Applied Deep Learning

Chapter 6: Recurrent NNs

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Fall 2025

Computing Loss: *Challenge*

We mentioned several times in this chapter that *we assume*

we can compute the loss between RNN's output sequence and label sequence

However, it is in general a challenge!

+ *Why is it a challenge? We did it easily in FNN and CNN chapters!*
– Because the problem there *was already properly segmented!*
+ *What do you mean by segmented?*
– Let's break it down!

Computing Loss: *Motivating Example*

Let's consider a simple example: *we have an image that includes a sequence of handwritten digits, e.g.,*

- *The sequence includes five digits*
- *Each digit is either 1, 2, 3, or 4*

Our task is to recognize this sequence, i.e., return the five digits in correct order

- *This is a classification task*
- *How can we do it? We use NNs*
  - ↳ *We train an NN over lots of images: we have lots of sequence of digits*
  - ↳ *We then use it to recognize new images*

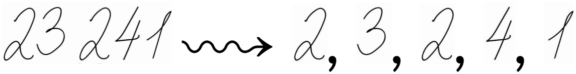
⇝ 23241

Let's say we are going to use a CNN

Computing Loss: *Motivating Example*

To use a CNN, we need to specify our input size

- *We segment an input image into a sequence of five images*
  - ↳ *These images are all as large as CNN's input size*

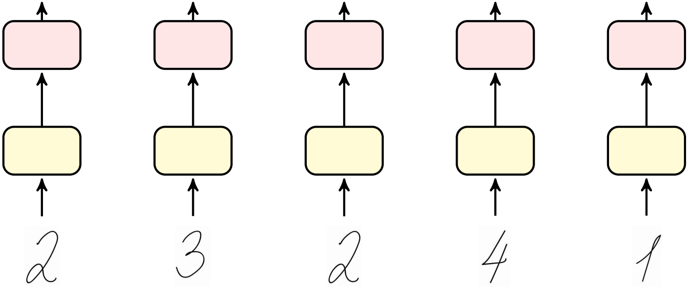


- *We label each image with its label, e.g.,*  *is labeled as 2*
- *We give these five images to our CNN and get five outputs*
  - ↳ *Assume we use softmax at the output layer*
  - ↳ *For each image, we get a vector of size 4 as output*
    - ↳ *Each entry represents probability of image being one of digits 1, 2, 3, and 4*
- *To compute loss, we compare each output with its corresponding label*

Computing Loss: *Motivating Example*

$$\mathbf{v}[1] = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{v}[2] = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{v}[3] = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{v}[4] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{v}[5] = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} y_1[1] \\ y_2[1] \\ y_3[1] \\ y_4[1] \end{bmatrix} = \mathbf{y}[1] \quad \mathbf{y}[2] \quad \mathbf{y}[3] \quad \mathbf{y}[4] \quad \mathbf{y}[5]$$



↳ $y_j[t]$ *is the probability of digit in time t being j*

Computing Loss: *Motivating Example*

Here, we already have the data segmented into

a sequence that for each time step has a label

So, computing loss is easy as pie!

$$\begin{aligned}\hat{R} &= \mathcal{L}\left(\mathbf{y}[1], \ldots, \mathbf{y}[5], \mathbf{v}[1], \ldots, \mathbf{v}[5]\right) \\ &= \sum_{t=1}^{5} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right) = \sum_{t=1}^{5} \hat{R}[t]\end{aligned}$$

When we compute gradients, we note that only $\hat{R}[t]$ depends on $\mathbf{y}[t]$: so, for a given output at time $t = i$ we can simply write

$$\nabla_{\mathbf{y}[i]} \hat{R} = \sum_{t=1}^{5} \nabla_{\mathbf{y}[i]} \hat{R}[t] = \nabla_{\mathbf{y}[i]} \hat{R}[i] = \nabla_{\mathbf{y}[i]} \mathcal{L}\left(\mathbf{y}[i], \mathbf{v}[i]\right)$$

Computing Loss: *One-to-One Correspondence*

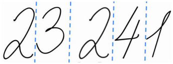
+ *But is it practical to do segmentation by hand?*
– No! This is why we built RNNs!

With RNNs, we address this learning task as bellow

- *We look at the complete image as a sequence of data*
  ↳ *We divide input into multiple equal-size frames*
- *We go over each frame separately*
  ↳ *We give the frame as the input along with previous state*
  ↳ *We compute a new state which can potentially give us the output*

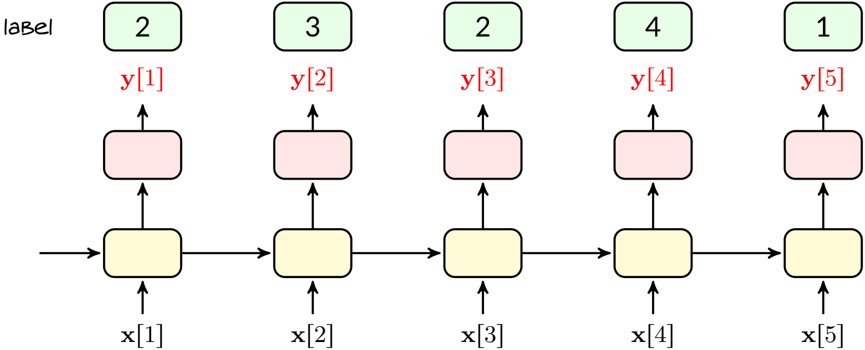
Computing Loss: *One-to-One Correspondence*

If we are extremely lucky; then, our segmentation looks like this



$$\rightsquigarrow \mathbf{x}[1], \mathbf{x}[2], \mathbf{x}[3], \mathbf{x}[4], \mathbf{x}[5]$$

and we have a label for each time step

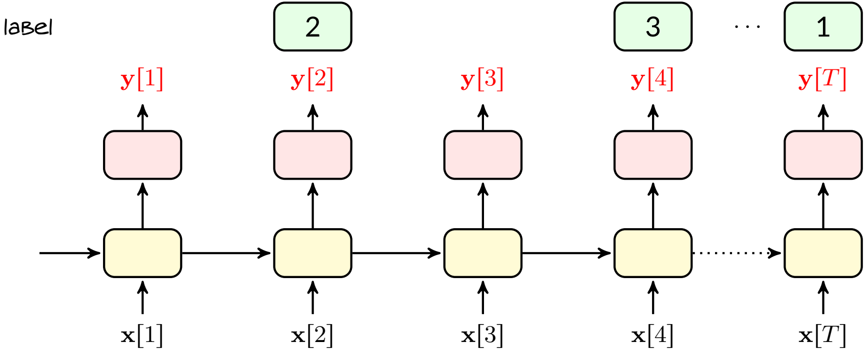


Computing Loss: *One-to-One Correspondence*

But, that's too good to happen! Usually we have

$\rightsquigarrow \mathbf{x}[1], \mathbf{x}[2], \ldots, \mathbf{x}[T]$

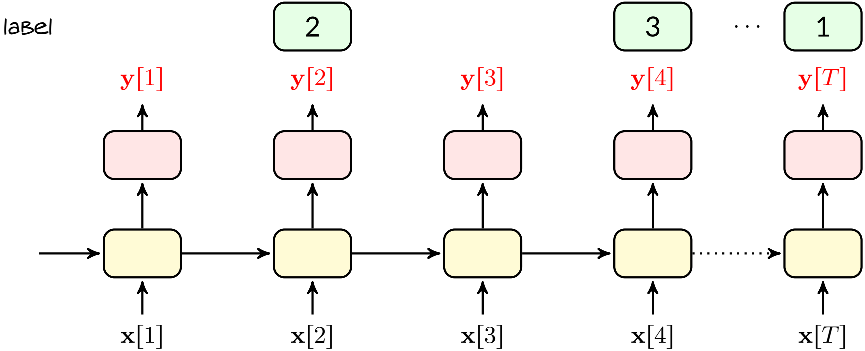
and we have a label in some time steps



Computing Loss: *One-to-One Correspondence*

In this typical case, two questions seem non-trivial

1. *Where should we put each label?* $\equiv$ *Where should we read each label?*
2. *What should we do with non-labeled outputs, e.g.,* $\mathbf{y}[1]$?



Computing Loss: *One-to-One Correspondence*

The key challenge in computing the loss is that *we do not have necessarily one-to-one correspondence* with *sequence data*

Correspondence Problem

With sequence data, we could have a data-sequence of length T that is labeled by a sequence of size $K < T$ where

no time index is specified for any label in the K-long label sequence

Correspondence problem exists pretty much in all practical sequence data

- *In speech recognition, multiple time frames correspond to a single word*
- *In text recognition, multiple image frames correspond to a single letter*
- $\ldots$

Correspondence Problem: *Formulation*

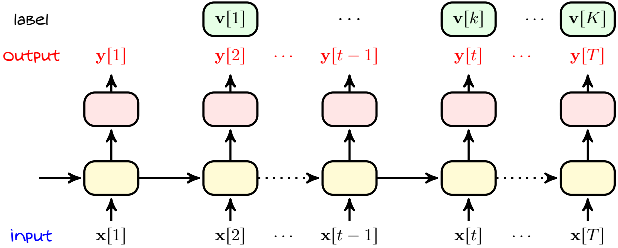
Let's formulate the problem clearly: *Say we have*

> *A sequence of data*
>
> $$\mathbf{x}[1:T] = \mathbf{x}[1], \ldots, \mathbf{x}[T]$$
>
> *that is labeled with the sequence of K true labels*
>
> $$\mathbf{v}[1:K] = \mathbf{v}[1], \ldots, \mathbf{v}[K]$$
>
> *where K and T can be different*

For this setting, we want to train an RNN with this data sequence: starting with an initial state, this RNN returns an output sequence

$$\mathbf{y}[1:T] = \mathbf{y}[1], \ldots, \mathbf{y}[T]$$

Correspondence Problem: *Formulation*



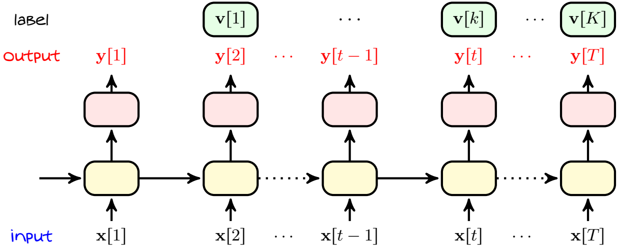
To be able to train this RNN, we need to

1. *define a loss function that computes* $\hat{R} = \mathcal{L}\left(\mathbf{y}[1:T], \mathbf{v}[1:K]\right)$

    ↳ *We need this loss function to be differentiable with respect to all outputs*

$$\nabla_{\mathbf{y}[1]}\hat{R}, \ldots, \nabla_{\mathbf{y}[T]}\hat{R}$$

Correspondence Problem: *Formulation*



To use this RNN after training, i.e., for inferring, we need to

2. *know how to map outputs to predicted labels*
   ↳ *We need to extract K labels from $\mathbf{y}[1:T]$, i.e.,*

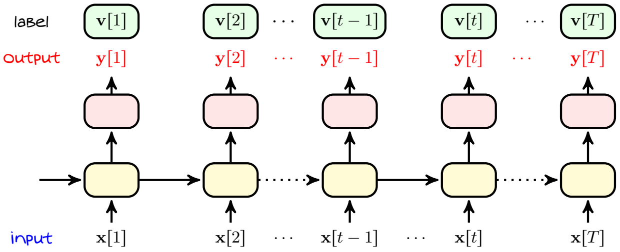
$$\mathbf{y}[1], \ldots, \mathbf{y}[T] \mapsto \hat{\mathbf{v}}[1], \ldots, \hat{\mathbf{v}}[K]$$

Let's look into different settings

Setting I: *Perfectly Segmented*

In some problems, *we have our data perfectly segmented*

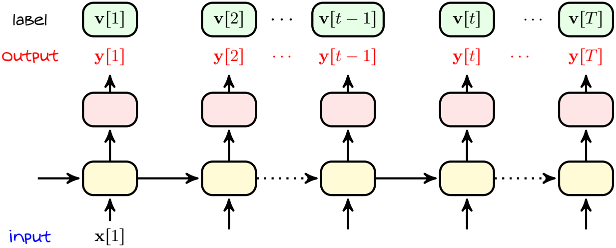
- *There is a separate label for each time step, i.e.,* $K = T$
  - ↳ *many-to-many type I*



Setting I: *Perfectly Segmented*

In some problems, *we have our data perfectly segmented*

- *There is a separate label for each time step, i.e.,* $K = T$
  - ↳ *many-to-many type I and one-to-many*



Attention

We can always treat a non-existing input entry as an empty

↳ *We are good as long as we have a label at each time* t

Setting I: *Defining Loss*

In such settings, we define the loss *to be aggregated loss over time*

$$\hat{R} = \sum_{t=1}^{T} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)$$

for some loss function $\mathcal{L}\left(\cdot, \cdot\right)$

The gradients are then trivially computed

Gradient with respect to particular output $\mathbf{y}[t]$ *is*

$$\nabla_{\mathbf{y}[t]} \hat{R} = \nabla_{\mathbf{y}[t]} \hat{R}[t] = \nabla_{\mathbf{y}[t]} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[t]\right)$$

Setting I: *Inference*

Inference in such setting is performed *by one-to-one mapping: at time t, we predict based on* $\mathbf{y}[t]$

$$\mathbf{y}[1] \mapsto \hat{\mathbf{v}}[1], \dots, \mathbf{y}[T] \mapsto \hat{\mathbf{v}}[T]$$

For instance, assume $\mathbf{y}[t]$ *is output of a softmax activation; then, we set*

$$\hat{\mathbf{v}}[t] = \operatorname{argmax} \mathbf{y}[t]$$

where argmax *returns the index of the largest entry, e.g.,*

$$\operatorname{argmax} \begin{bmatrix} 0.1 \\ 0.7 \\ 0.2 \\ 0 \end{bmatrix} = 2$$

Setting II: *Known Segments*

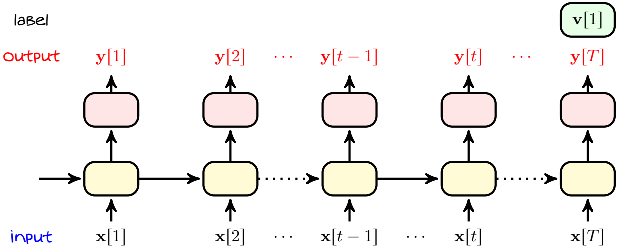
In some problems, *we have only one label for the whole sequence, i.e.,* $K = 1$

↳ *It corresponds to many-to-one type of problems*

↳ *This can be that we have really only one label, e.g., content classification*

↳ *It can be that we know the time index* t *at which each label is assigned*

↳ *We split data to sub-sequences with each sub-sequence having only one label*



Setting II: *Defining Loss for Dumb NN*

A naive approach to define loss is *to set it be the loss between last output and label*, i.e.,

$$\hat{R} = \mathcal{L}\left(\mathbf{y}[T], \mathbf{v}[1]\right)$$

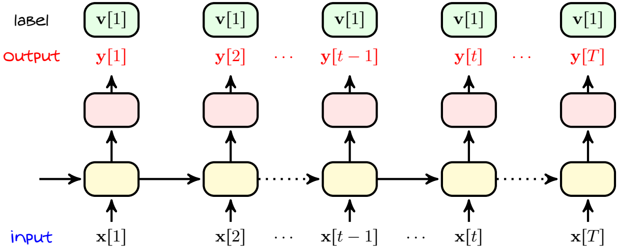
With this loss, *gradient with respect to particular output* $\mathbf{y}[t]$ *is*

$$\nabla_{\mathbf{y}[t]} \hat{R} = \begin{cases} \nabla_{\mathbf{y}[T]} \mathcal{L}\left(\mathbf{y}[T], \mathbf{v}[1]\right) & t = T \\ 0 & t \neq T \end{cases}$$

+ *But does it make sense to ignore all other outputs?*
- Not at all! *We are training a dumb NN that can respond only when it's over with the whole sequence!*

Setting II: *Loss for Smarter Training*

An extremely smart NN is the one who knows the label before the input speaks!



For this NN, the loss is

$$\hat{R} = \sum_{t=1}^{T} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[1]\right)$$

But, we should be careful! We should not expect NN to know everything from potentially irrelevant input!

Setting II: *Defining Proper Loss*

A realistic approach is *to define the loss via a weighted sum*, i.e.,

$$\hat{R} = \sum_{t=1}^{T} w_t \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[1]\right)$$

where w_t is the weight at time t

- *initially w_t is small*
  - ↳ *we do not expect the NN to know everything from very beginning*
- *it gradually increases up to its maximum w_T*
  - ↳ *by time T the NN should know the label*

With this loss, *gradient with respect to particular output $\mathbf{y}[t]$ is*

$$\nabla_{\mathbf{y}[t]} \hat{R} = w_t \nabla_{\mathbf{y}[t]} \hat{R}[t] = w_t \nabla_{\mathbf{y}[t]} \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[1]\right)$$

Setting II: *Inference*

Inference in such setting is performed *by many-to-one mapping: we only predict based on* $\mathbf{y}[1:T]$

$$\mathbf{y}[1:T] \mapsto \hat{\mathbf{v}}[1]$$

For instance, assume $\mathbf{y}[t]$ *is output of a softmax activation; then, we set*

$$\tilde{\mathbf{v}}[t] = \operatorname{argmax} \mathbf{y}[t]$$

and then take a (potentially weighted) majority vote: $\hat{\mathbf{v}}[1]$ *is the class that most often estimated with occurrence at each time being weighted by some weight*

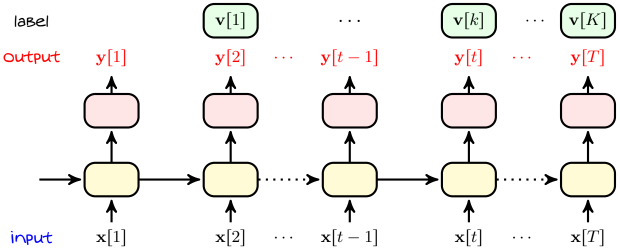
Setting III: *Unknown Segments*

Most common case is that *we have a label sequence shorter than our data*

↳ *Each label in this sequence is corresponding to a segment of input*

↳ *We do not know where this segment begins and where it ends*

↳ *There might be even no clear answer to that!*



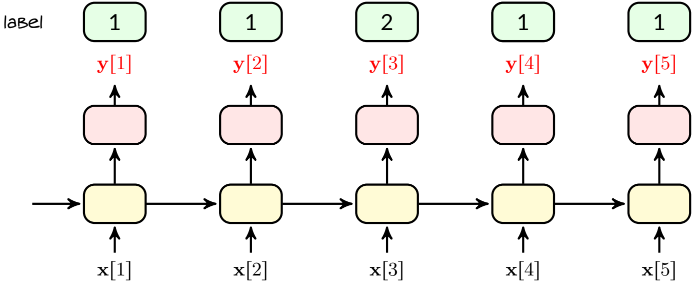
Note that we are dealing with a sequence to sequence model: we want to learn

relation between sequence $\mathbf{x}[1:T]$ and sequence $\mathbf{v}[1:K]$!

Setting II: *Example*

Assume we have image  *that is divided into a sequence of five pixel vectors*

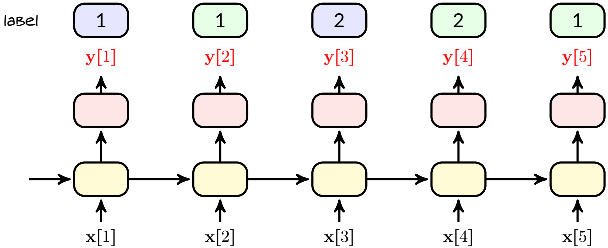
- *Since it is a training data, it is labeled as 121*
  - ↳ *We do not know after which output we should expect RNN to know first, second* or *third digit!*



+ *Sounds impossible!*
– *Only impossible is impossible!* Let's carry on and see what we can do!

Setting II: *Genie-Defined Loss*

Assume a genie has told us end of each segment

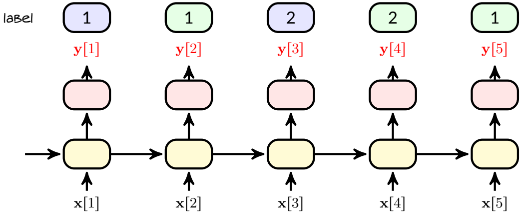


We can fill the empty labels with repetition, and then define the loss as

$$\hat{R} = \sum_{k=1}^{K} \sum_{t=i_{k-1}+1}^{i_k} w_t \mathcal{L}\left(\mathbf{y}[t], \mathbf{v}[k]\right)$$

where i_k is where label $\mathbf{v}_k$ ends, e.g., in above diagram $i_1 = 2$

Setting II: *Defining Loss*



We don't have the genie: we could assume that i_k is something to learn!

$$\hat{R}(\mathbf{i}) = \sum_{k=1}^{K} \sum_{t=i_{k-1}+1}^{i_k} w_t \mathcal{L}(\mathbf{y}[t], \mathbf{v}[k])$$

where $\mathbf{i} = [0, i_1 \dots, i_K]$ is something we need to learn

Setting II: *Optimal Segmentation*

+ *How could we learn $\mathbf{i}$? Should we compute also $\nabla_{\mathbf{i}} \hat{R}$?*
– Well! You may try! *But, obviously i_k is an integer!*

Optimal Segmentation

Optimal approach for finding $\mathbf{i}$ is to train the NN for all possible choice for $\mathbf{i}$ and then find the final training loss $\hat{R}(\mathbf{i})$. The optimal segmentation is then given by

$$\mathbf{i}^\star = \operatorname*{argmin}_{\mathbf{i}} \hat{R}(\mathbf{i})$$

+ *Is it computationally feasible?*
– No! *The number of possible choice for $\mathbf{i}$ grows exponentially with T! We need to go for sub-optimal approaches*

Setting II: *Number of Possible Segmentations*

+ *How is it exponentially large?*
– Let's look at our example

In our example, *we should assign label sequence 121 to a sequence of length 5: each entry of output sequence in this case can be labeled by 1 (the first one), 2 or 1 (the last one). This means that we have* 3 *choices of label for each time interval; thus, the total number of possible segmentations is around* 3^5.

In general number of segmentations grows exponentially with T

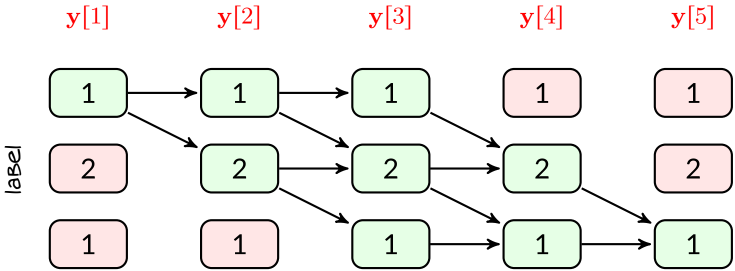
+ *But wait a moment! We have also counted the case of labeling all outputs with 1! This cannot be the case!*
– This is right! *It is in general much less than* 3^5 *but it's still exponential*

Let's see the exact possible segmentations!

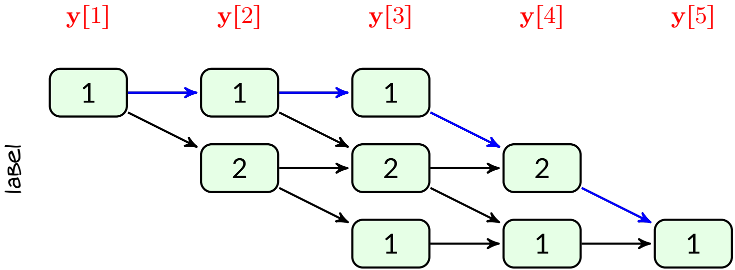
Setting II: *Number of Possible Segmentations*

We intend to compare each of $\mathbf{y}[1], \ldots, \mathbf{y}[5]$ *with a label*

- *We know that the label sequence is 121*
  - ↳ *First output is definitely in the first segment: it's label is definitely 1*
  - ↳ *Second output could be still in the first segment or in the second segment*
  - ↳ *Third output could be in the first, second, or third segment*
  - ↳ *Our labels should finish by the end of output sequence: fourth output cannot be in first segment*
  - ↳ *Last output could be only in the third segment*



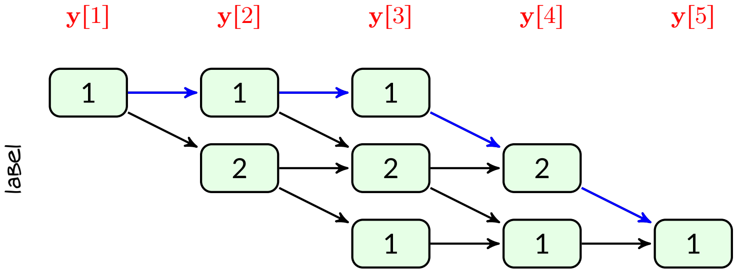
Setting II: *Showing Segmentations on Graph*



Though it's exponentially large: *we see that each segmentation corresponds to one path on this graph*

Blue path corresponds to $i_1 = 3$, $i_2 = 4$, *and* $i_3 = 5$, *i.e.*, $\mathbf{i} = [0, 3, 4, 5]$

Setting II: *Loss on Segmentation Graph*

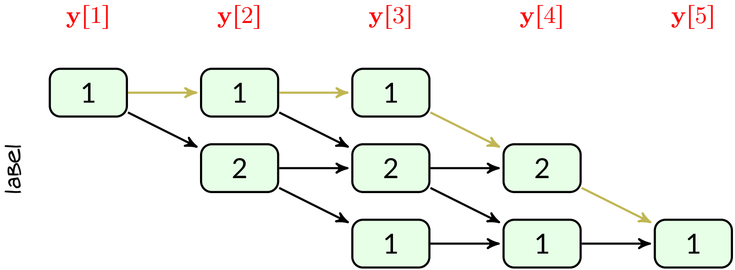


We can compute the loss for each segmentation directly on this graph: let's say that we have L different paths on the graph. For each path, we can write an expanded label sequence, e.g.,

Expanded label sequence of blue path is { 1, 1, 1, 2, 1}

This sequence is of length T and we show it for path ℓ with $\tilde{\mathbf{v}}_\ell[t]$

Setting II: *Loss on Segmentation Graph*



For each path $\ell = 1, \dots, L$, the loss is computed by aggregating the losses between outputs and extended labels

$$\hat{R}_\ell = \sum_{t=1}^{T} w_t \mathcal{L}\left(\mathbf{y}[t], \tilde{\mathbf{v}}_\ell[t]\right) = \sum_{t=1}^{T} \hat{R}_\ell[t]$$

It again decomposes into sum of T terms with only one being function of $\mathbf{y}[t]$

Setting II: *Optimal Segmentation on Graph*

We can represent the optimal segmentation on the graph as below

```
OptimalSegmentTraining():
```

1: Initiate with $\hat{R} = +\infty$ and some random $\ell^\star = \varnothing$
2: **for** $\ell = 1, \ldots, L$ **do**
3: Let the loss be $\hat{R}_\ell$
4: Train for sufficient epochs
5: **if** *After training* $\hat{R}_\ell < \hat{R}$ **then**
6: $\hat{R} \leftarrow \hat{R}_\ell$ and $\ell^\star \leftarrow \ell$
7: **end if**
8: **end for**
9: Return learnable parameters and $\ell^\star$

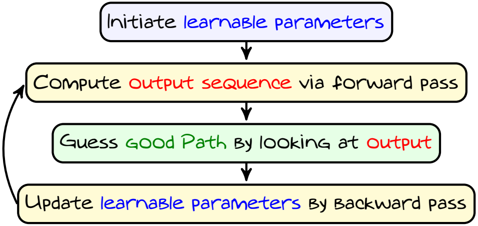
+ *Say we could be over with this infeasible training! How do we use the trained RNN for inference?*
– In this case, *we have $\ell^\star$ which gives us optimal segmentation: we infer label of each segment based on its corresponding outputs*

Setting II: *Maximum-Likelihood Segmentation*

Since optimal segmentation is infeasible, people uses *maximum-likelihood approach* that is *well-known in detection and coding theory*

Maximum-Likelihood Segmentation

Start with an initial guess for optimal path on segmentation graph and do one step of training; then, improve the guess based on the outputs of next forward pass and go for next step of training



Let's look at its pseudo-code

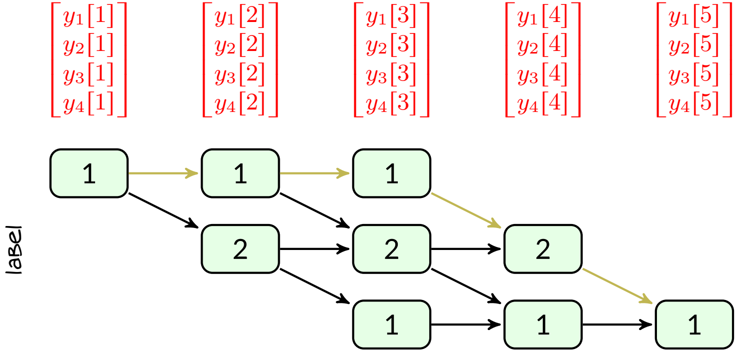
Setting II: *Maximum-Likelihood Segmentation*

`MaxLikelihoodTraining():`

1: **for** Iteration $i = 1, \ldots, I$ **do**
2: Pass forward through time: Compute output sequence $\mathbf{y}[1:T]$
3: Compute $p\left(\tilde{\mathbf{v}}_\ell[1:T] | \ell\right)$ for each path ℓ on segmentation graph ⊛
4: Update $\ell^* = \operatorname{argmax}_\ell p\left(\tilde{\mathbf{v}}_\ell[1:T] | \ell\right)$ ⊛
5: Set loss to $\hat{R}_{\ell *}$ and backpropagate over RNN
6: Update learnable parameters
7: **end for**
8: Return learnable parameters and ℓ^*

+ *Why we call it maximum likelihood?*
– Because we *guess path by maximizing the likelihood* $p\left(\tilde{\mathbf{v}}_\ell[1:T] | \ell\right)$
+ *But how can find likelihood of a path?*
– We can use output sequence $\mathbf{y}[1:T]$

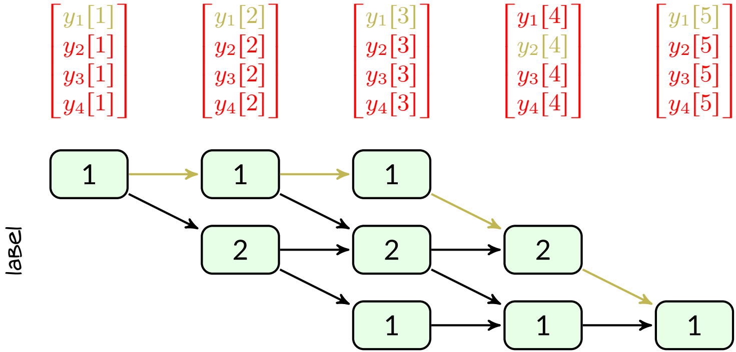
Setting II: *Finding Likelihood on Segmentation Graph*



Assume that each label could be 1, 2, 3, or 4: *at each time t the RNN returns a 4-dimensional vector whose entries are probability of each class*

we can multiply the probabilities of classes on the path

Setting II: *Finding Likelihood on Segmentation Graph*

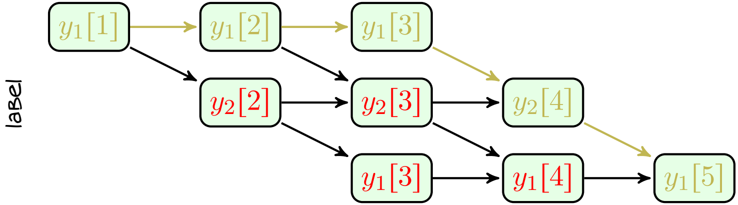


For instance, the yellow path has a likelihood

$$p\left(\tilde{\mathbf{v}}_\ell[1:T]|\ell\right) = \prod_{t=1}^{T} p\left(\tilde{\mathbf{v}}_\ell[t]|\ell\right) = y_1[1]y_1[2]y_1[3]y_2[4]y_1[5]$$

Setting II: *Finding Likelihood on Segmentation Graph*

$$\begin{bmatrix} y_1[1] \\ y_2[1] \\ y_3[1] \\ y_4[1] \end{bmatrix} \quad \begin{bmatrix} y_1[2] \\ y_2[2] \\ y_3[2] \\ y_4[2] \end{bmatrix} \quad \begin{bmatrix} y_1[3] \\ y_2[3] \\ y_3[3] \\ y_4[3] \end{bmatrix} \quad \begin{bmatrix} y_1[4] \\ y_2[4] \\ y_3[4] \\ y_4[4] \end{bmatrix} \quad \begin{bmatrix} y_1[5] \\ y_2[5] \\ y_3[5] \\ y_4[5] \end{bmatrix}$$



Or better to say: we just put output entries in graph and move on the path

$$p\left(\tilde{\mathbf{v}}_\ell[1:T] | \ell\right) = \prod_{t=1}^{T} y_{\tilde{v}_\ell[t]}[t] = y_1[1] y_1[2] y_1[3] y_2[4] y_1[5]$$

Setting II: *Maximum-Likelihood Segmentation*

+ *OK! We can find the likelihood, but how can we maximize it? It's again an exponentially large search!*

$$\ell^* = \operatorname*{argmax}_{\ell} p\left(\tilde{\mathbf{v}}_{\ell}[1:T] | \ell\right)$$

– Well! If we only need the maximum, *it turns not to be exponential*

We can readily show that finding maximum likelihood on the graph is a dynamic programming problem and can be solved by the Viterbi algorithm

Maximum likelihood training can be implemented efficiently

Setting II: *Maximum-Likelihood Inference*

MaxLikelihoodTraining():

1: **for** Iteration $i = 1, \ldots, I$ **do**
2: Pass forward through time: Compute output sequence $\mathbf{y}[1:T]$
3: Compute $p\left(\tilde{\mathbf{v}}_\ell[1:T]|\ell\right)$ for each path ℓ on segmentation graph ⊛
4: Update $\ell^* = \operatorname{argmax}_\ell p\left(\tilde{\mathbf{v}}_\ell[1:T]|\ell\right)$ ⊛
5: Set loss to $\hat{R}_{\ell*}$ and backpropagate over RNN
6: Update learnable parameters
7: **end for**
8: Return learnable parameters and ℓ^*

+ *How can we use our RNN for inference after training via maximum likelihood segmentation?*
– We have access to ℓ^*: *we predict the label of each segment based on its corresponding outputs*

Setting II: *Connectionist Temporal Classification*

It turns out that *maximum-likelihood could stick to a bad local minimum, i.e.,*

it quickly converges to a path ℓ^* that is much different from $\ell^\star$

+ *Is there any solution to this?*
– Yes! We can use *connectionist temporal classification (CTC) loss*

CTC Loss

Instead of searching for a best segmentation and then minimizing its loss, we learn directly from unsegmented data by minimizing the average loss over all possible segmentations, i.e., we define loss to be

$$\hat{R} = \mathbb{E}_\ell \left\{ \hat{R}_\ell \right\} = \sum_{\ell=1}^{L} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \hat{R}_\ell$$

and train the RNN by finding learnable parameters that minimize this loss

Setting II: *CTC Loss*

+ *But, why should it be a better choice of loss?*
– Because *we are sure that optimal segmentation is contributing to our loss*

$$\hat{R} = \mathbb{E}_\ell \left\{ \hat{R}_\ell \right\} = \sum_{\ell=1}^{L} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \hat{R}_\ell$$
$$= p\left(\ell^\star | \tilde{\mathbf{v}}_{\ell^\star}[1:T]\right) \hat{R}_{\ell^\star} + \sum_{\ell \neq \ell^\star} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \hat{R}_\ell$$

+ *Agreed! Now, how should we determine* $p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right)$ *?*
– Just use the Bayes rule!
+ *What about the expectation? It is at the end sum of exponentially large number of terms!*
– We can again *go on the graph and determine it via dynamic programming*

Setting II: *CTC Loss*

The CTC loss can be written as

$$\hat{R} = \sum_{\ell=1}^{L} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \hat{R}_\ell = \sum_{\ell=1}^{L} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \sum_{t=1}^{T} w_t \mathcal{L}\left(\mathbf{y}[t], \tilde{\mathbf{v}}_\ell[t]\right)$$
$$= \sum_{t=1}^{T} w_t \underbrace{\sum_{\ell=1}^{L} p\left(\ell | \tilde{\mathbf{v}}_\ell[1:T]\right) \mathcal{L}\left(\mathbf{y}[t], \tilde{\mathbf{v}}_\ell[t]\right)}_{\breve{R}[t]} = \sum_{t=1}^{T} w_t \breve{R}[t]$$

This has been shown that $\breve{R}[t]$ can be recursively computed[1]*:*

by some approximation we are able to readily compute $\nabla_{\mathbf{y}[t]} \breve{R}[t]$

and we set $\nabla_{\mathbf{y}[t']} \breve{R}[t] \approx \mathbf{0}$ for $t' \neq t$

[1]Check out the original paper

Setting II: *Training with CTC Loss*

```
CTC_Training():
```

1: **for** iteration $i = 1, \ldots, I$ **do**
2: Pass forward through time: Compute output sequence $\mathbf{y}[1:T]$
3: Compute CTC loss $\hat{R}$ and $\nabla_{\mathbf{y}[t]} \hat{R}$ by recursion
4: Backpropagate through time and update learnable parameters
5: **end for**
6: Return learnable parameters

This looks like standard training loop now

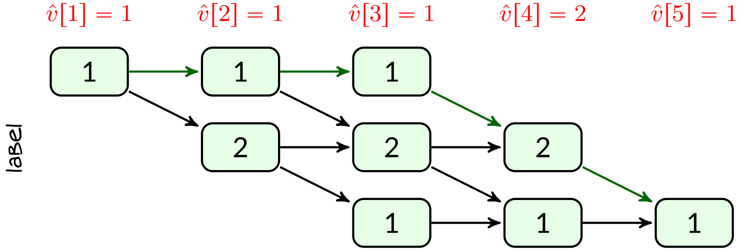
the loss is only replaced with CTC loss

+ *What about inference?*
- Well! We should figure it out, *since the training loop does not compute any segmentation path!*

Setting II: *Inference with CTC-Trained RNN*

Let's get back to our simple example: *assume that after training with CTC loss we give an image of handwritten* 121 to the RNN

- *RNN divides it into 5 frames and is able to track optimal segmentation*
  - ↳ *The first three frames belong to the first segment*
  - ↳ *The remaining frames belong to the second and third segments*
- *RNN infers from output sequence $\hat{v}[1:5]$ but does not return optimal path*



Setting II: *Inference with CTC-Trained RNN*

We can conclude from $\hat{v}[1:5]$ that the sequence is {1,2,1} *if we are sure the sequence has no repetition*

Label Encoding and Decoding

CTC uses this fact and constructs following encoding and decoding method: it introduces a new label called "blank:-" which does not belong to set of classes

- *While training, it adds blank between any two repetitions*
  - ↳ *For instance, we encode 112 $\mapsto$ 1-12, or 111 $\mapsto$ 1-1-1*
- *For inference, it removes any repetition in inferred sequence $\hat{v}[1:T]$ and then drops blanks*
  - ↳ *For instance, we decode 1-11-312 $\mapsto$ 11312, or 3333-3121 $\mapsto$ 33121*

Setting II: *Training and Inference with CTC*

```
CTC_Training():
```

1: **for** iteration $i = 1, \ldots, I$ **do**
2: Add blanks to the label sequences with repetition
3: Pass forward through time: Compute output sequence $\mathbf{y}[1:T]$
4: Compute CTC loss $\hat{R}$ and $\nabla_{\mathbf{y}[t]} \hat{R}$ by recursion
5: Backpropagate through time and update learnable parameters
6: **end for**
7: Return learnable parameters

```
CTC_Inference():
```

1: Pass forward through time the input and compute output $\mathbf{y}[1:T]$
2: Infere encoded sequence $\hat{v}[1:T]$ from $\mathbf{y}[1:T]$
3: Remove repetitions from $\hat{v}[1:T] \mapsto \hat{v}[1:T']$
4: Remove blanks from $\hat{v}[1:T'] \mapsto \hat{v}[1:K]$
5: Return $\hat{v}[1:K]$

In PyTorch: *CTC Loss*

We can access CTC loss in `torch.nn` module as

```
torcn.nn.CTCLoss()
```

Few notes about CTC loss implementation

- *We need to specify the index of blank label*
  - ↳ *It should be out of our set of classes*
  - ↳ *By default, it is set to* `blank = 0`
- *When we define our model, we should always take blank label into account*
  - ↳ *If we do classification with C classes, model should return $C+1$ classes with blank being one of them*
- *PyTorch considers cross-entropy loss function, i.e.,* $\mathcal{L}(\mathbf{y}, \tilde{\mathbf{v}}) = \mathrm{CE}(\mathbf{y}, \tilde{\mathbf{v}})$
- *As input to CTC loss:* $\mathbf{y}$ *should be logarithm of probabilities*
  - ↳ *We can activate the output layer with logarithmic softmax*