \in (y_2^{IY}, y_2^F)}(Score(BestPath(y_0^B \rightarrow Parent)))$$

Viterbi algorithm



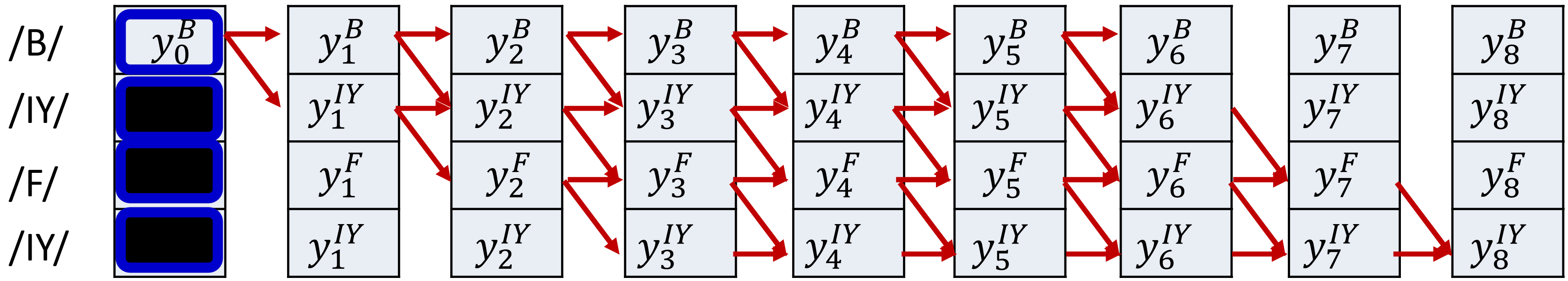
- Dynamically track the best path (and the score of the best path) from the source node to every node in the graph
  - At each node, keep track of
    - The best incoming parent edge
    - The score of the best path from the source to the node through this best parent edge
- Eventually compute the best path from source to sink

Viterbi algorithm



- First, some notation:
- $y_t^{S(r)}$ is the probability of the target symbol assigned to the r-th row in the t-th time (given inputs $X_0 \dots X_t$)
  - E.g., S(0) = /B/
    - The scores in the 0th row have the form y_t^B
  - E.g. S(1) = S(3) = /IY/
    - The scores in the 1st and 3rd rows have the form y_t^{IY}
  - E.g. S(2) = /F/
    - The scores in the 2nd row have the form y_t^F

Viterbi algorithm



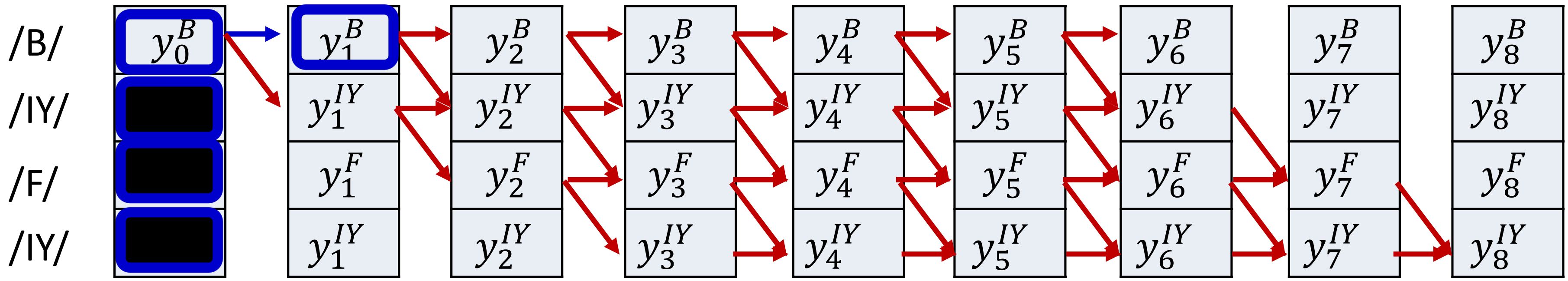
- Initialization:

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

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

BP := Best Parent
Bscr := Bestpath Score to node

Viterbi algorithm



- Initialization:

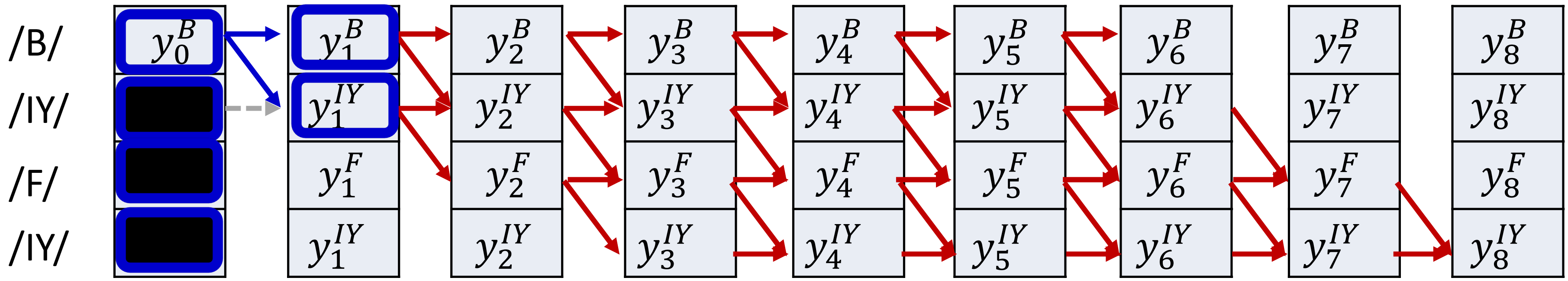
$$BP(0,i) = \ null, \ i = 0 \ldots K-1$$

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

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

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

Viterbi algorithm



- Initialization:

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

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

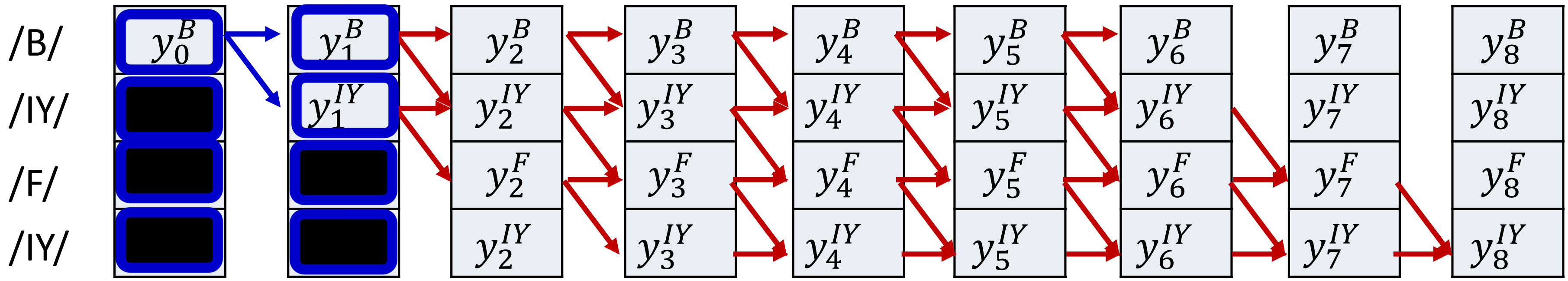
- for $t \ = \ 1 \ldots T-1$

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

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

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

Viterbi algorithm



- Initialization:

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

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

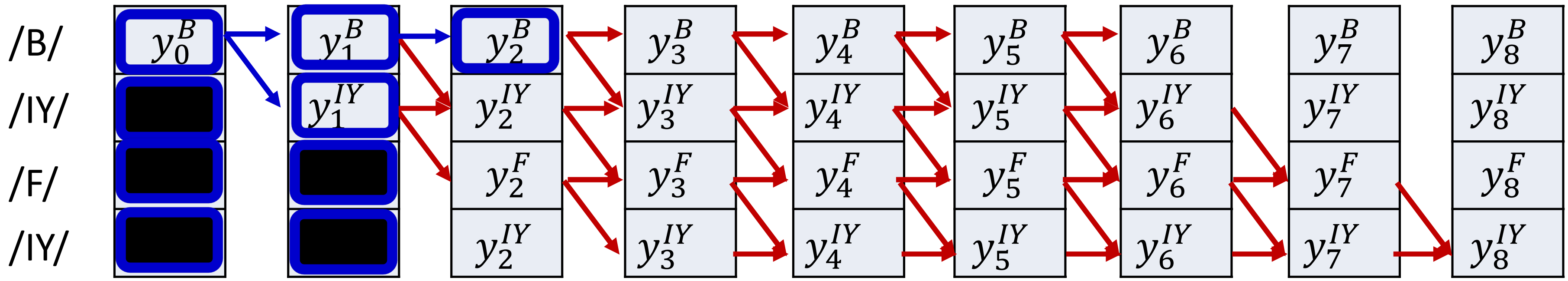
- for $t = 1 \ldots T-1$

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

for $l = 1 \ldots K-1$

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

Viterbi algorithm



- Initialization:

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

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

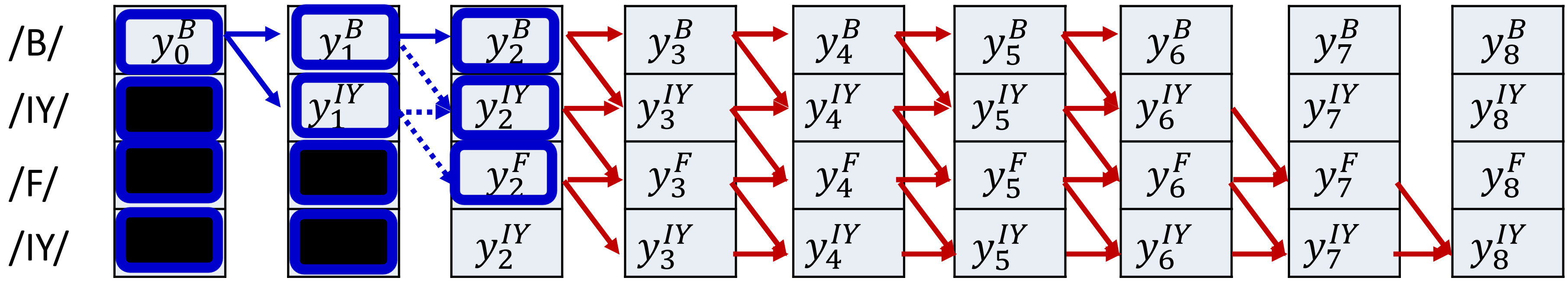
- for $t = 1 \ldots T-1$

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

for $l = 1 \ldots K-1$

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

Viterbi algorithm



- Initialization:

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

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

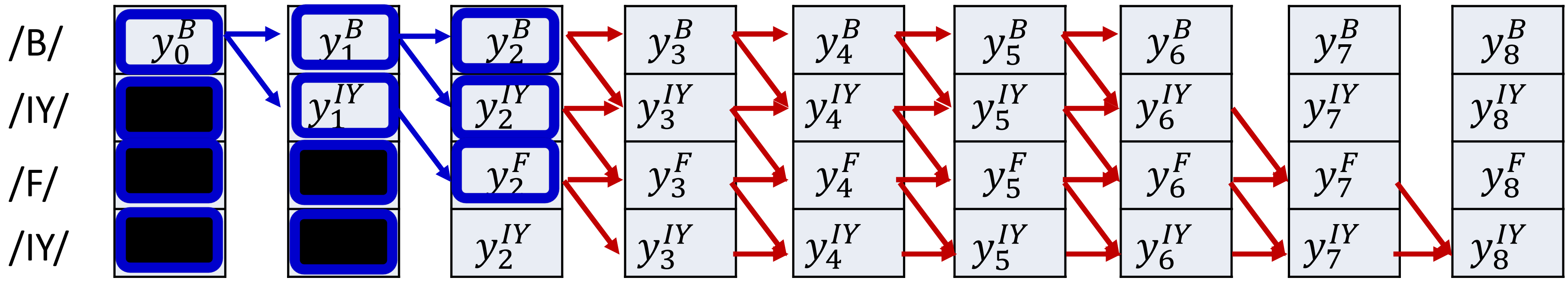
- for $t \ = \ 1 \ldots T-1$

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

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

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

Viterbi algorithm



- Initialization:

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

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

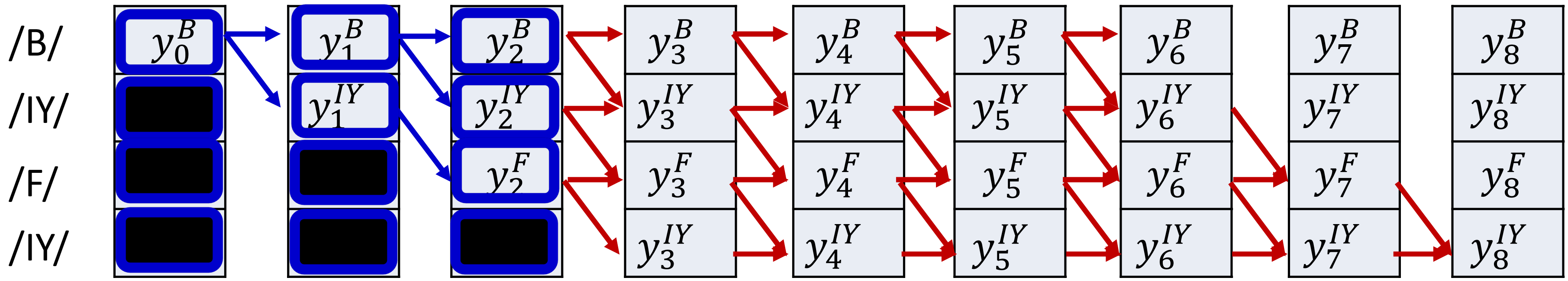
- for $t \ = \ 1 \dots T-1$

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

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

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

Viterbi algorithm



- Initialization:

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

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

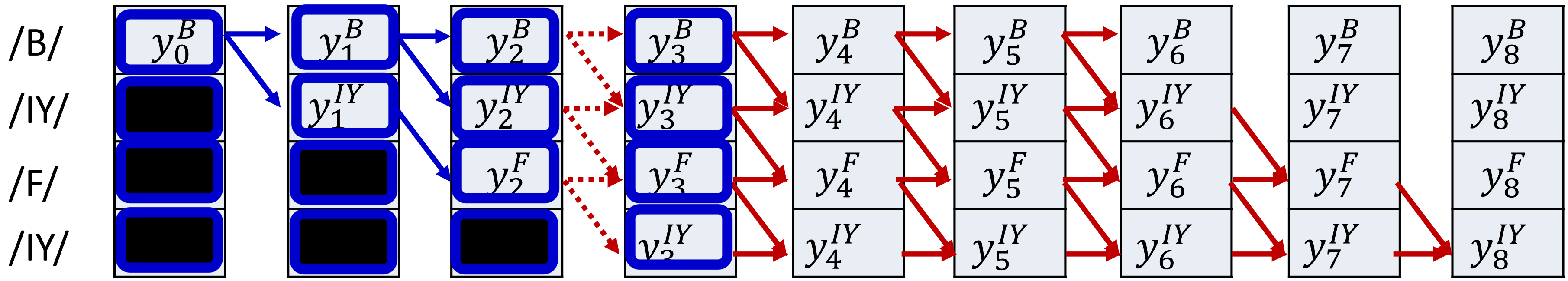
- for $t \; = \; 1 \ldots T-1$

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

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

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

Viterbi algorithm



- Initialization:

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

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

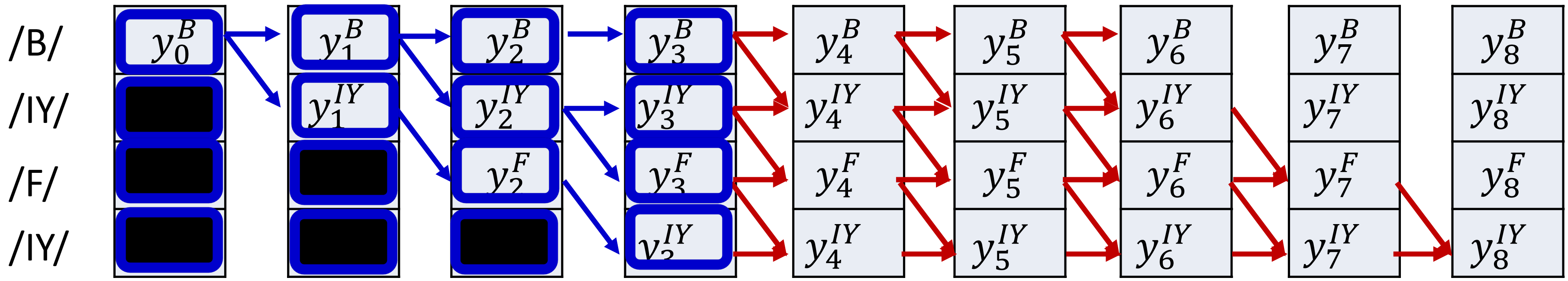
- for $t = 1 \ldots T-1$

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

for $l = 1 \ldots K-1$

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

Viterbi algorithm



- Initialization:

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

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

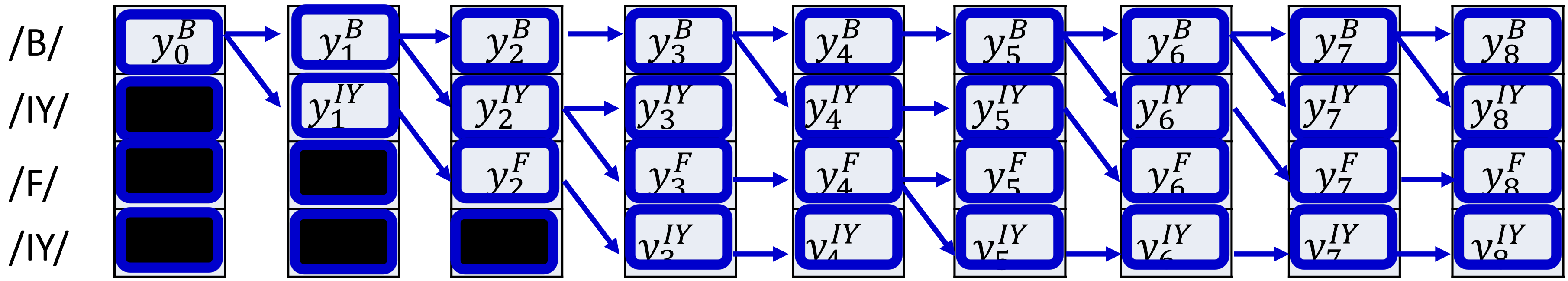
- for $t = 1 \ldots T-1$

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

for $l = 1 \ldots K-1$

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

Viterbi algorithm



- Initialization:

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

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

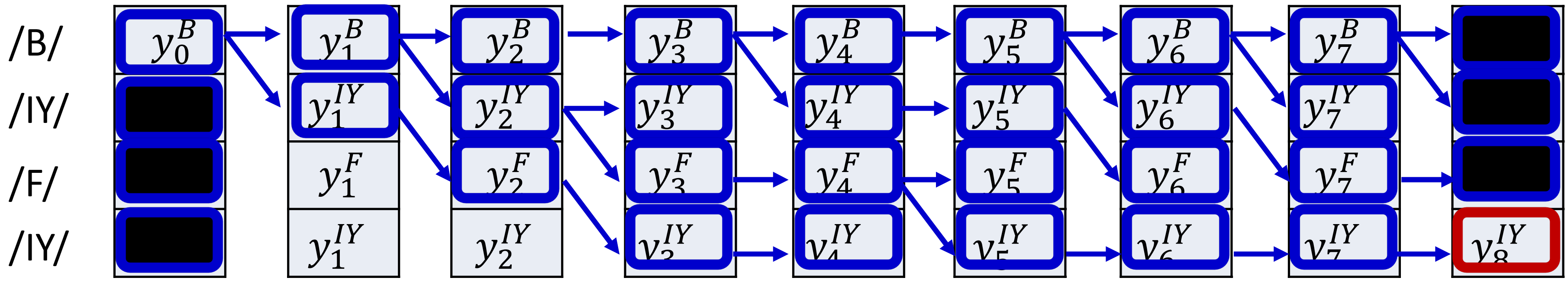
- for $t \ = \ 1 \ldots T-1$

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

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

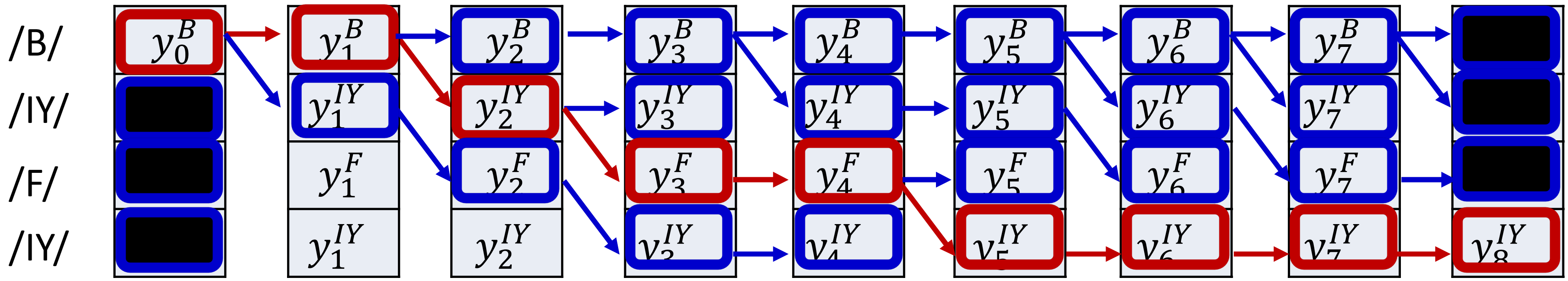
- $BP(t,l) = \begin{pmatrix} l-1 : \ \ if \ \left(Bscr(t-1,l-1) > \ Bscr(t-1,l)\right) \ l-1; \\ l : else \end{pmatrix}$
- $Bscr(t,l) \ = \ Bscr(BP(t,l)) \ \times y_t^{S(l)}$

Viterbi algorithm



- $s(T-1) = S(K-1)$

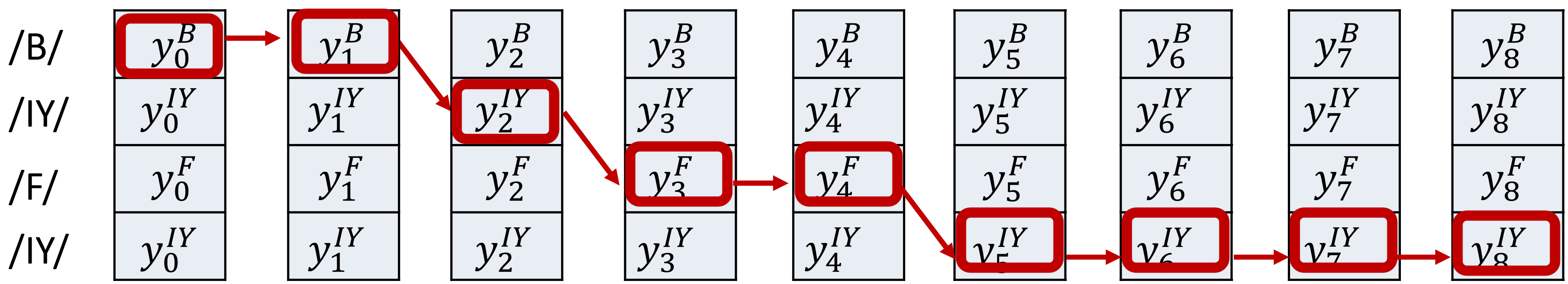
Viterbi algorithm



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

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

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

#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 = 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))*s(t,i)

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

Do not need explicit construction of output table

Information about order already in symbol sequence S(i), so we can use y(t,S(i)) instead of composing s(t,i) = y(t,S(i)) and using s(t,i)

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

VITERBI

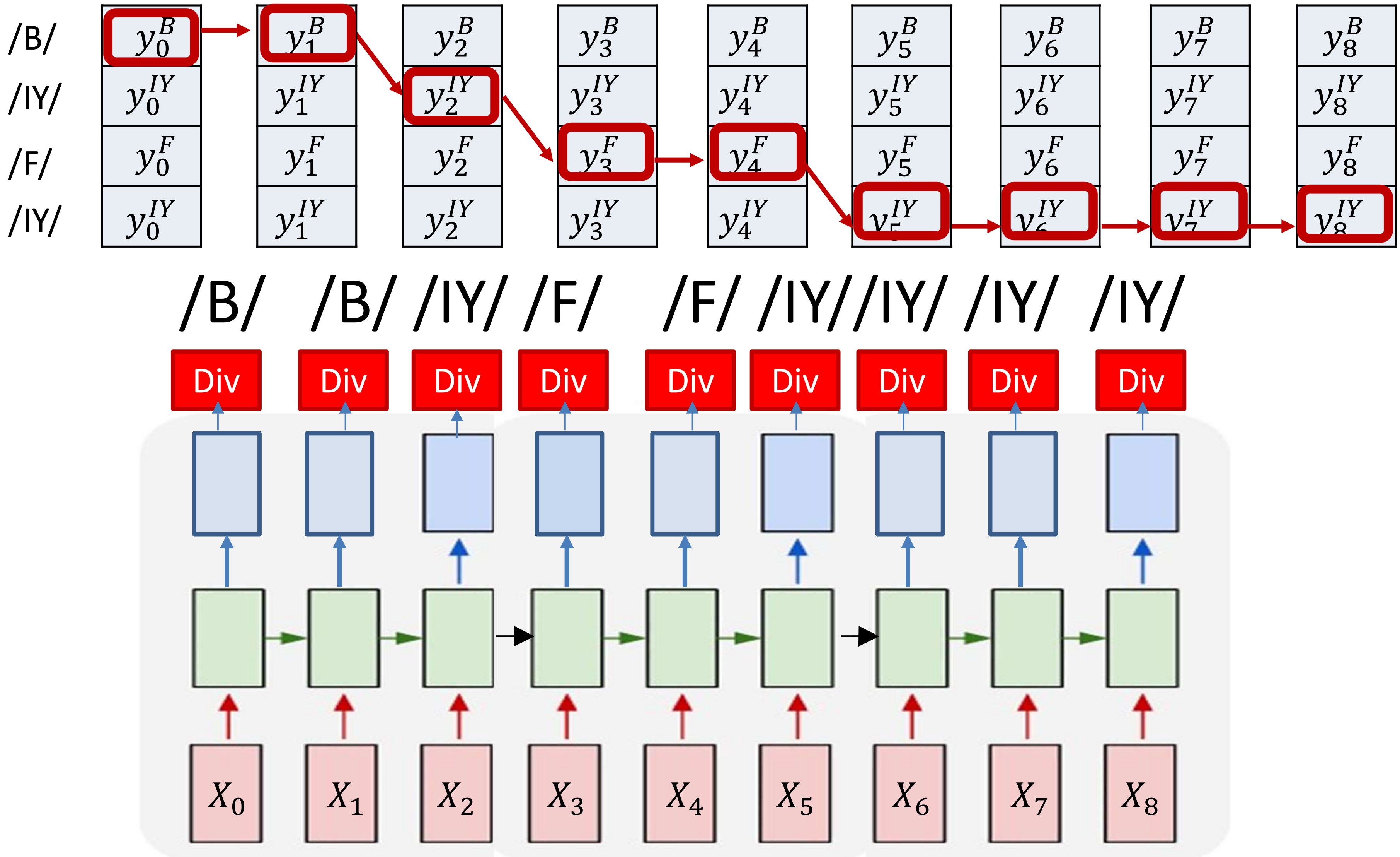
```
#N is the number of symbols in the target output
#S(i) is the ith symbol in target output
#T = length of input
```

Without explicit construction of output table

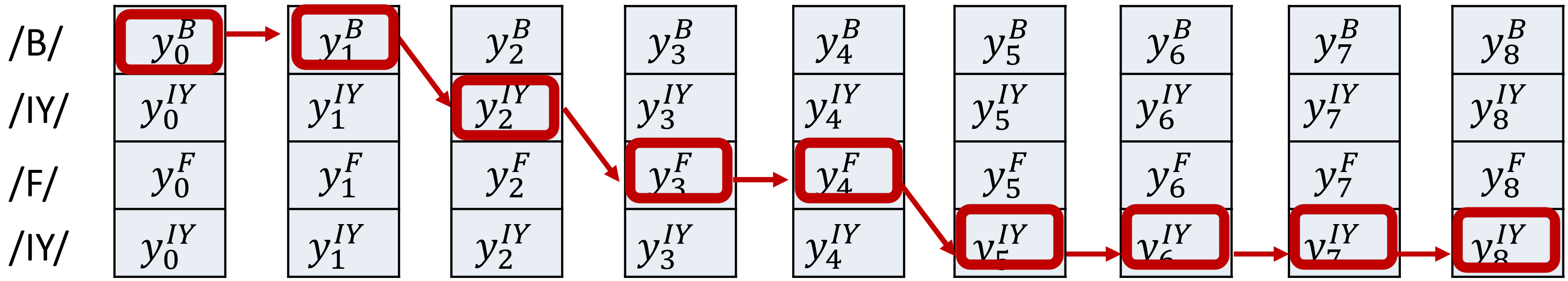
```
# First, at t = 1
BP(1,1) = -1
Bscr(1,1) = y(1,S(1))
Bscr(1,2:N) = -infty
for t = 2:T
    BP(t,1) = 1;
    Bscr(t,1) = Bscr(t-1,1)*y(t,S(1))
    for i = 2:min(t,N)
        BP(t,i) = Bscr(t-1,i) > Bscr(t-1,i-1) ? i : i-1
        Bscr(t,i) = Bscr(t-1,BP(t,i))*y(t,S(i))

# 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

Assumed targets for training with the Viterbi algorithm



Gradients from the alignment



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

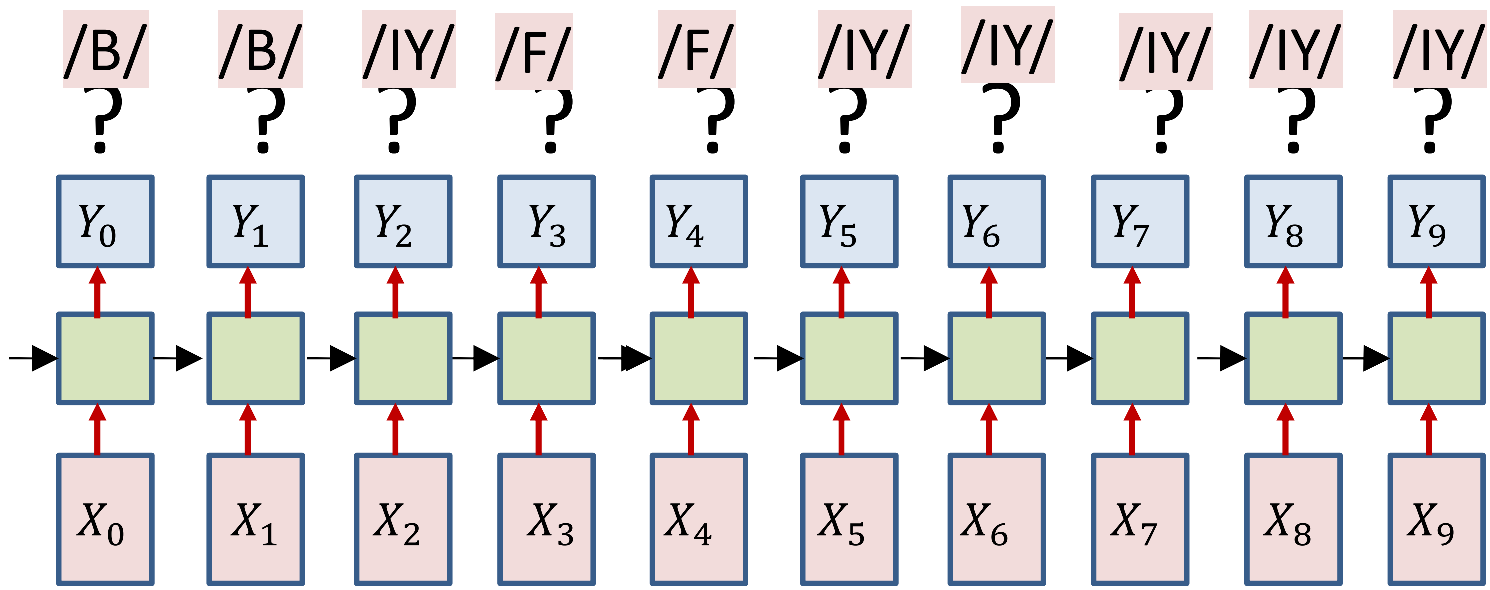
$$DIV = \sum_t KL\left(Y_t, symbol_t^{bestpath}\right) = -\sum_t \log Y\left(t, symbol_t^{bestpath}\right)$$

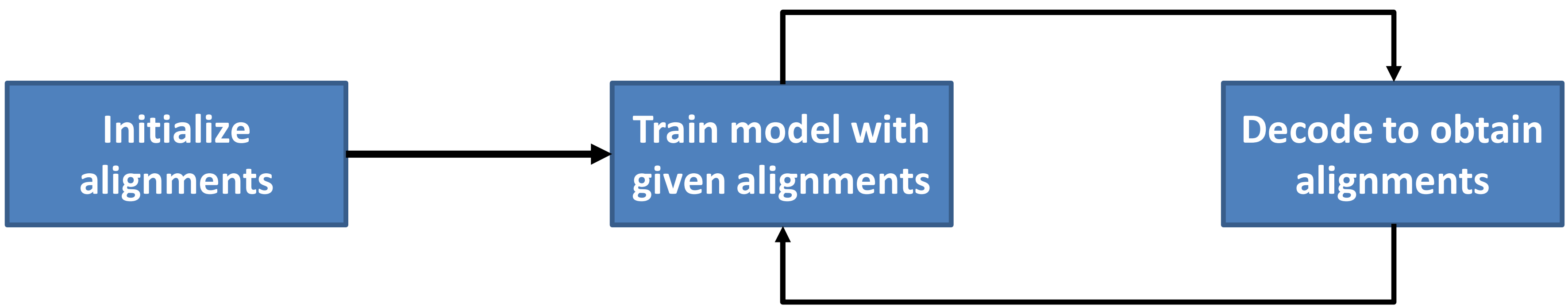
- The gradient w.r.t the t-th output vector Y_t

$$\nabla_{Y_t} DIV = \begin{bmatrix} 0 & 0 & \cdots & \dfrac{-1}{Y\left(t, symbol_t^{bestpath}\right)} & 0 & \cdots & 0 \end{bmatrix}$$

  – Zeros except at the component corresponding to the target *in the estimated alignment*

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

- Option 1:
  - Determine alignments for every training instance
  - Train model (using SGD or your favorite approach) on the entire training set
  - Iterate

- Option 2:
  - During SGD, for each training instance, find the alignment during the forward pass
  - Use in backward pass

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..

Next Class

- Training without explicit alignment..
  - Connectionist Temporal Classification
  - Separating repeated symbols
- The CTC decoder..