How to train transducer-based ASR systems

(Without memory bottlenecks)

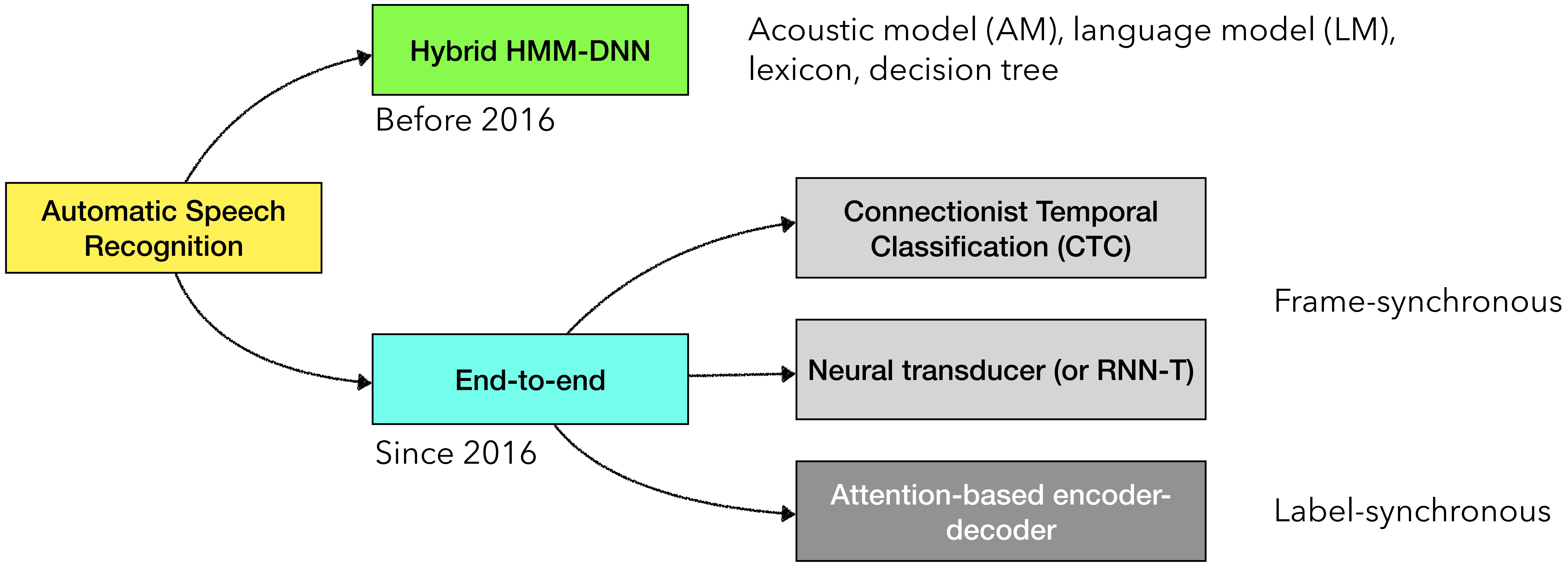
Desh Raj
Speech Technologies Reading Group
October 14, 2022

Overview

- Preliminary: Transducer-based ASR
- The problem of memory
- Method 1: Efficient implementation (Microsoft)
- Method 2: Alignment-restricted training (Meta)
- Method 3: Pruned transducer (k2)

Preliminary

All the major ASR approaches



Preliminary

Connectionist Temporal Classification (CTC)

- Given input speech $\mathbf{X}$, find best word sequence $\mathbf{Y}$
- Need to compute $P(\mathbf{Y} \mid \mathbf{X})$
- For training, loss is $-\log P(\mathbf{Y} \mid \mathbf{X})$
- For inference, $\hat{\mathbf{Y}} = \mathrm{argmax}_{\mathbf{Y}} P(\mathbf{Y} \mid \mathbf{X})$

- **Problem:** Do not know alignment between $\mathbf{X}$ and $\mathbf{Y}$

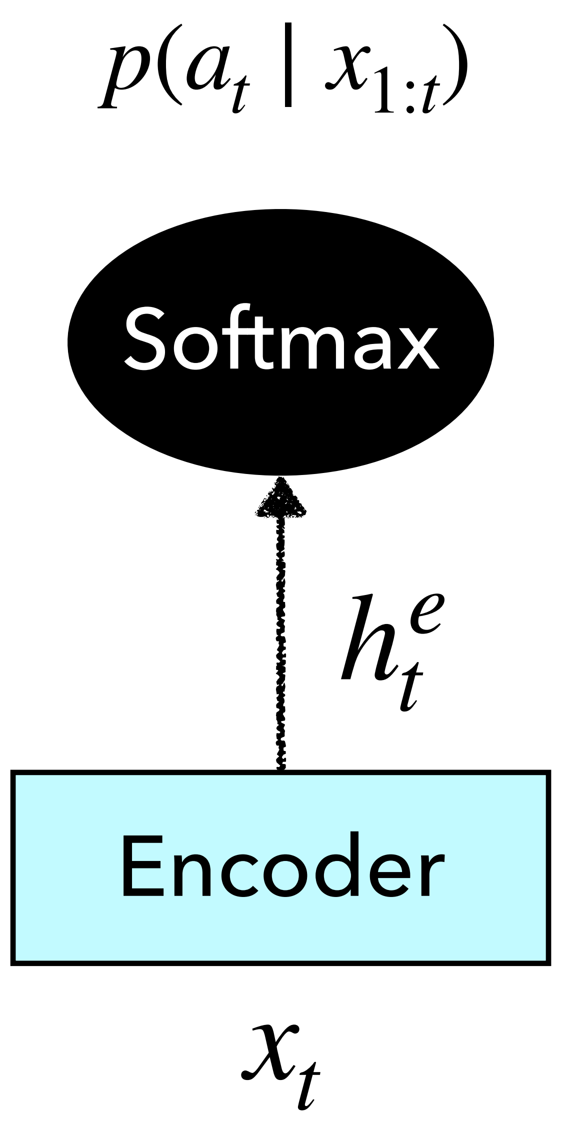
Preliminary

Connectionist Temporal Classification (CTC)

- **Problem:** Do not know alignment between $\mathbf{X}$ and $\mathbf{Y}$
- **Solution:** sum over all possible **alignments**

$$
\begin{aligned}
P(\mathbf{Y} \mid \mathbf{X}) &= \sum_{A \in \mathscr{A}_{\mathbf{Y}}^{T}} P(A, \mathbf{Y} \mid \mathbf{X}) = \sum_{A \in \mathscr{A}_{\mathbf{Y}}^{T}} P(\mathbf{Y} \mid A, \mathbf{X}) P(A \mid \mathbf{X}) \\
&= \sum_{A \in \mathscr{A}_{\mathbf{Y}}^{T}} P(\mathbf{Y} \mid A) P(A \mid \mathbf{X}) = \sum_{A \in \mathscr{A}_{\mathbf{Y}}^{T}} P(A \mid \mathbf{X}) \\
&= \sum_{A \in \mathscr{A}_{\mathbf{Y}}^{T}} \boxed{\prod_{t=1}^{T} P(a_t \mid \mathbf{X})}
\end{aligned}
$$

Conditional independence of outputs



Preliminary

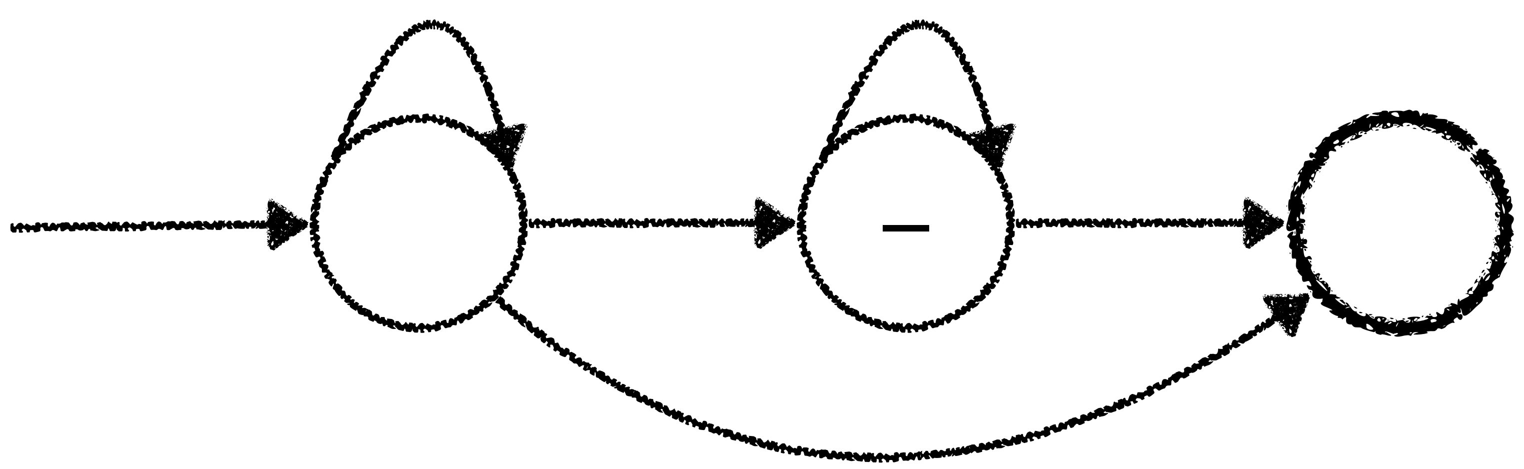
Connectionist Temporal Classification (CTC)

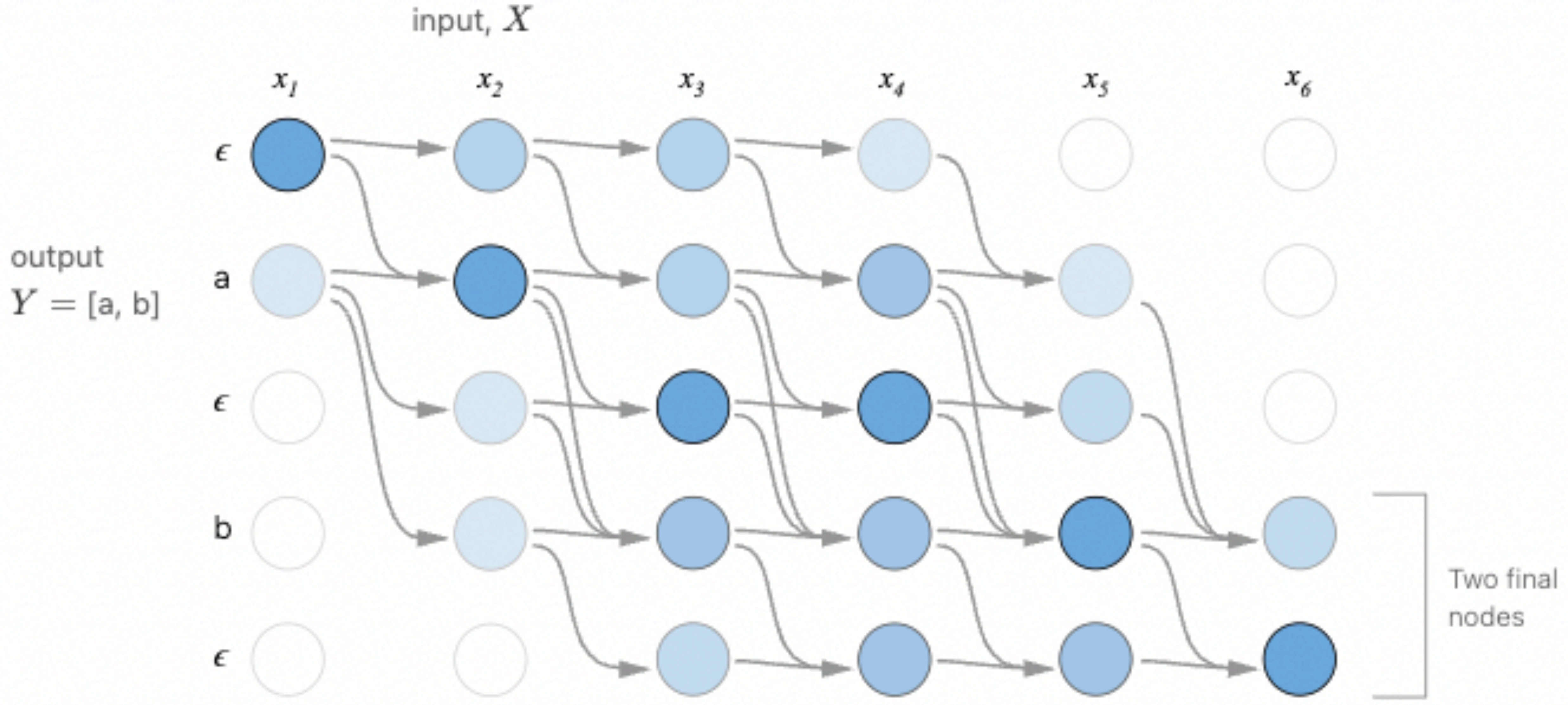
- What is an **alignment**?
- Example: $\mathbf{X}$ is of length 5, $\mathbf{Y}$ is CAT
- Alignments: CCAAT, ϵCATT, CAAϵT, etc.
- To get word from alignment, first collapse repetitions, then remove ϵ
- Now we only need a way to sum over all such alignments
- **Problem:** Exponentially many alignments

Preliminary

Connectionist Temporal Classification (CTC)

- **Problem:** Exponentially many alignments
- **Solution:** dynamic programming
- Similar to HMM forward algorithm





Node (s, t) in the diagram represents $\alpha_{s,t}$ – the CTC score of the subsequence $Z_{1:s}$ after t input steps.

https://distill.pub/2017/ctc/

Preliminary

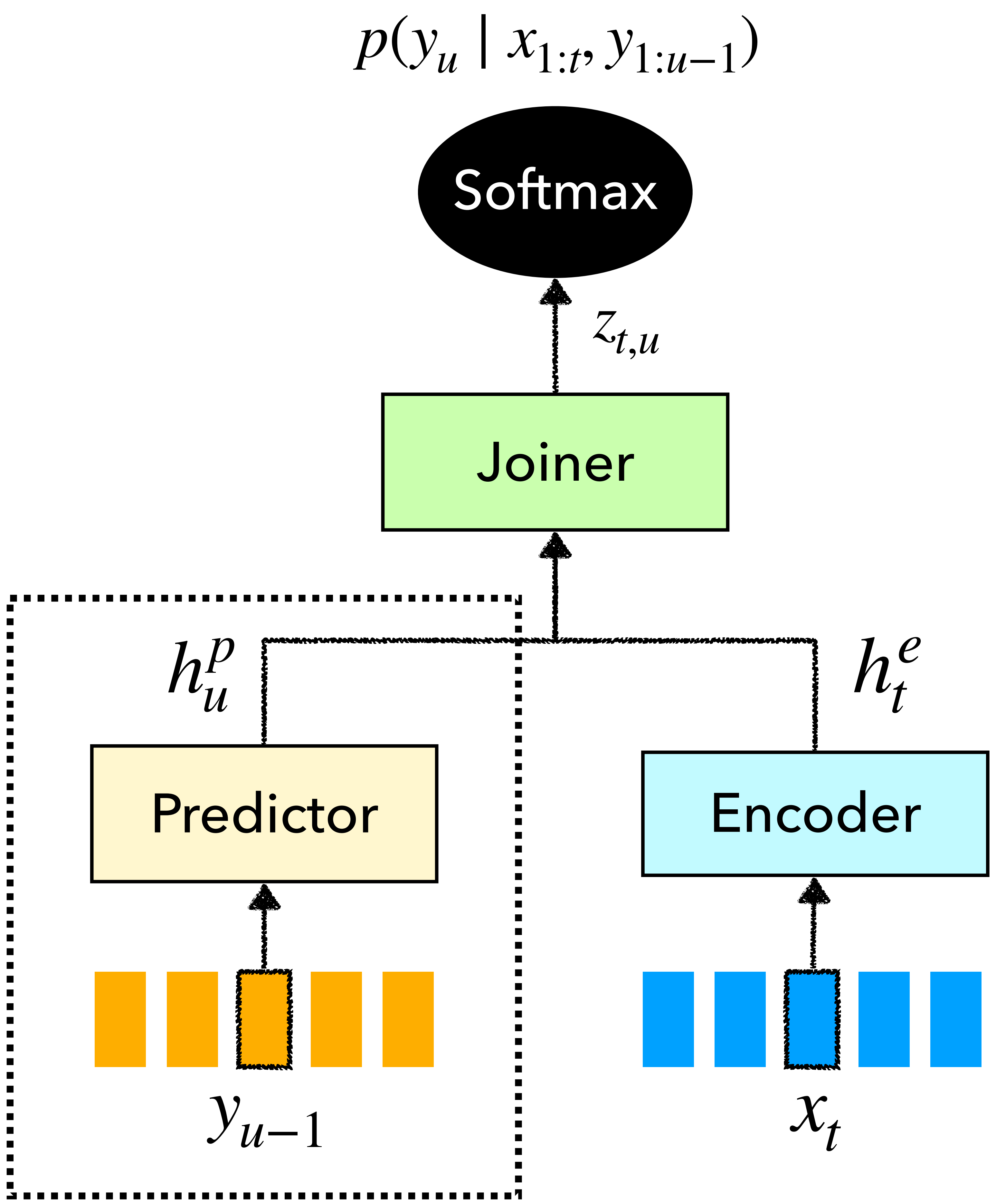
Problems with CTC

1. Conditional independence of outputs
2. Output sequence must be shorter than input sequence

RNN-Transducer

Solves both of the problems with CTC

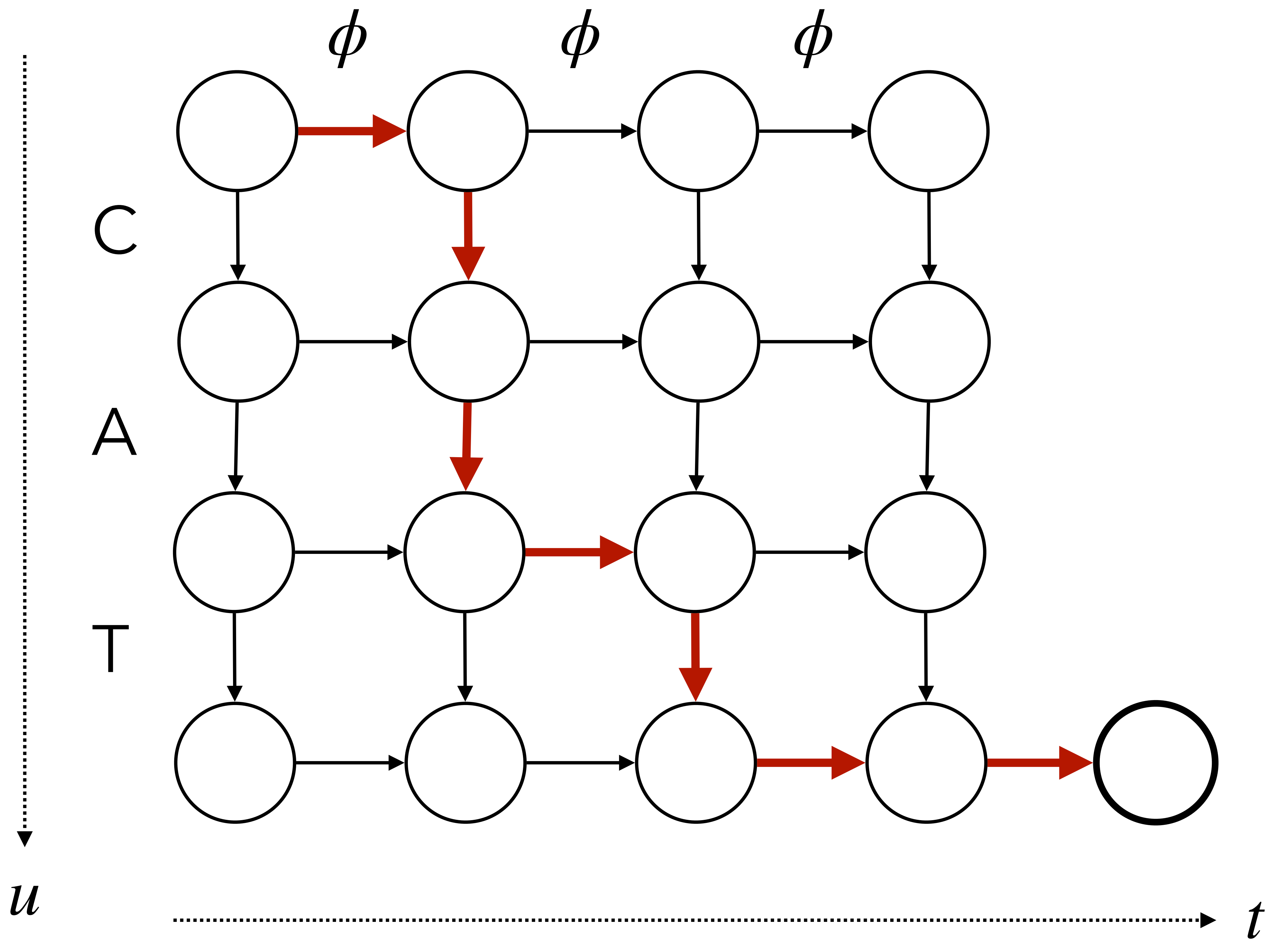
1. Conditional independence of outputs
   - *Use a predictor network (autoregressive model on previous outputs)*
2. Output sequence must be shorter than input sequence
   - *Allow multiple outputs at each time step*

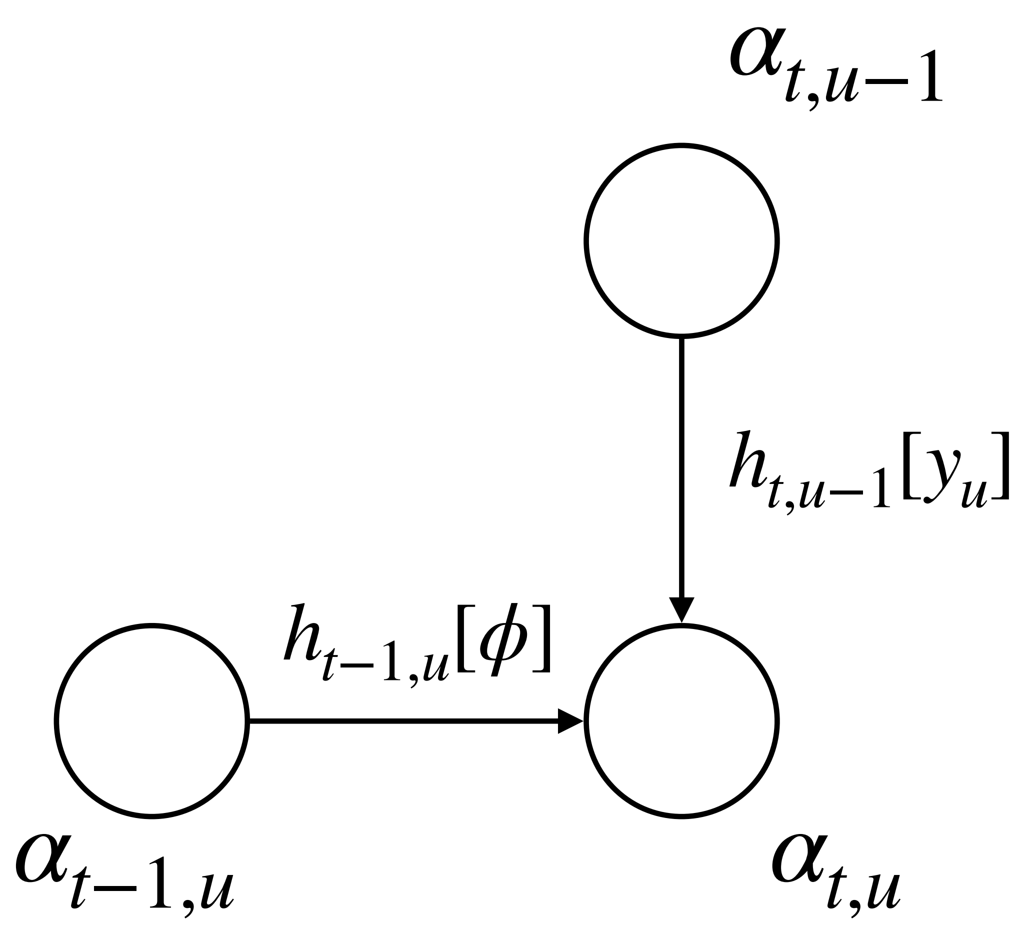


RNN-Transducer

Alignments

$$\alpha(t, u) = \alpha(t-1,u)h_{t-1,u}[\phi] + \alpha(t, u-1)h_{t,u-1}[y_u]$$





Forward algorithm

RNN-Transducer

The memory problem

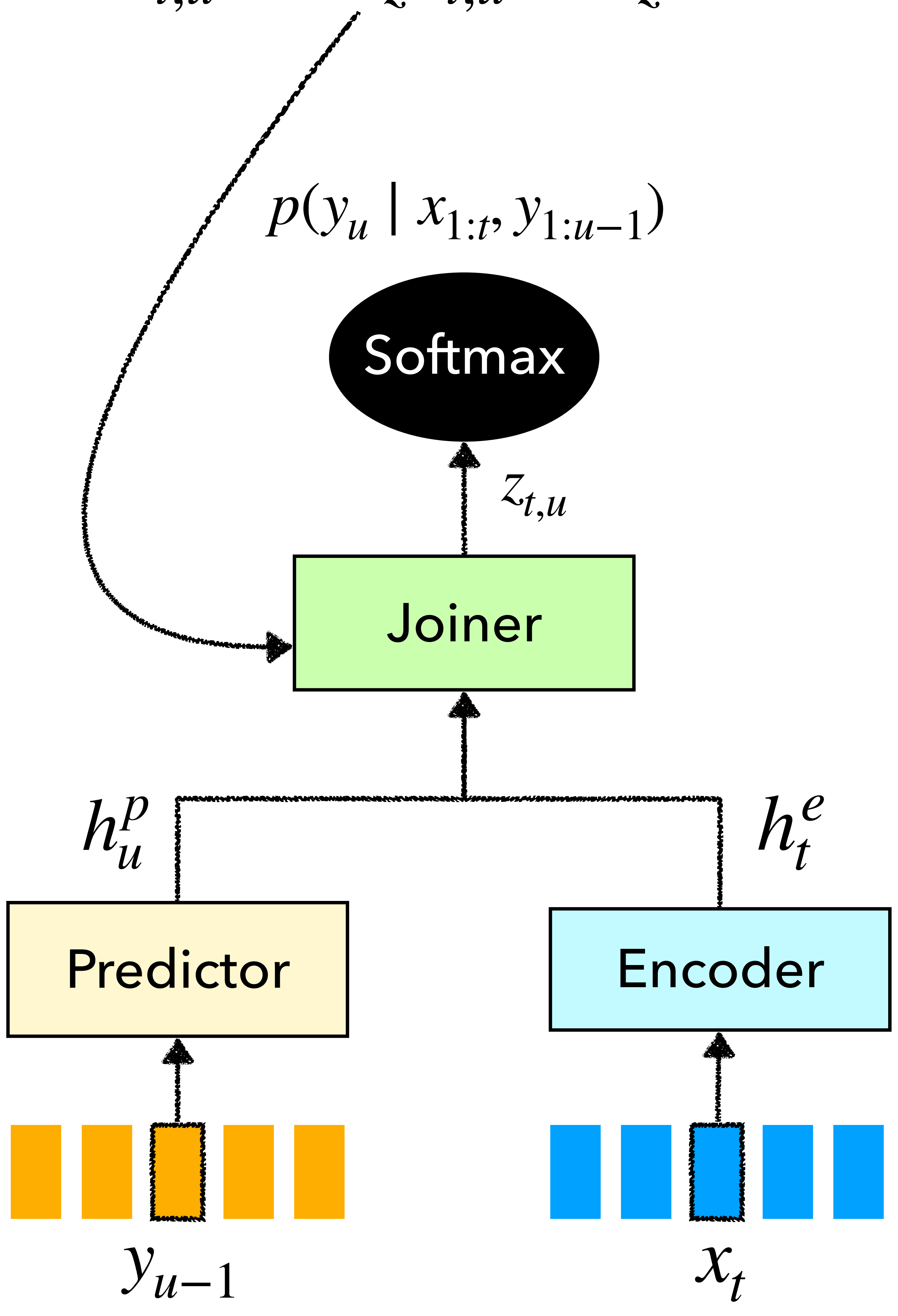
- Suppose B is batch size, T is number of frames in input (padded), U is output sequence length (padded), D is output dimension (equal to vocabulary size + 1), F is hidden layer dimension.

- h^e has shape (B, T, F); h^p has shape (B, U, F)

- To combine h^e and h^p, we will use broadcasting, resulting in 4-D tensor of size (B, T, U, D).

$$h_{t,u} = \psi(W_E h_t^e + W_P h_u^p + b_z)$$

$$z_{t,u} = W_z h_{t,u} + b_z$$

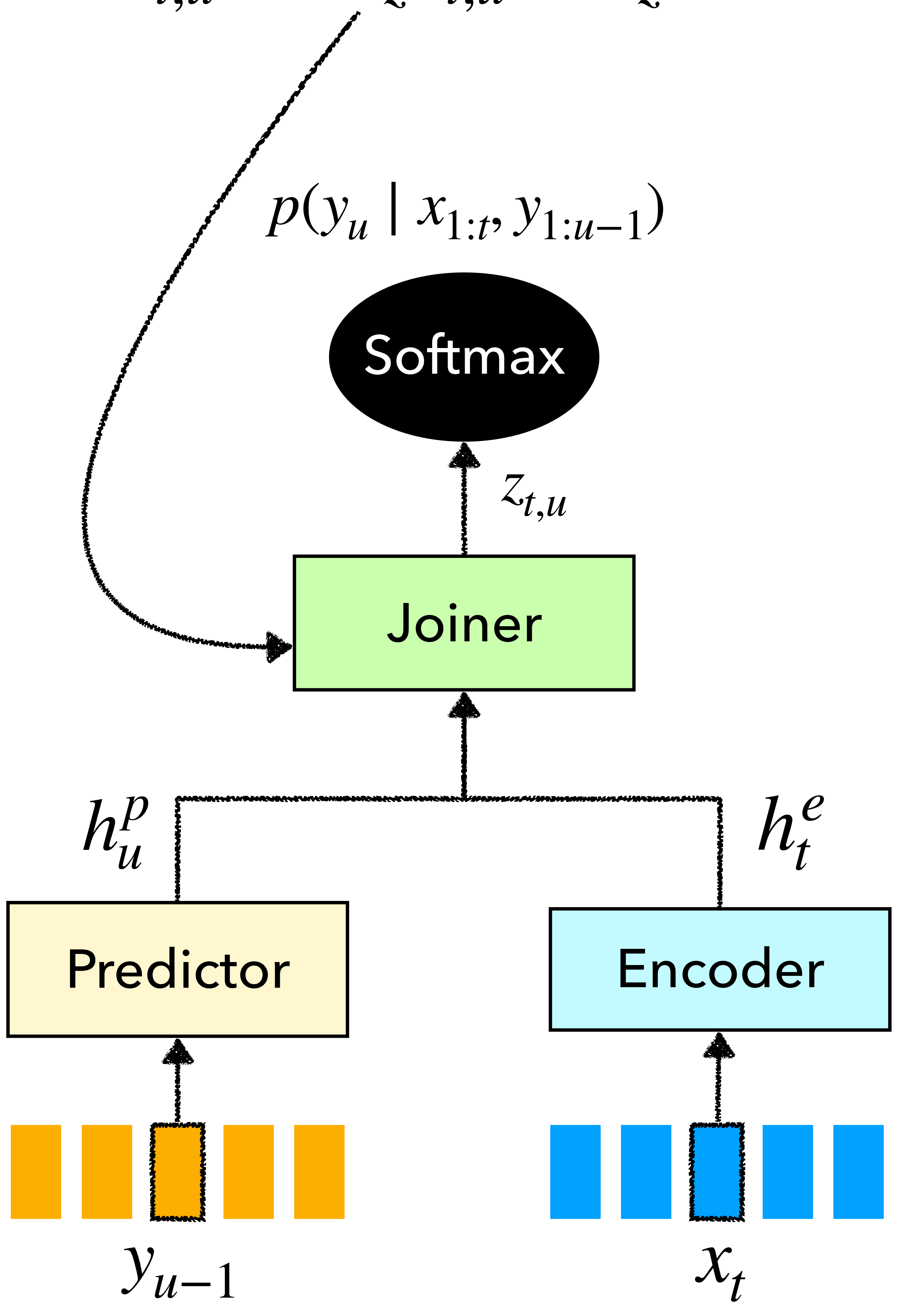


RNN-Transducer

The memory problem

$$h_{t,u} = \psi(W_E h_t^e + W_P h_u^p + b_z)$$

$$z_{t,u} = W_z h_{t,u} + b_z$$



- To combine h^e and h^p, we will use broadcasting, resulting in 4-D tensor of size (B, T, U, D).
- For a simple case, B=32, T=1000 (10 seconds), U=100 (~5 words/sec), D=1000, this equal 3.2 x 10^9, or 12.8 GB with single-precision floats.
- And this is just for storing the logits.

https://lorenlugosch.github.io/posts/2020/11/transducer/

Method 1: Efficient Implementation

Problem (a): Logit tensor contains lot of padding tokens

- z has shape (B, T, U, D), but a lot of these elements are just padding.
- Can we efficiently create this tensor?
- Naive method: sort sequences by length in training to reduce padding?
- Results in worse accuracy compared to randomized mini-batches

Method 1: Efficient Implementation

Idea (a): concatenate instead of broadcast

- For each batch element z_b of shape (T_b, U_b, D) – convert into 2-D tensor of shape $(T_b \times U_b, D)$.
- Concatenate all such elements along the axis 0
- Results in 2D tensor of shape $(\sum_{b \in B} T_b \times U_b, D)$
- This tensor has NO padding tokens!

Method 1: Efficient Implementation

Problem (b): Need to store 3 large tensors

- Standard modular implementation: logits -> softmax -> RNN-T loss.
- For getting the derivative, we need to store 3 large tensors (chain rule):
  - Derivative of loss w.r.t softmax output
  - Softmax output tensor
  - Derivative of softmax output w.r.t logits

Method 1: Efficient Implementation

Idea (b): function merging

- Directly pass logits to RNN-T loss and compute derivatives without going through softmax.
- Only need to store 1 large tensor instead of 3!

$$\frac{\partial L}{\partial z_{t,u}^{k}} = \frac{P(k \mid t,u)\alpha(t,u)}{P(\mathbf{y} \mid \mathbf{x})} \left[\beta(t,u) - \begin{cases} \beta(t,u+1) & \text{if } k = y_{u+1} \\ \beta(t+1,u) & \text{if } k = \varnothing \\ 0 & \text{otherwise} \end{cases} \right]$$

Method 1: Efficient Implementation

Benchmarking

- Originally proposed by Microsoft [1]

[1] J. Li, R. Zhao, H. Hu, and Y. Gong, "Improving RNN Transducer Modeling for End-to-End Speech Recognition." IEEE ASRU 2019.

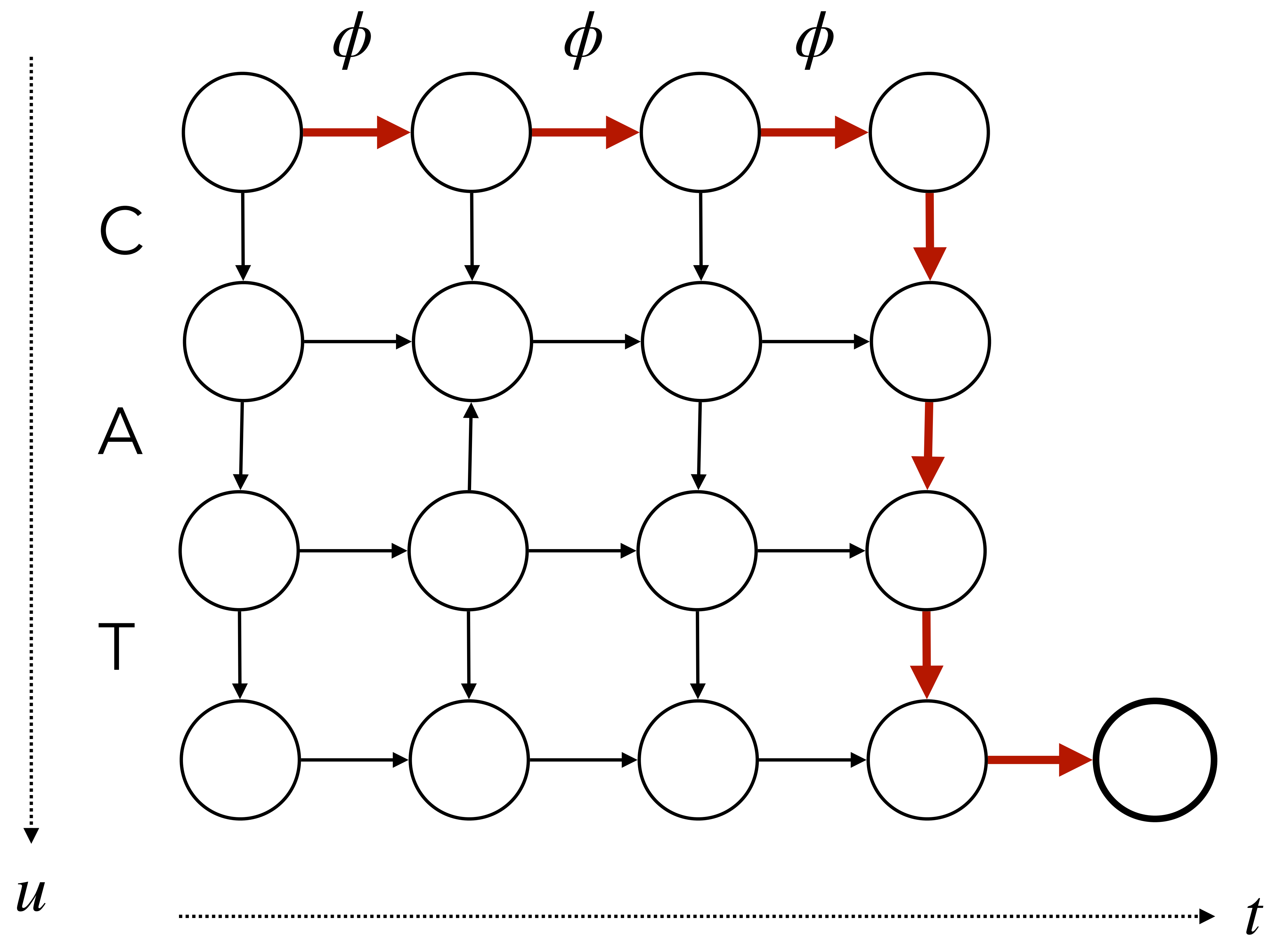
- Open source re-implementation available here: https://github.com/csukuangfj/optimized_transducer

- Benchmark (from https://github.com/csukuangfj/transducer-loss-benchmarking):

Method	Avg. step time (us)	Peak memory (MB)
warp_transducer (ESPNet)	275852	19072.6
optimized_transducer	376954	7495.9

Method 2: Alignment-restricted training

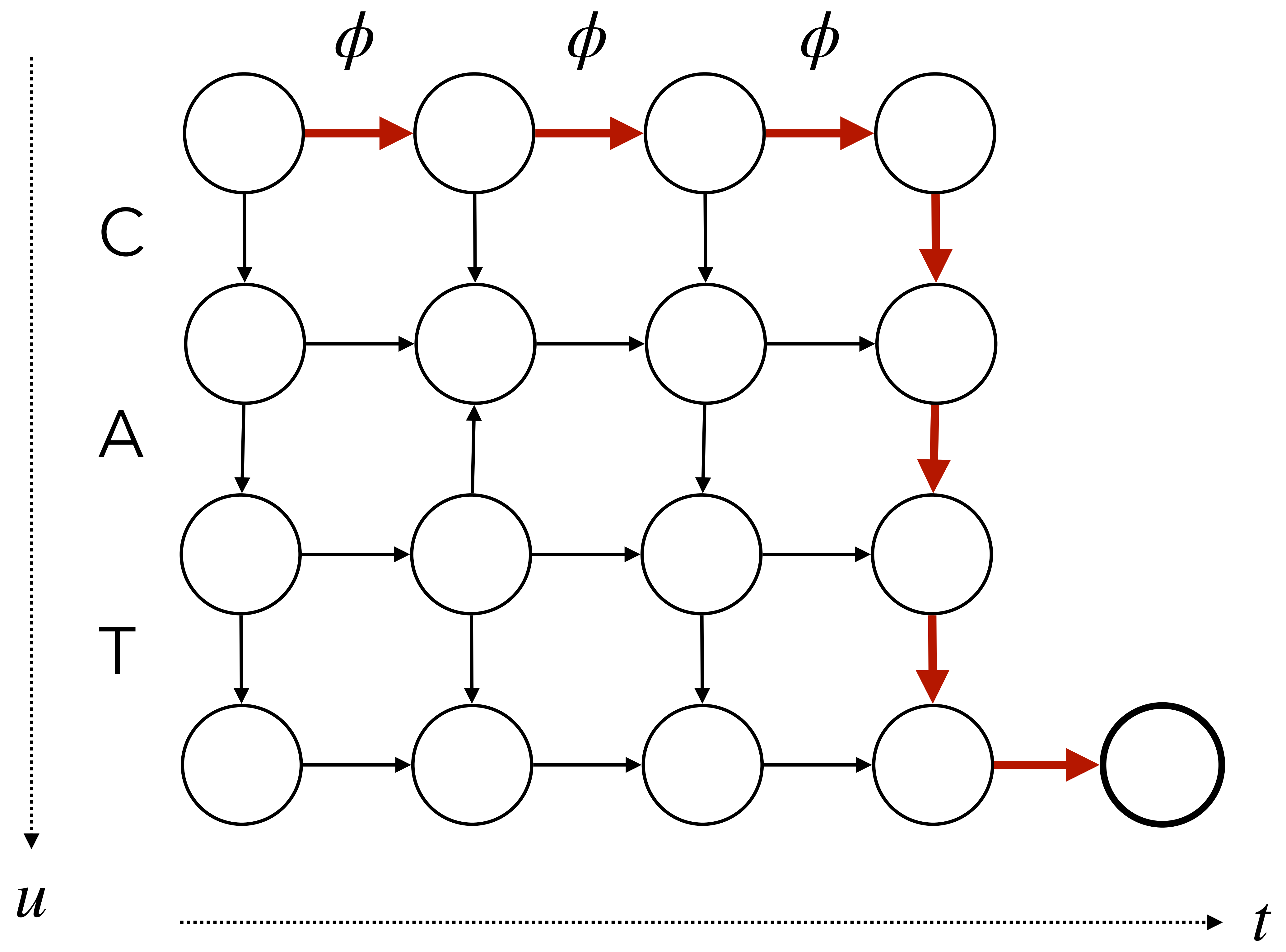
Alignment-free training can cause token emission delays



- Remember RNN-T lattice?
- Since RNN-T performs alignment-free training (sum over all alignments), it could very well learn to wait until the last time-step to output all tokens.
- This can cause token emission delay, which is bad for streaming ASR.

Method 2: Alignment-restricted training

Key idea: enforce alignment between input and output

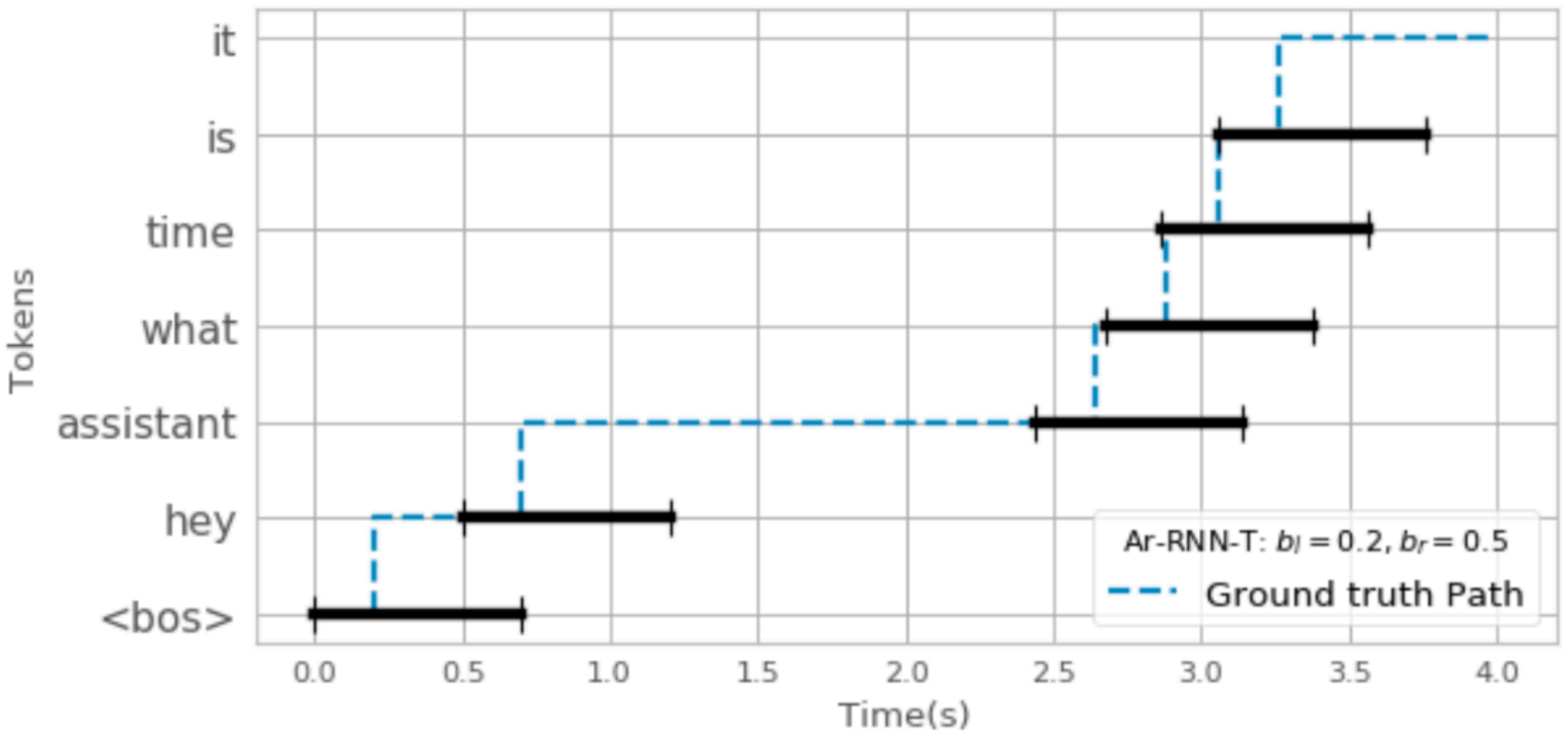


- If we roughly know that a token should be emitted at a particular time-step (+- some buffer), we can enforce this in training.
- This means that instead of summing over "all possible alignments", we are summing over a restricted set of alignments.

[2] J. Mahadeokar *et al*., "Alignment Restricted Streaming Recurrent Neural Network Transducer." IEEE SLT 2020.

Method 2: Alignment-restricted training

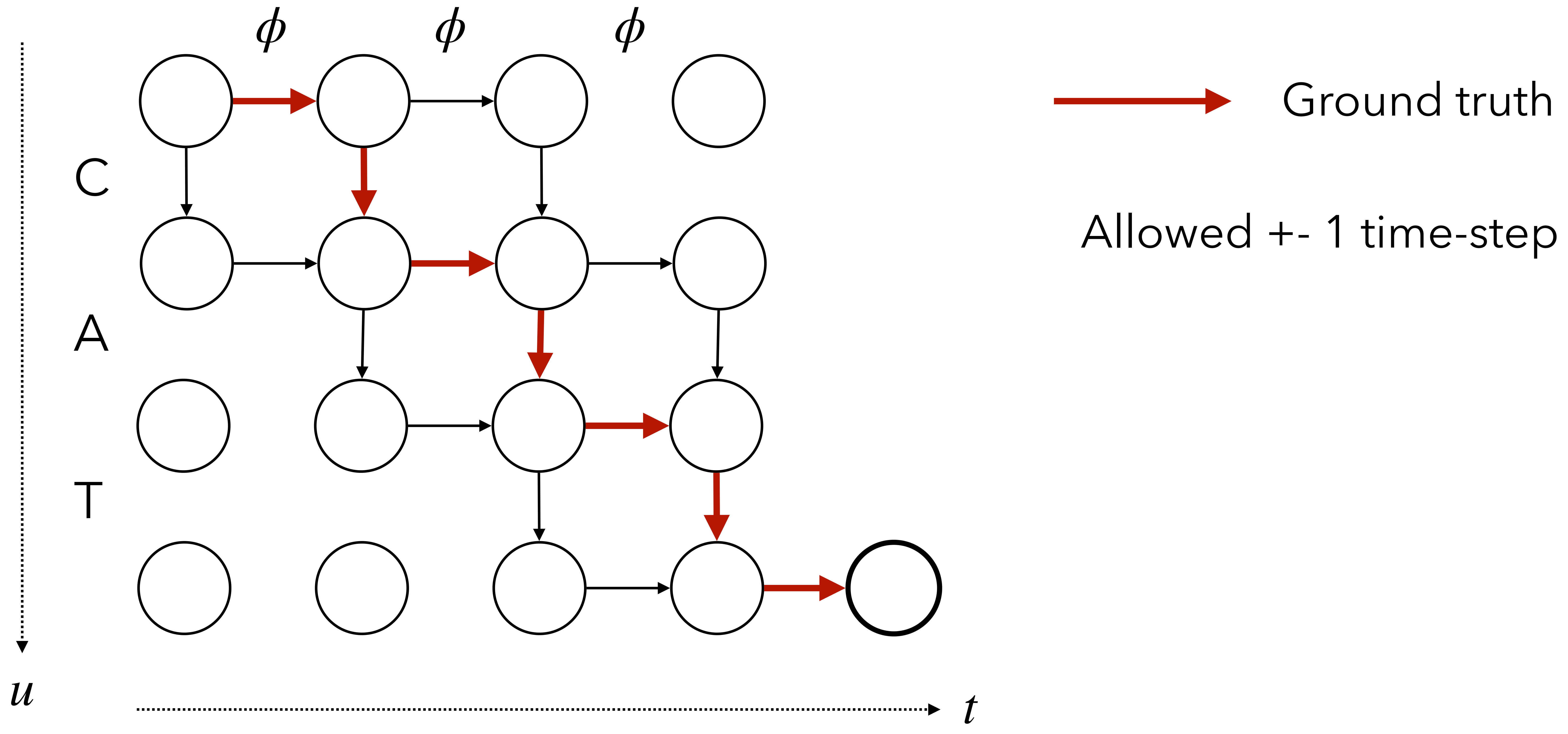
Key idea: enforce alignment between input and output



We can get the external alignment from an HMM-based aligner.

Method 2: Alignment-restricted training

Simple example



Method 2: Alignment-restricted training

How is it implemented in loss computation?

Standard RNN-T

$$\alpha(t, u) = \alpha(t-1, u) h_{t-1,u}[\phi] + \alpha(t, u-1) h_{t,u-1}[y_u]$$

AR-RNNT

$$\bar{h}_{t,u-1}[y_u] = \begin{cases} h_{t,u-1}[y_u], & \text{if } \; v_{lu} \leq t \leq v_{ru} \\ 0, & \text{otherwise} \end{cases}$$

$$v_{lu} = a_u - b_l$$
$$v_{ru} = a_u + b_r$$

And likewise for backward computation (to get gradients)

Method 2: Alignment-restricted training

Some details

- External alignments are obtained using a hybrid model trained with cross-entropy loss.
- These alignments are at word-level, but RNN-T output is at subword-level. How to reconcile?
- Evenly split the time between the word-pieces.

Method 2: Alignment-restricted training

But how does it help in saving memory?

- Pre-compute the valid time ranges (v_{lu}, v_{ru}) for every y_u
- Similar to Method 1, concatenate sequences into a 2D tensor, only keeping the time-steps that fall in the valid ranges.
- Results in 2D tensor of shape $(\sum_{b \in B} (b_l + b_r) \times U_b, D)$
- Allows training with **4x** the batch size!

Method 2: Alignment-restricted training

Limitation

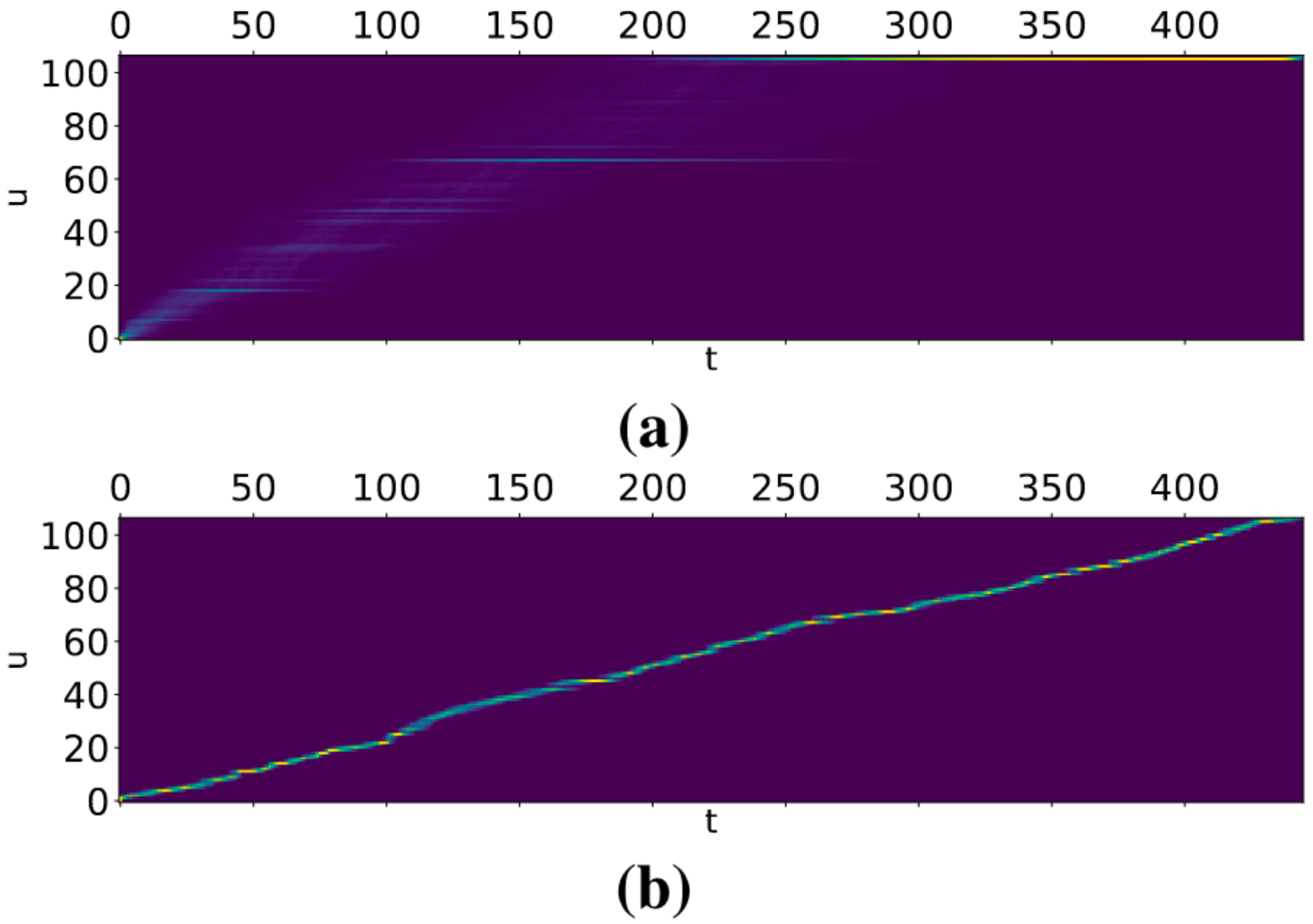
- The major limitation here is that we need to train a hybrid ASR system to obtain the external alignments.

Method 3: Pruned RNN-T

[3] F. Kuang *et al*., "Pruned RNN-T for fast, memory-efficient ASR training." InterSpeech 2022.

Very similar to AR-RNNT, but no need for alignments

- Consider the following figure showing node gradients during RNN-T training at the beginning and after model has been trained for a while:



1. At each time-step, only small range of nodes have non-zero gradient.

2. Position of nodes with non-zero gradient changes monotonically.

Method 3: Pruned RNN-T

Key idea: Restrict U for each time step

- At each time step, we can restrict U to a small set S.
- Logit tensor becomes (B, T, S, D), where S is a small number like 5.
- If U = 100, this means a 20x memory saving.

- **Question:** How do we generate the set S for each time-step?

Method 3: Pruned RNN-T

Solution: Use a "simple" joiner to approximate S

- A 2-step process is used to compute the final loss.
- Recall original joiner:

$$h_{t,u} = \psi(W_E h_t^e + W_P h_u^p + b_z)$$
$$z_{t,u} = W_z h_{t,u} + b_z$$

- We don't want to compute this "full" joiner for all U tokens.

Method 3: Pruned RNN-T

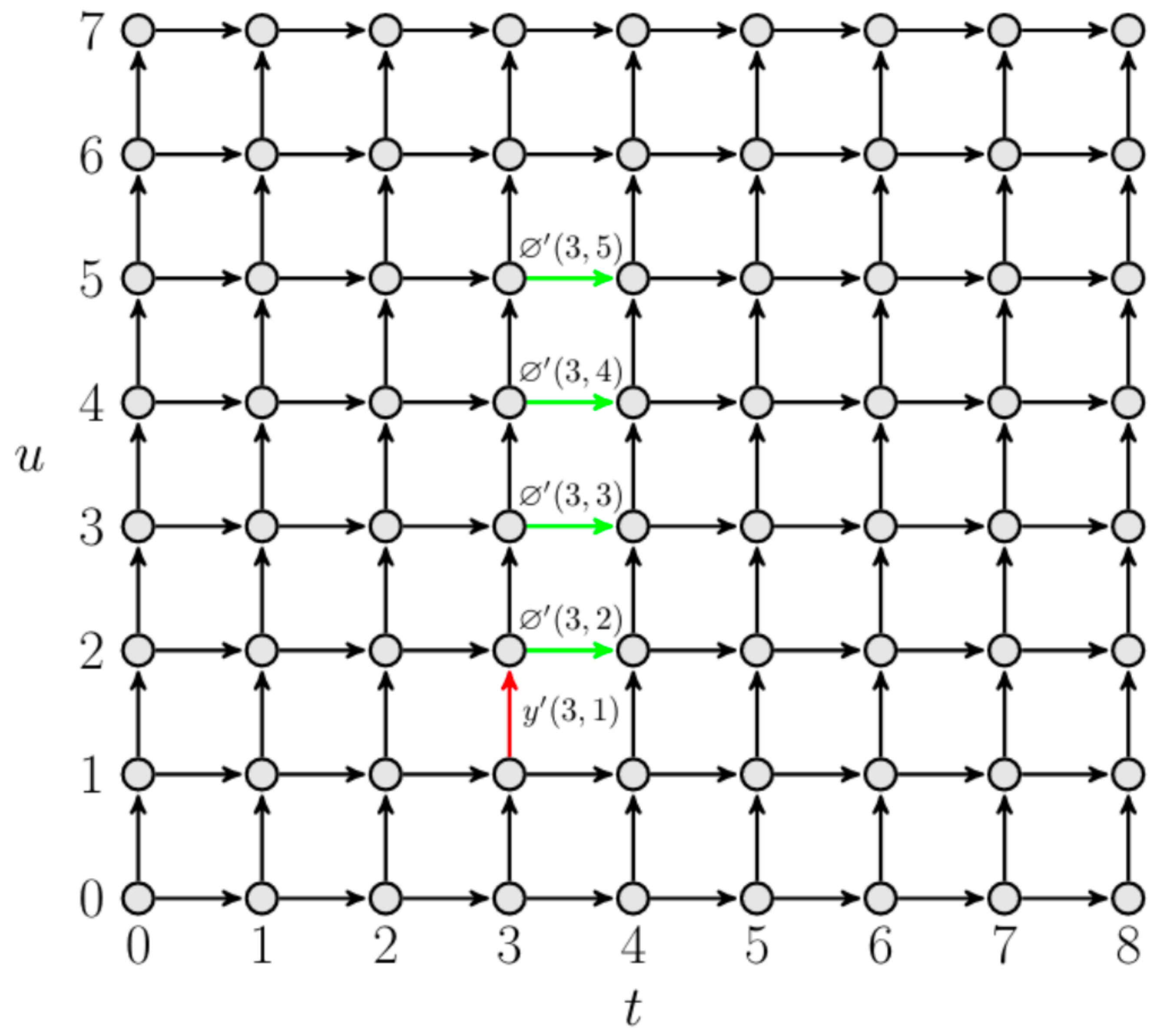
Solution: Use a "simple" joiner to approximate S

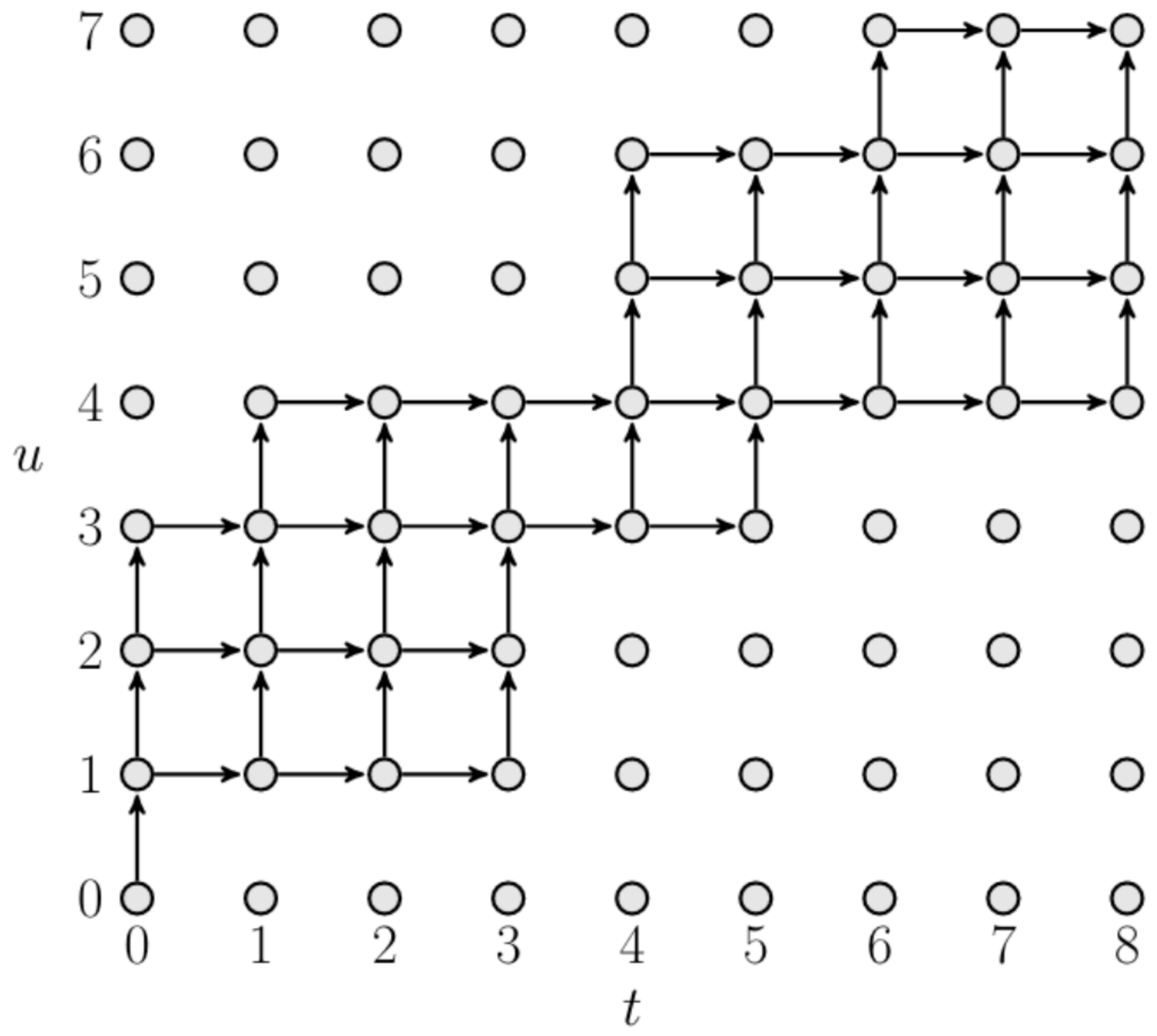
- "Simple" joiner: project h^e and h^p to dimension D, add them (treating as log-probabilities, and then normalize.
- $\alpha(t, u)[v] = h_t^e[v] + h_u^p[v] - h_{\text{norm}}(t, u)$
- $h_{\text{norm}}(t, u)$ can be computed using LogSumExp trick
- In this way, we avoid computing the large tensor (B, T, U, D)

Method 3: Pruned RNN-T

Obtaining pruning bounds using the simple joiner

- We want to use the $\alpha(t, u)$ computed using the "simple" joiner to prune the RNN-T lattice.





Method 3: Pruned RNN-T

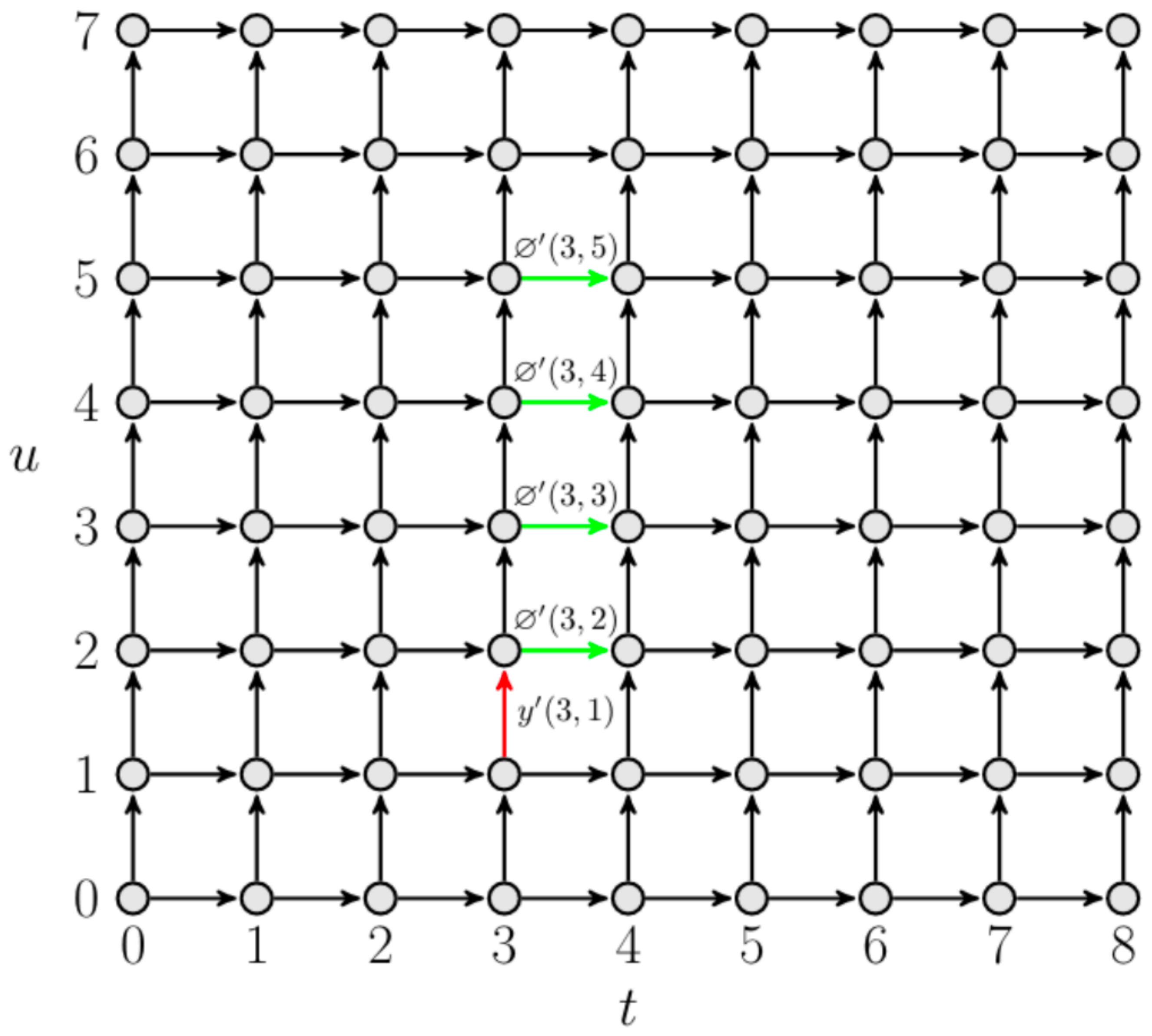
Obtaining pruning bounds using the simple joiner

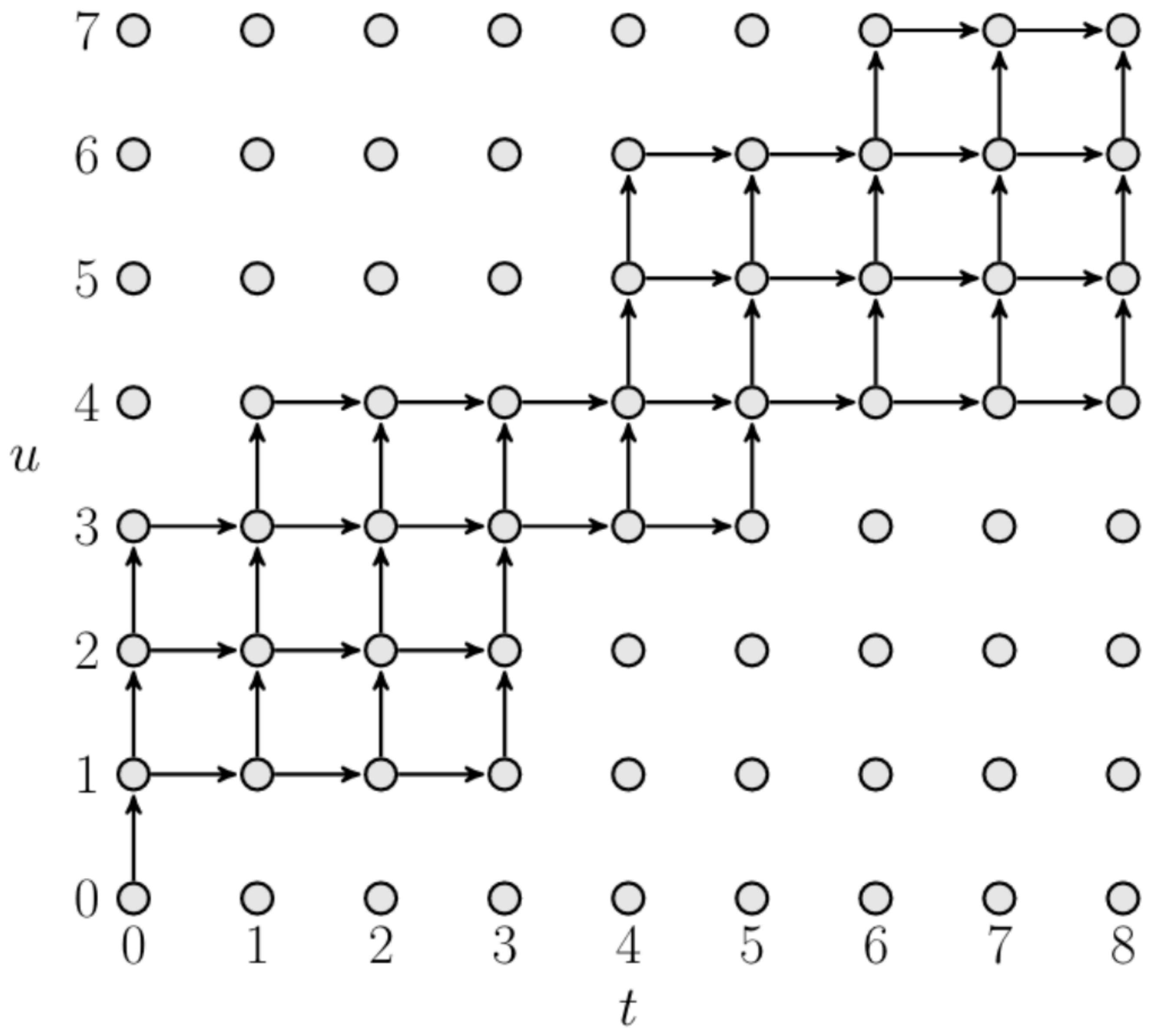
- Compute the derivatives $y'(t,u)$ and $\phi'(t,u)$ w.r.t the simple joiner loss.
- These can be interpreted as the "occupation counts" for taking the upward and rightward transitions.

Method 3: Pruned RNN-T

Obtaining pruning bounds using the simple joiner

- Suppose $S = 4$ and $p_t = 2$, for $t = 3$.
- This means that we will retain u = {2,3,4,5}





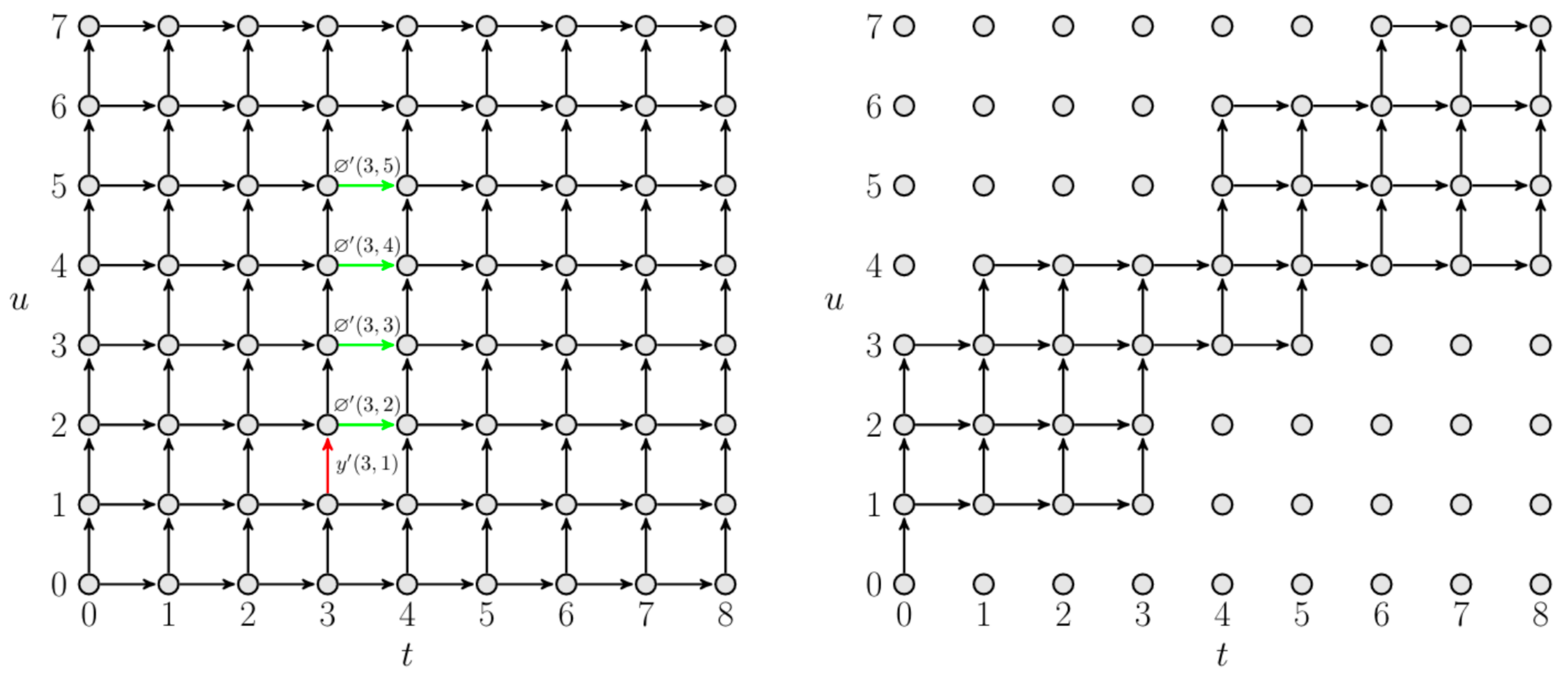
Method 3: Pruned RNN-T

Obtaining pruning bounds using the simple joiner

- Minimum retained log-prob for this choice of p_t

$$\phi'(t,2) + \phi'(t,3) + \phi'(t,4) + \phi'(t,5) - y'(t,1)$$

- Compute argmax for all values of p_t
- Adjust the bounds to ensure continuity. For example, consecutive time-steps should be monotonically increasing.



Method 3: Pruned RNN-T

Benchmarking

- Implemented in k2: https://github.com/k2-fsa/k2
- Benchmark (from https://github.com/csukuangfj/transducer-loss-benchmarking):

Table 1: *Speed and memory usage for different RNN-T loss implementations using fixed batch size 30.*

	Average time per batch (ms)	Peak memory usage (GB)
torchaudio	544	18.48
optimized_transducer	377	7.32
warp-transducer	276	18.63
SpeechBrain	459	18.63
pruned RNN-T	**64**	**3.73**

Standard RNN-T using modular PyTorch

Efficient 2-D tensor using CUDA

Standard RNN-T using CUDA

Standard RNN-T using Numba

Pruned RNN-T using CUDA

AR-RNNT vs. Pruned RNN-T

- Basically, we are using the “simple” joiner output to get an **approximate alignment** between the encoder output and the prediction network output (instead of an externally provided alignment as in AR-RNNT).
- We then use this approximate alignment to prune the lattice.
- Overall, AR-RNNT and pruned RNNT are the same idea but implemented differently.

Key take-aways

- Transducers are most popular for ASR in industry
- But they require **large memory** (due to BxTxUxD tensor)
- Efficiently storing the logits by **removing padding** saves cost
- We can also leverage the fact that there is an obvious alignment between encoder and decoder to **prune the lattice**
  - This **alignment** can either be obtained from a hybrid system (AR-RNNT)
  - Or computed using a **simple joiner** in first pass (pruned RNNT)