# ECE 1508S2: Applied Deep Learning

## Chapter 7: Sequence-to-Sequence Models

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Winter 2024

# Learning Sequence from Sequence

In many applications, our dataset contains data-points that of the form

$$(\mathbf{x}[t], \mathbf{v}[t])$$

# Learning Sequence from Sequence

In many applications, our dataset contains data-points that of the form

$$(\mathbf{x}[t], \mathbf{v}[t])$$

*This is still standard supervised learning where*

- *input data is a sequence, and*
- *label is a sequence*

## Learning Sequence from Sequence

In many applications, our dataset contains data-points that of the form

$$(\mathbf{x}[t], \mathbf{v}[t])$$

*This is still standard supervised learning where*

- *input data is a sequence, and*
- *label is a sequence*

*We have already seen several examples with RNNs*

- *We want to train a machine translator*
  - ↳ *Dataset contains sequences of German sentences with English translations*
- *We want to caption an image*
  - ↳ *Dataset contains images with sequences of caption sentences*
- *We want to predict next words*
  - ↳ *Dataset contains sequences of sentences with label being last word*

# Sequence-to-Sequence Problem

+ *What is the key point in such learning problems?*
- They are often called a *sequence-to-sequence problem, since we intend to learn an output sequence from an input sequence*

# Sequence-to-Sequence Problem

+ *What is the key point in such learning problems?*
- They are often called a *sequence-to-sequence problem, since we intend to learn an output sequence from an input sequence*

## Sequence-to-Sequence (Seq2Seq) Models

*A Seq2Seq model is a model, e.g., a NN, that takes a sequence as an input and returns an output sequence*

# Sequence-to-Sequence Problem

+ *What is the key point in such learning problems?*
– They are often called a *sequence-to-sequence problem, since we intend to learn an output sequence from an input sequence*

## Sequence-to-Sequence (Seq2Seq) Models

*A Seq2Seq model is a model, e.g., a NN, that takes a sequence as an input and returns an output sequence*

+ *Then isn't RNN a Seq2Seq model?*
– Sure! Strictly speaking *even MLPs and CNNs are Seq2Seq models with sequences of length 1!*

*Despite this definition, when we talk about Seq2Seq models in practice, we mainly refer to architectures with encoder and decoder*

# First Seq2Seq Model

Let's start with a simple task: *we intend to train a model that generates coherent sentences*

# First Seq2Seq Model

Let's start with a simple task: *we intend to train a model that generates coherent sentences*

- *After training it should be able to generate meaningful sentences*
  - ↳ *If we give in an incomplete sentence, it can complete it*
  - ↳ *If we don't give it an input, it generates a random meaningful sentence*

# First Seq2Seq Model

Let's start with a simple task: *we intend to train a model that generates coherent sentences*

- *After training it should be able to generate meaningful sentences*
  - ↳ *If we give in an incomplete sentence, it can complete it*
  - ↳ *If we don't give it an input, it generates a random meaningful sentence*

---

Since, we only know RNNs up to now, we are going to use an RNN

- *As model we want to train an RNN*
  - ↳ *It could be an LSTM, a GRU, or even a basic RNN*
  - ↳ *It can be shallow or deep*
- *We are doing to train this RNN via a given dataset*
  - ↳ *We compute the loss by some loss function, e.g., cross-entropy*
  - ↳ *We could also use CTC loss in case we have correspondence issue*

# Seq2Seq Model: *Basic Language Model*

*We intend to train a model that generates coherent sentences*

*This is a basic natural language model*

# Seq2Seq Model: *Basic Language Model*

*We intend to train a model that generates coherent sentences*

*This is a basic natural language model*

*You learn in detail about it in ECE 1786: Creative Applications of NLP*

↳ *You may consider taking it in the next fall semester*

# Seq2Seq Model: *Basic Language Model*

*We intend to train a model that generates coherent sentences*

*This is a basic natural language model*

*You learn in detail about it in ECE 1786: Creative Applications of NLP*

↳ *You may consider taking it in the next fall semester*

---

*Let's write our dataset first: it contains sequences of coherent English sentences*

Ali Bereyhi is the coolest professor at UofT!

# Seq2Seq Model: *Basic Language Model*

*We intend to train a model that generates coherent sentences*

*This is a basic natural language model*

*You learn in detail about it in ECE 1786: Creative Applications of NLP*

↳ *You may consider taking it in the next fall semester*

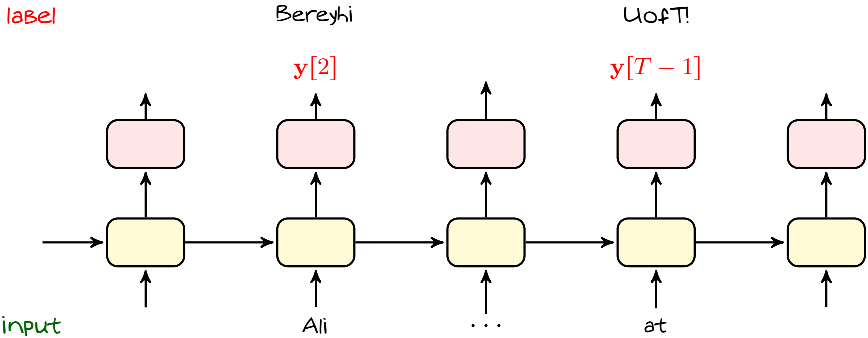
---

*Let's write our dataset first: it contains sequences of coherent English sentences*

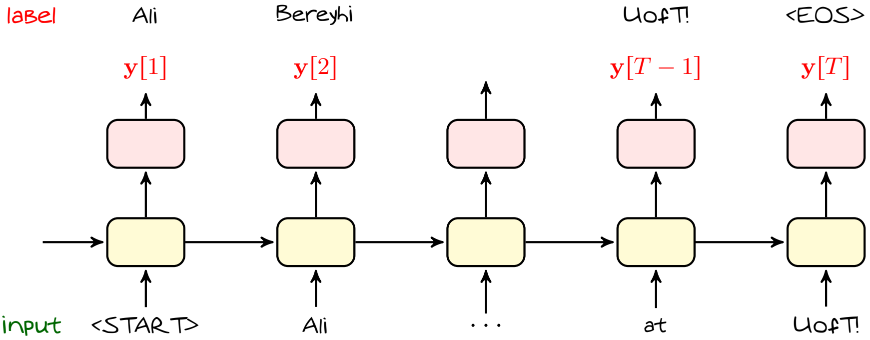
Ali Bereyhi is the coolest professor at UofT!

- *Sentences are of different lengths, i.e., $T$ is different for each sequence*
  - ↳ *Each word in a sentence is one input entry*
  - ↳ *We label each word with its next word in the sentence*

# Basic Language Model: *Dataset*

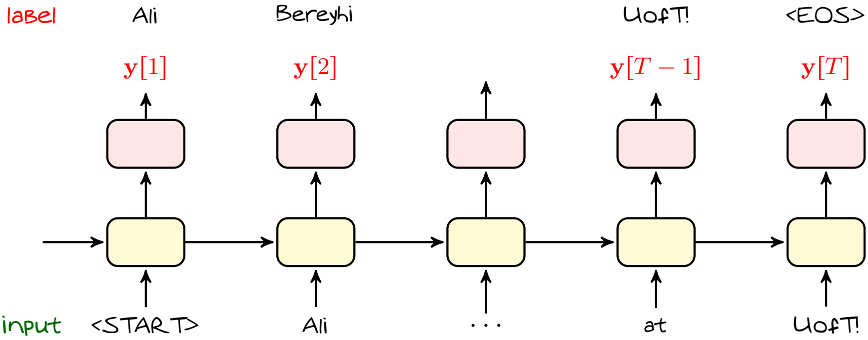


# Basic Language Model: *Dataset*




- *To be able to predict first word and end of sentence we add two new words*
  - ↳ *We tag the beginning of sentence with <START>*
  - ↳ *We label the end of sentence with <EOS>*

# Basic Language Model: *Dataset*




- *To be able to predict first word and end of sentence we add two new words*
  - ↳ *We tag the beginning of sentence with <START>*
  - ↳ *We label the end of sentence with <EOS>*
- *We do not have the correspondence issue in this problem*

# Basic Language Model: *Dataset*

+ *How can we feed those words to our RNN?*
– We convert them to *vectors by some method*
  ↳ *You can learn those methods in ECE 1786*

# Basic Language Model: *Dataset*

+ *How can we feed those words to our RNN?*
– We convert them to *vectors by some method*
  ↳ *You can learn those methods in ECE 1786*

---

*The basic approach is to make a token for each word*

- *We list all possible words and index them by* $1$ *to* $D$
  ↳ $D$ *could be very large: just imagine how many words we could say!*
  ↳ *The set of these words is what we call vocabulary*

# Basic Language Model: *Dataset*

+ *How can we feed those words to our RNN?*
- We convert them to *vectors by some method*
  - ↳ *You can learn those methods in ECE 1786*

---

*The basic approach is to make a token for each word*

- *We list all possible words and index them by* $1$ *to* $D$
  - ↳ $D$ *could be very large: just imagine how many words we could say!*
  - ↳ *The set of these words is what we call vocabulary*
- *We add* <START> *with index* $0$ *and* <EOS> *with* $D+1$

# Basic Language Model: *Dataset*

+ *How can we feed those words to our RNN?*
- We convert them to *vectors by some method*
  - ↳ *You can learn those methods in ECE 1786*

---

*The basic approach is to make a token for each word*

- *We list all possible words and index them by* $1$ *to* $D$
  - ↳ $D$ *could be very large: just imagine how many words we could say!*
  - ↳ *The set of these words is what we call vocabulary*
- *We add* <START> *with index* $0$ *and* <EOS> *with* $D+1$
- *We show each character by its one-hod vector which is called token, e.g.,*

$$\text{<START>} \mapsto \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix} \qquad\qquad \text{<EOS>} \mapsto \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix}$$

# Basic Language Model: *Dataset*

+ *How can we feed those words to our RNN?*
– We convert them to *vectors by some method*
  ↳ *You can learn those methods in ECE 1786*

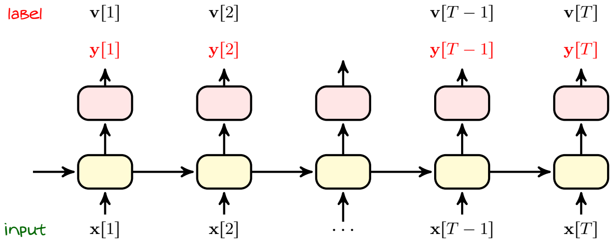
---

*The basic approach is to make a token for each word*

- *We list all possible words and index them by $1$ to $D$*
  ↳ *$D$ could be very large: just imagine how many words we could say!*
  ↳ *The set of these words is what we call vocabulary*
- *We add <START> with index $0$ and <EOS> with $D+1$*
- *We show each character by its one-hod vector which is called token, e.g.,*

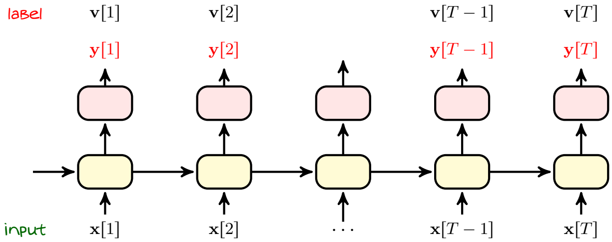
$$\text{<START>} \mapsto \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix} \in \{0,1\}^{D+2} \qquad \text{<EOS>} \mapsto \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix} \in \{0,1\}^{D+2}$$

# Basic Language Model: *Dataset*



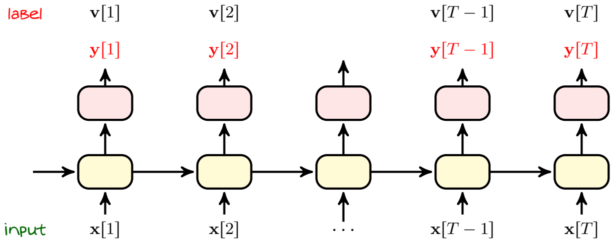

- *We can replace every word with its token*
  - ↳ *We now have a standard many-to-many setting*
  - ↳ *We could also use "embedding" to assign vectors to words*
    - ↳ *This will be taught in ECE 1786*

# Basic Language Model: *Training*



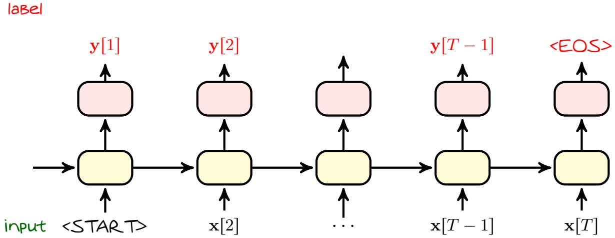

- *We now train the RNN with our dataset*
  - ↳ *We break dataset into mini-batches*
  - ↳ *For each data-point we pass first forward and then backward through time*
  - ↳ *We compute aggregated loss for each point and average over mini-batch*

# Basic Language Model: *Training*



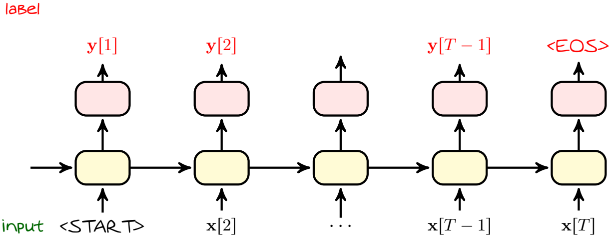

- *We now train the RNN with our dataset*
  - ↳ *We break dataset into mini-batches*
  - ↳ *For each data-point we pass first forward and then backward through time*
  - ↳ *We compute aggregated loss for each point and average over mini-batch*
- *After a certain number of epochs, we have the trained RNN*

# Basic Language Model: *Inference*




- *If we want to generate a random sentence, we can give <START>*
  - ↳ *It generates a word in each time step*
  - ↳ *Intuitively, these sentences are correlated to what RNN learned from dataset*

# Basic Language Model: *Inference*




- *If we want to generate a random sentence, we can give <START>*
  - ↳ *It generates a word in each time step*
  - ↳ *Intuitively, these sentences are correlated to what RNN learned from dataset*
- *If we want to complete the sentence, we give the initial part*
  - ↳ *It keeps on generating till <EOS>*
  - ↳ *Intuitively, this is correlated to what RNN learned and the input part*

# Sequence Generation: *Caption Generation*

Sentence completion worked simply with RNN: *mainly following the fact that entries of input and output sequences are of same nature*

↳ *They are both tokens*

↳ *But in practice, we may have different types of sequences*

# Sequence Generation: *Caption Generation*

Sentence completion worked simply with RNN: *mainly following the fact that entries of input and output sequences are of same nature*

↳ *They are both tokens*

↳ *But in practice, we may have different types of sequences*

---

*Let's consider another example: we want to train an NN that gets an image and writes a caption for it*

- *It gets as input a single image: a sequence of length one*
  - ↳ *For instance a* $256 \times 256$ *RGB image of a cat*

# Sequence Generation: *Caption Generation*

Sentence completion worked simply with RNN: *mainly following the fact that entries of input and output sequences are of same nature*

↳ *They are both tokens*

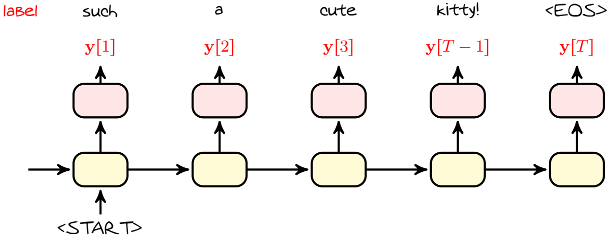
↳ *But in practice, we may have different types of sequences*

---

*Let's consider another example: we want to train an NN that gets an image and writes a caption for it*

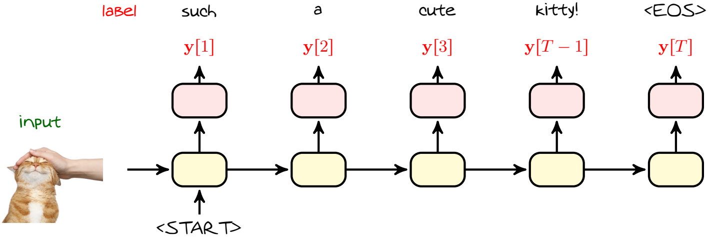
- *It gets as input a single image: a sequence of length one*
  - ↳ *For instance a $256 \times 256$ RGB image of a cat*
- *It returns a sentence: potentially a along sequence*
  - ↳ *For instance the sentence* such a cute kitty!

# Caption Generation: *Model*



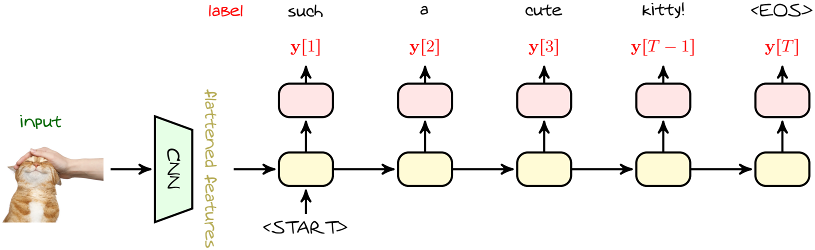

- *We can to generate a meaningful sentence with our basic language model*
  - ↳ *The sentence is probably not relevant to the image*
  - ↳ *We need to make the RNN speak about the cat image*

# Caption Generation: *Model*



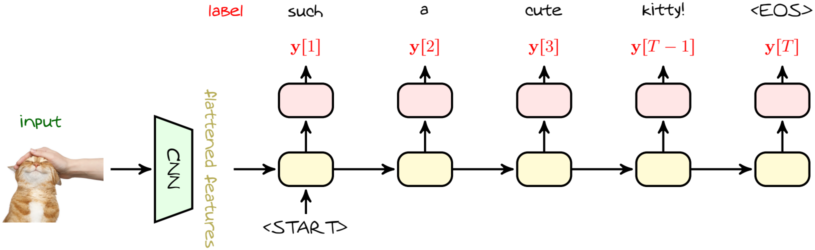

- *We can to generate a meaningful sentence with our basic language model*
  - ↳ *The sentence is probably not relevant to the image*
  - ↳ *We need to make the RNN speak about the cat image*
- *Maybe, we could set initial state of the RNN depending on the image*
  - ↳ *We need to extract features of image*
  - ↳ *Those features are going to explain what is inside the image*

# Caption Generation: *Encoder-Decoder*




- *We can extract a rich vector of features from the image via a CNN*
  ↳ *We use multiple convolutional layers to extract features*
  ↳ *We flatten those features and give it as a initial state to the RNN*

# Caption Generation: *Encoder-Decoder*




- *We can extract a rich vector of features from the image via a CNN*
  - ↳ *We use multiple convolutional layers to extract features*
  - ↳ *We flatten those features and give it as a initial state to the RNN*

*This architecture is called an encoder-decoder* model

↳ *A CNN is used to encode input to a good vector of features*

↳ *An RNN is used to decode extracted features to a desired label sequence*

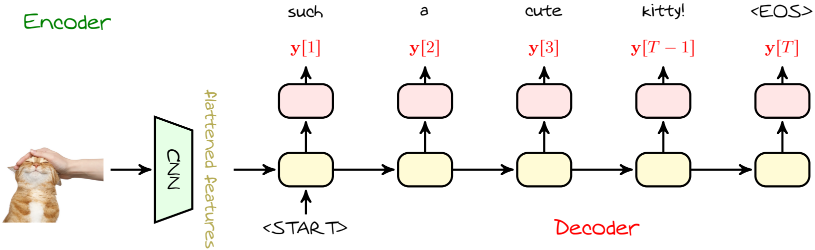
# Encoder-Decoder Architecture

## Encoder-Decoder

*Encoder-decoder architecture comprises of two separate NNs*

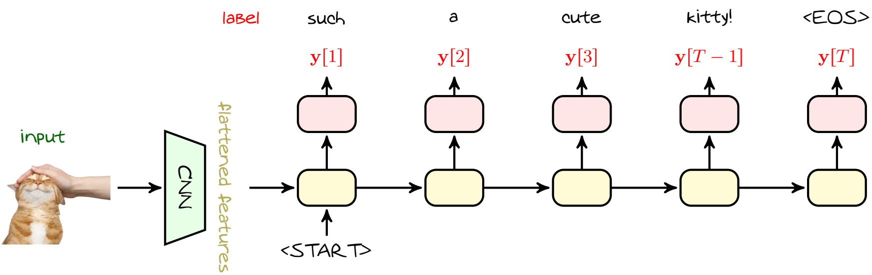
1. *Encoder takes the input sequence and encodes it into vector of features*
2. *Decoder takes vector of features and decodes it into output sequence*

+ *What kind of NNs should we use?*
- Pretty much everything is allowed!



# Back to *Caption Generation*

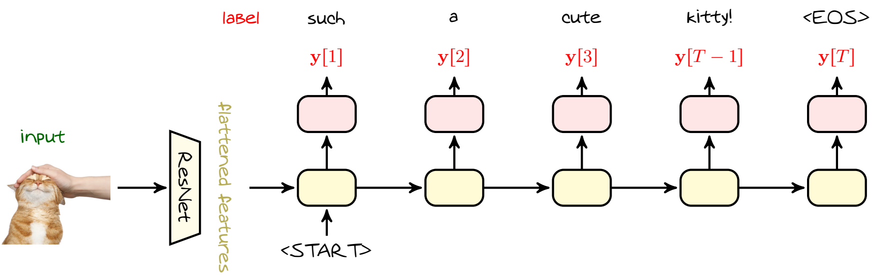
*We used a CNN for encoding and an RNN for decoding*




*We can replace CNN with any other architecture the extracts features*

# Back to *Caption Generation*

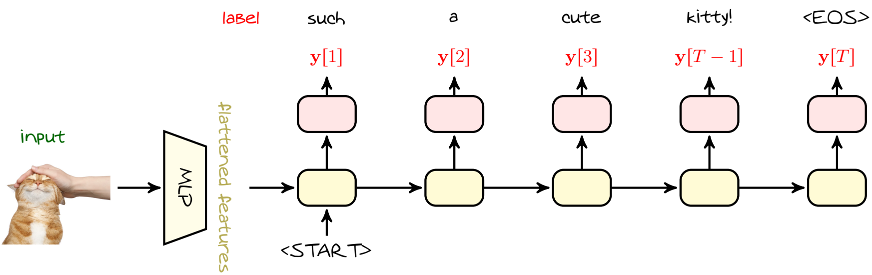
*We used a CNN for encoding and an RNN for decoding*




*We can replace CNN with any other architecture the extracts features*

# Back to *Caption Generation*

*We used a CNN for encoding and an RNN for decoding*




*We can replace CNN with any other architecture the extracts features*

# RNN as Conditional Distribution

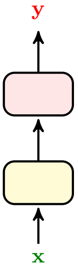
+ *But, why should those features make RNN speak about cat?*
– It makes RNN to *generate random words conditional to input image*

# RNN as Conditional Distribution

+ *But, why should those features make RNN speak about cat?*
– It makes RNN to *generate random words* conditional *to input image*

---

When we are dealing with classification: *NN can be seen as a machine that computes distribution and based on its input, it generates random outputs*

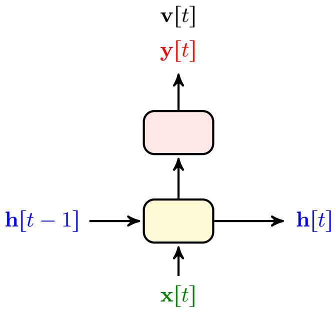



*In classification* $\mathbf{y}$ *is a vector if probability*

- *Its length equals to the number of classes*
- *Its entry $k$ represents the probability of class $k$*
  - ↳ *We can say that*

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \dots \\ y_K \end{bmatrix} \leftrightsquigarrow y_k \propto \Pr\{\text{label} = k | \mathbf{x}\}$$

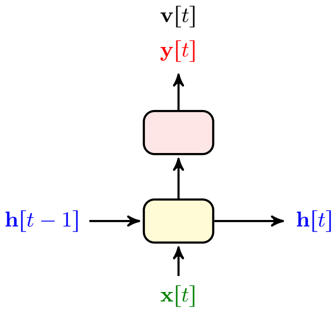
# RNN as Conditional Distribution




*Similarly the output of RNN in each time can be seen as*

$$y_k[t] \propto \Pr\left\{\text{label} = k | \mathbf{h}[t-1], \mathbf{x}[t]\right\} = p\left(\mathbf{v}[t] | \mathbf{h}[t-1], \mathbf{x}[t]\right)$$

# RNN as Conditional Distribution



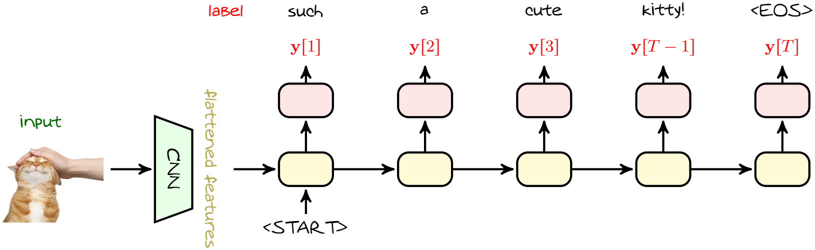

*Similarly the output of RNN in each time can be seen as*

$$y_k[t] \propto \Pr\{\text{label} = k | \mathbf{h}[t-1], \mathbf{x}[t]\} = p(\mathbf{v}[t] | \mathbf{h}[t-1], \mathbf{x}[t])$$

*Since* $\mathbf{h}[t-1]$ *already contains memory about* $\mathbf{x}[1:t-1]$, *we could say*

$$y_k[t] \propto p(\mathbf{v}[t] | \mathbf{x}[1:t])$$

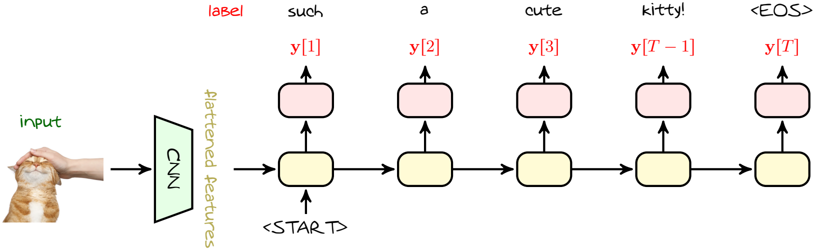
# Caption Generation: *Dataset*




*To train this architecture, we collect a dataset*

- *It contains several images*
  - ↳ *They could potentially be of different classes*
- *For each image, we have a sample caption*
  - ↳ *These sentences are again of different lengths*

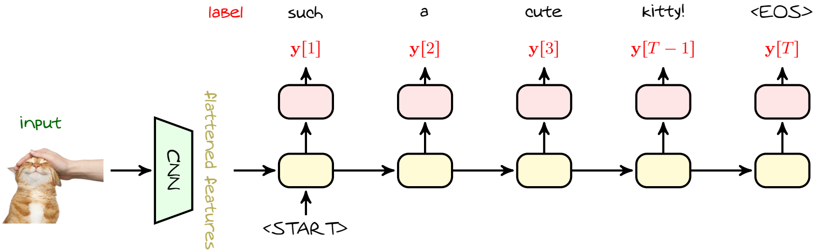
# Caption Generation: *Training*




*Say we want to train it on one sample: first we pass forward*

- *We first tokenize the words in captions to take them as one-hot labels*
- *We pass the image forward through CNN and get the feature vector*
- *We initiate the RNN with the feature vector and give input* <START>
- *We pass forward through time till we see have the output sequence*

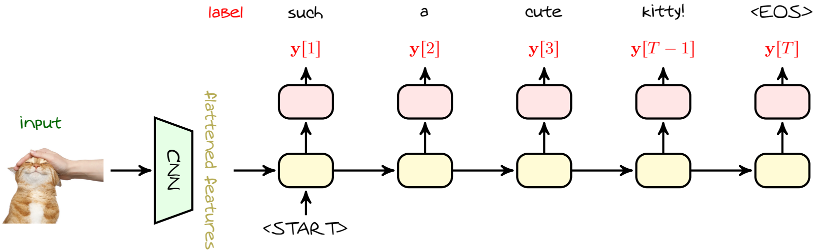
# Caption Generation: *Training*




*Say we want to train it on one sample: then we pass backward*

- *We compute the loss between the output sequence and one-hod labels*
  - ↳ *We can simply use the cross-entropy function*
- *We backpropagate through time till we arrive at the beginning of decoder*

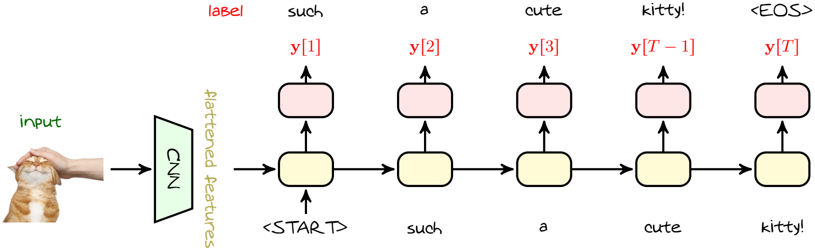
# Caption Generation: *Training*




*Say we want to train it on one sample: then we pass backward*

- *We compute the loss between the output sequence and one-hod labels*
  ↳ *We can simply use the cross-entropy function*
- *We backpropagate through time till we arrive at the beginning of decoder*
- *We have* $\nabla_{\text{features}} \hat{R}$
  ↳ *So we backpropagate through the RNN*
- *We update all weights and go for the next round*

# Caption Generation: *Inference*




*Say we finished with training: we want to caption a new image*

- *We send it over network and read the output sequence*
- *We could also set output of each time step as input for next time*
  - ↳ *We could also do it while training*
  - ↳ *Intuitively, it could help the RNN writing more coherent sentence*

# Seq2Seq Model: *Basic Translator*

Now, let's take a step further: *we want to build a model that translates German sentences to English*

- *We need a Seq2Seq model*
  - ↳ *We have a sequence of input German words*
  - ↳ *We need to return a sequence of English words*
  - ↳ *These sequences could be of different lengths*

# Seq2Seq Model: *Basic Translator*

Now, let's take a step further: *we want to build a model that translates German sentences to English*

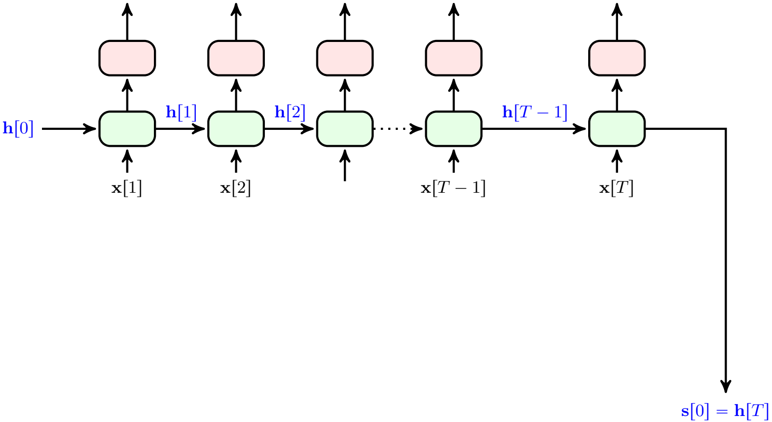
- *We need a Seq2Seq model*
  - ↳ *We have a sequence of input German words*
  - ↳ *We need to return a sequence of English words*
  - ↳ *These sequences could be of different lengths*

Since we know encoder-decoder model: *say we use it to build our translator*

- *We need a an encoder that takes a German sentence*
  - ↳ *RNN is a good choice, since we have input sequence*
- *We need a an decoder that returns the English translation*
  - ↳ *RNN is again the choice, since we have another sequence*

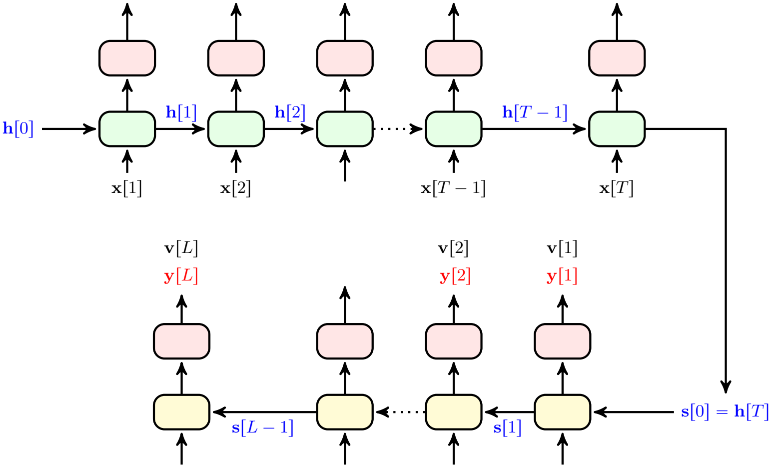
# Basic Translator: *Encoder-Decoder Model*

*So, the model for our translator looks like this*



# Basic Translator: *Encoder-Decoder Model*

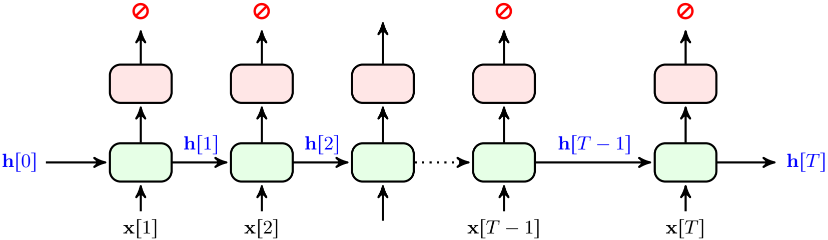
*So, the model for our translator looks like this*



# Basic Translator: *Dataset*

*To train this model, we collect some dataset: in this dataset*

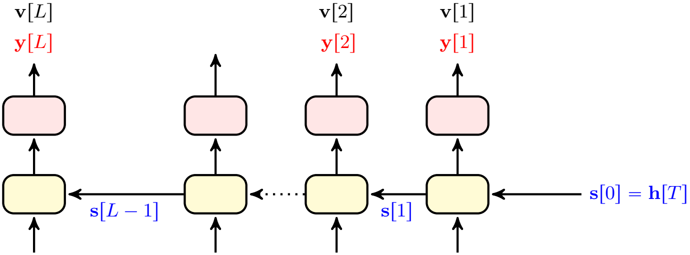
- *We have German sentences*
  - ↳ *We tokenize each sentence and represent it with a sequence* $\mathbf{x}[1:T]$



# Basic Translator: *Dataset*

*To train this model, we collect some dataset: in this dataset*

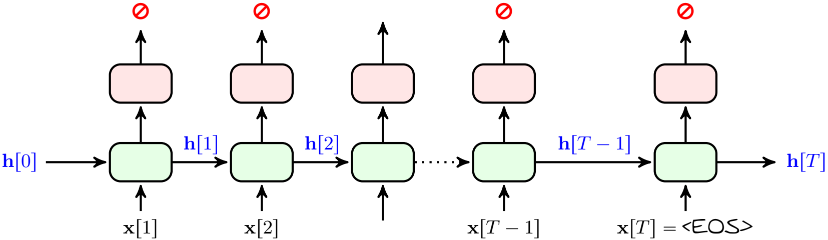
- *Corresponding to each German sentence, we have the English translation*
  - ↳ *We tokenize it as well and represent it with a sequence* $\mathbf{v}[1:L]$
  - ↳ $L$ *and* $T$ *are not of the same length*



# Basic Translator: *Training*

*Say we want to train it for sample sentence: we start with forward pass*

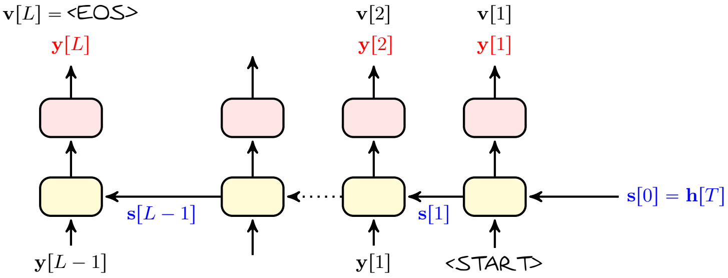
- *We pass the tokens through time forward till we arrive at* <EOS>
  - ↳ *At this point we have* $\mathbf{h}[T]$ *at the output of encoder*
- *We have already computed all variables inside this encoder*
  - ↳ *We need them in them backward pass*



# Basic Translator: *Training*

*Say we want to train it for sample sentence: we start with forward pass*

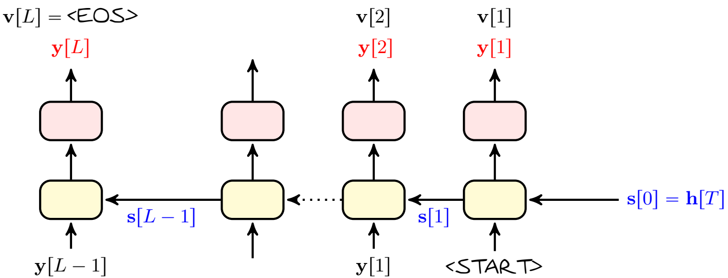
- *We initiate the decoder with* $\mathbf{s}[0] = \mathbf{h}[T]$
  - ↳ *We could also give token of* <START> *as first input*
  - ↳ *We can give* $\mathbf{y}[\ell - 1]$ *as the input at time* $\ell$
- *We continue till we get to label* <EOS>



# Basic Translator: *Training*

*Say we want to train it for sample sentence: now we pass backward*

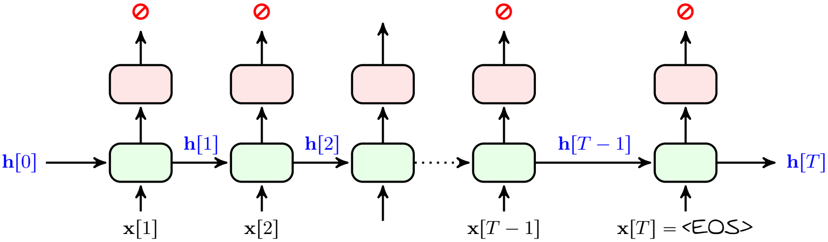
- *We compute loss by between the labels and outputs*
  - ↳ *We aggregate cross-entropy losses over time*
- *We backpropagate through time*
  - ↳ *We get to* $\nabla_{\mathbf{s}[0]} \hat{R} = \nabla_{\mathbf{h}[T]} \hat{R}$



# Basic Translator: *Training*

*Say we want to train it for sample sentence: now we pass backward*

- *We have* $\nabla_{\mathbf{h}[T]} \hat{R}$
  - ↳ *We backpropagate again over time*
- *We update all the weights and start a new round*

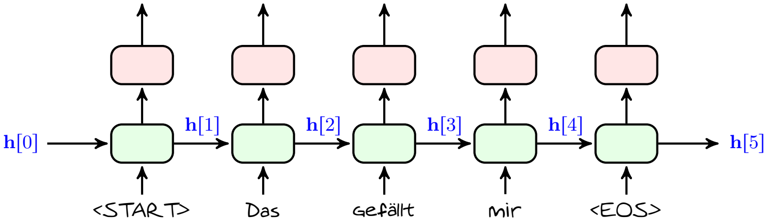


# Basic Translator: *Inference*

*Say we have trained this model and we want to use it to translate*

*"Das gefällt mir"* $\rightsquigarrow$ *I like it*

*Let's start with encoding*

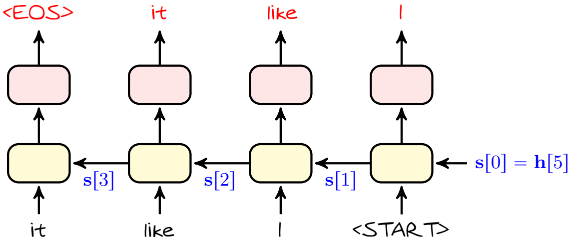


# Basic Translator: *Inference*

*Say we have trained this model and we want to use it to translate*

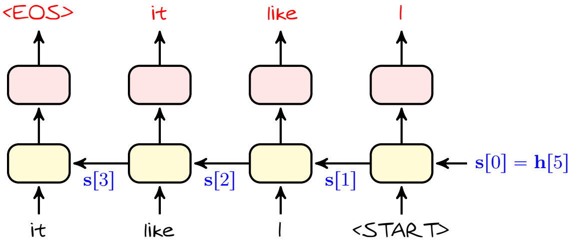
*"Das gefällt mir"* $\rightsquigarrow$ *I like it*

*Now, for decoding we start with* $\mathbf{h}[5]$



# Basic Translator: *Inference*

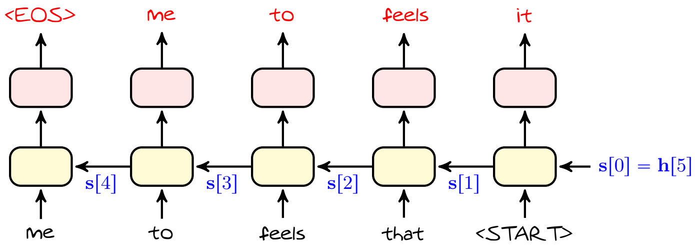
*If we are lucky, the token of word "I" has highest probability in* $\ell = 1$




*and probably the RNN keeps generating a correct sentence*

# Basic Translator: *Inference*

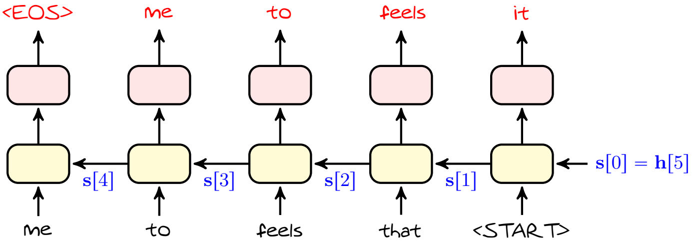
*If we are unlucky, another token might be slightly higher in probability at* $\ell = 1$




*In this case, a small mistake can lead to a sequence of mistakes, e.g., say*

# Basic Translator: *Inference*

*If we are unlucky, another token might be slightly higher in probability at* $\ell = 1$

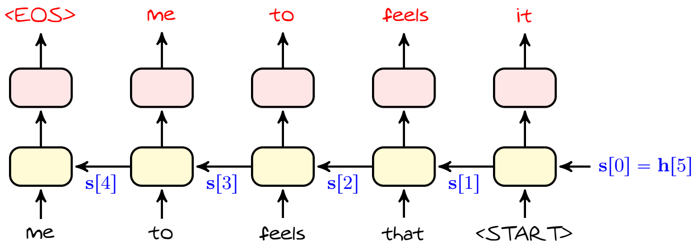



*In this case, a small mistake can lead to a sequence of mistakes, e.g., say*

- *word* "it" *has slightly higher chance at the beginning*
  - ↳ *e.g.,* "it": $0.121$ *and* "I": $0.119$*, thus, we choose it as the first word*

# Basic Translator: *Inference*

*If we are unlucky, another token might be slightly higher in probability at $\ell = 1$*

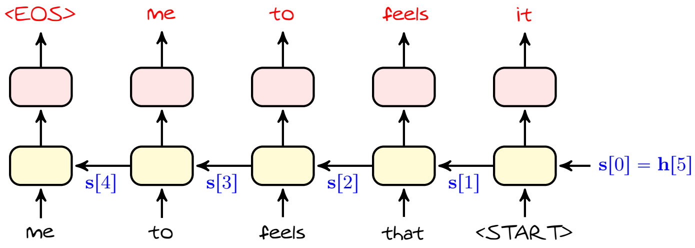



*In this case, a small mistake can lead to a sequence of mistakes, e.g., say*

- *word "it" has slightly higher chance at the beginning*
  - ↳ *e.g., "it": $0.121$ and "I": $0.119$, thus, we choose it as the first word*
- *since we chose "it", RNN needs to generate the next word accordingly*
  - ↳ *with "it" as input: "feels" has higher chances than "like"*

# Basic Translator: *Inference via Decoding*

*If we are unlucky, another token might be slightly higher in probability at* $\ell = 1$



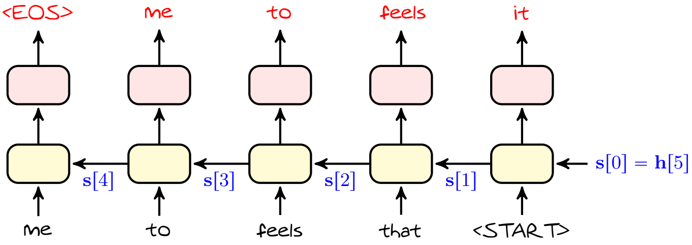

## Decoding

*To avoid error propagation, a better idea is to find the sequence with highest probability, i.e., sequence* $\mathbf{v}^\star[1:L]$ *that has highest conditional probability*

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

# Basic Translator: *Inference via Decoding*

*If we are unlucky, another token might be slightly higher in probability at* $\ell = 1$




Intuitively, this means: *we do not classify in each time*

- *We wait till the sentence is over*
- *We consider all possible combinations*
- *We find the one which has highest conditional probability*
  - ↳ *We need to compute this conditional probability*

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right) = p\left(\mathbf{v}[1:L] | \mathbf{h}[T]\right)$$

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$
\begin{aligned}
p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right) &= p\left(\mathbf{v}[1:L] | \mathbf{h}[T]\right) \\
&= \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell] | \mathbf{h}[T], \mathbf{v}[1:\ell-1]\right) \rightsquigarrow \text{Bayes chain rule}
\end{aligned}
$$

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$
\begin{aligned}
p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right) &= p\left(\mathbf{v}[1:L] | \mathbf{h}[T]\right) \\
&= \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell] | \mathbf{h}[T], \mathbf{v}[1:\ell-1]\right) \rightsquigarrow \text{Bayes chain rule} \\
&\propto \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell] | \mathbf{s}[\ell-1], \mathbf{y}[\ell-1]\right)
\end{aligned}
$$

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$
\begin{aligned}
p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right) &= p\left(\mathbf{v}[1:L] | \mathbf{h}[T]\right) \\
&= \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell] | \mathbf{h}[T], \mathbf{v}[1:\ell-1]\right) \rightsquigarrow \text{Bayes chain rule} \\
&\propto \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell] | \mathbf{s}[\ell-1], \mathbf{y}[\ell-1]\right) \propto \prod_{\ell=1}^{L} y_{v[\ell]}[\ell]
\end{aligned}
$$

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$p\left(\mathbf{v}[1:L]|\mathbf{x}[1:T]\right) = p\left(\mathbf{v}[1:L]|\mathbf{h}[T]\right)$$
$$= \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell]|\mathbf{h}[T], \mathbf{v}[1:\ell-1]\right) \rightsquigarrow \text{Bayes chain rule}$$
$$\propto \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell]|\mathbf{s}[\ell-1], \mathbf{y}[\ell-1]\right) \propto \prod_{\ell=1}^{L} y_{v[\ell]}[\ell]$$

$y_{v[\ell]}[\ell]$ *means the entry of* $\mathbf{y}[\ell]$ *that corresponds to the class of* $\mathbf{v}[\ell]$, *e.g.,*

$$\mathbf{v}[\ell] = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{y}[\ell] = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.4 \\ 0.2 \end{bmatrix}$$

# Basic Translator: *Inference via Decoding*

*Let's try finding this probability first*

$$p\left(\mathbf{v}[1:L]|\mathbf{x}[1:T]\right) = p\left(\mathbf{v}[1:L]|\mathbf{h}[T]\right)$$
$$= \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell]|\mathbf{h}[T], \mathbf{v}[1:\ell-1]\right) \rightsquigarrow \text{Bayes chain rule}$$
$$\propto \prod_{\ell=1}^{L} p\left(\mathbf{v}[\ell]|\mathbf{s}[\ell-1], \mathbf{y}[\ell-1]\right) \propto \prod_{\ell=1}^{L} y_{v[\ell]}[\ell]$$

$y_{v[\ell]}[\ell]$ *means the entry of* $\mathbf{y}[\ell]$ *that corresponds to the class of* $\mathbf{v}[\ell]$, *e.g.,*

$$\mathbf{v}[\ell] = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{y}[\ell] = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.4 \\ 0.2 \end{bmatrix} \rightsquigarrow y_{v[\ell]}[\ell] = 0.3$$

# Basic Translator: *Inference via Decoding*

*Since we are more comfortable with sum, we can use log-likelihood*

$$\mathbf{v}^\star[1:L] = \operatorname*{argmax}_{\mathbf{v}[1:L]} \log p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right)$$

$$= \operatorname*{argmax}_{\mathbf{v}[1:L]} \sum_{\ell=1}^{L} \log y_{v[\ell]}[\ell]$$

# Basic Translator: *Inference via Decoding*

*Since we are more comfortable with sum, we can use log-likelihood*

$$\mathbf{v}^\star[1:L] = \operatorname*{argmax}_{\mathbf{v}[1:L]} \log p\left(\mathbf{v}[1:L] | \mathbf{x}[1:T]\right)$$

$$= \operatorname*{argmax}_{\mathbf{v}[1:L]} \sum_{\ell=1}^{L} \log y_{v[\ell]}[\ell]$$

+ *But, is it feasible to find the sequence with highest sum?*
- No! *Since we deal here with an exponentially large case*

*For a vocabulary with $D$ words in it, the number of possible sequences is*

$$(D+2)^L$$

# Basic Translator: *Inference via Decoding*

*In practice, however, we know that many of those combinations are invalid, e.g.,*

*"I are potato" is not an invalid English sentence!*

*So we can use sub-optimal search methods to find a good sequence*

- *We do not necessarily hit the best sequence*
  - ↳ *But, for fair length, we can be close*
  - ↳ *We will be definitely better that naive instant decoding*

# Basic Translator: *Inference via Decoding*

*In practice, however, we know that many of those combinations are invalid, e.g.,*

*"I are potato" is not an invalid English sentence!*

*So we can use sub-optimal search methods to find a good sequence*

- *We do not necessarily hit the best sequence*
  - ↳ *But, for fair length, we can be close*
  - ↳ *We will be definitely better that naive instant decoding*
- *Most famous approach is beam search via top-$k$*
  - ↳ *At time step $\ell$ we find $k$ sequences with highest sum log-likelihoods till $\ell$*
  - ↳ *We update top-$k$ sequences by searching among children on sequence tree*
  - ↳ *We finally select the sequence with highest probability among those top-$k$*

# Basic Translator: *Inference via Decoding*

*In practice, however, we know that many of those combinations are invalid, e.g.,*

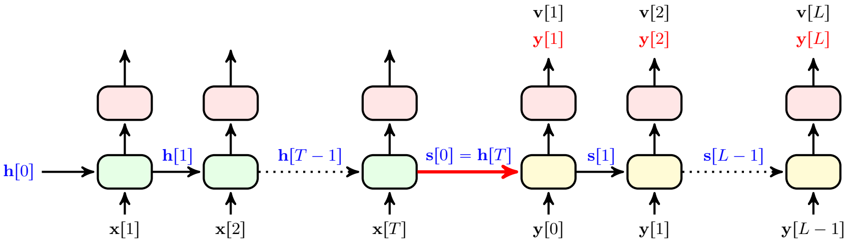
*"I are potato" is not an invalid English sentence!*

*So we can use sub-optimal search methods to find a good sequence*

- *We do not necessarily hit the best sequence*
  - ↳ *But, for fair length, we can be close*
  - ↳ *We will be definitely better that naive instant decoding*
- *Most famous approach is beam search via top-$k$*
  - ↳ *At time step $\ell$ we find $k$ sequences with highest sum log-likelihoods till $\ell$*
  - ↳ *We update top-$k$ sequences by searching among children on sequence tree*
  - ↳ *We finally select the sequence with highest probability among those top-$k$*
- *Since decoded sequences can be of different length, we usually maximize normalized log-likelihood*

$$\mathbf{v}^\star[1:L] = \underset{\mathbf{v}[1:L]}{\operatorname{argmax}} \frac{1}{L} \sum_{\ell=1}^{L} \log y_{v[\ell]}[\ell]$$

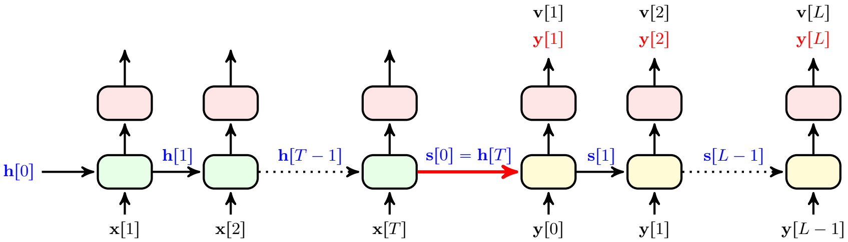
# Translating Long Texts




With standard RNN encoder and decoder: *model works up to some length*

- *The decoder could get lost at some point*
  - ↳ *It could miss the case of the word or its order*

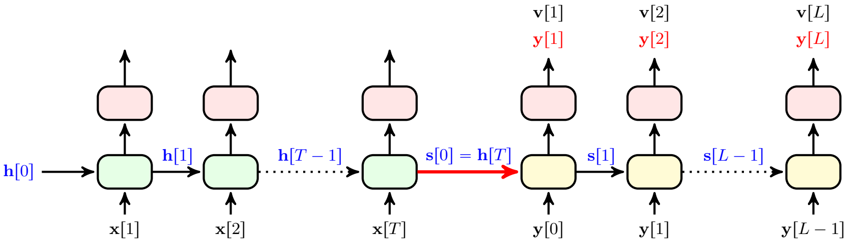
# Translating Long Texts




With standard RNN encoder and decoder: *model works up to some length*

- *The decoder could get lost at some point*
  - ↳ *It could miss the case of the word or its order*

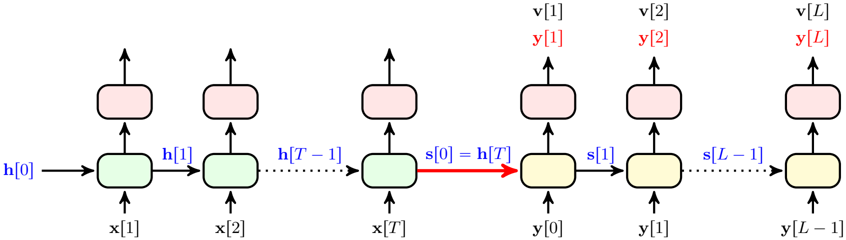
# Translating Long Texts




With standard RNN encoder and decoder: *model works up to some length*

- *The decoder could get lost at some point*
  - ↳ *It could miss the case of the word or its order*
  - ↳ *Imagine it wants to translate:*
    *"Ich habe den Apfel genommen, über den wir beim letzten Mal gesprochen haben"* ⇝ *"I took the apple that we talked about last time"*

# Information Bottleneck



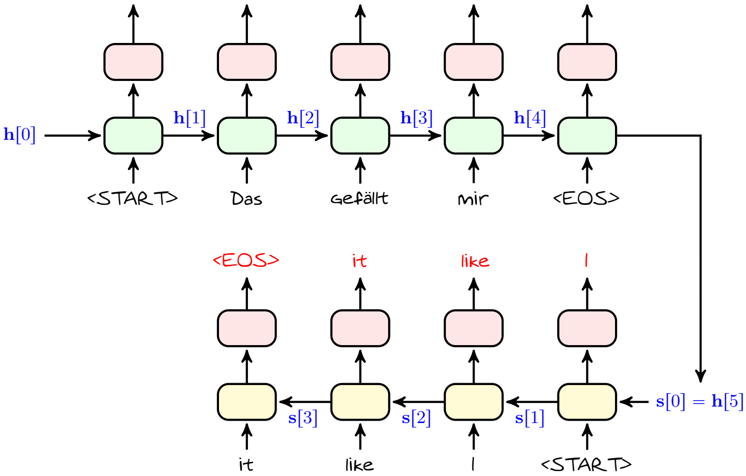

The source of this problem is that

*decoder gets all its information through a single bottleneck*

*This problem is known as information bottleneck problem*

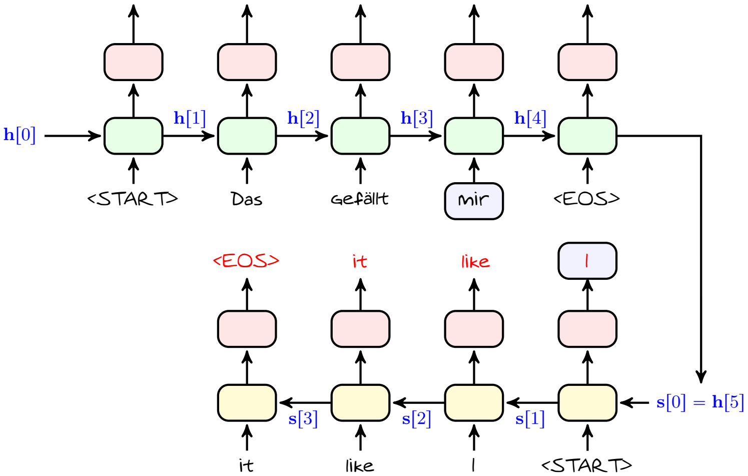
# Attention: *Finding Relevant Input*

*Let's look back at our translator:*



# Attention: *Finding Relevant Input*

*Let's look back at our translator: "I" is given in translation because of "mir"*



# Attention: *Finding Relevant Input*

If we could tell the decoder, it could

*use hidden state of time $t = 4$ to generate its output word*

*We could intuitively say that in this case*

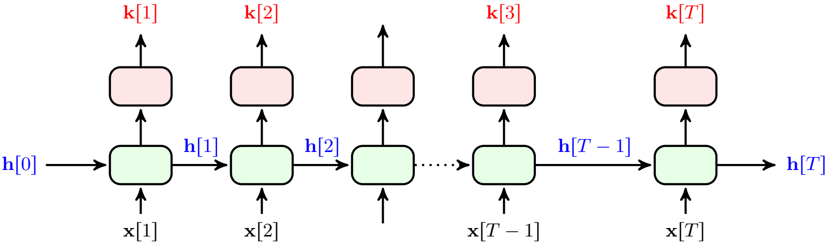
$$\mathbf{y}[1] = f\left(\mathbf{s}[0], \text{<START>}, \text{mir}\right)$$

*which is more likely to be "I" as compared to the case in which*

$$\mathbf{y}[1] = f\left(\mathbf{s}[0], \text{<START>}\right)$$

*Attention mechanism formulates mathematically this intuitive idea*

# Attention: *Generating Keys*



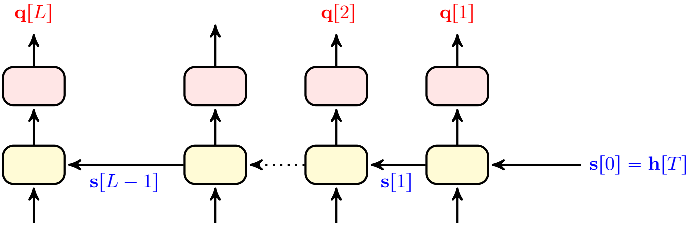

*With attention, we generate a key for each time step at encoding*

- *Keys are learned by an arbitrary layer, e.g.,*

$$\mathbf{k}[t] = \sigma\left(\mathbf{W}_{\mathrm{k}}\mathbf{h}[t]\right)$$

- *We can even use multiple layer*
  - ↳ *Not really needed in most applications*

# Attention: *Generating Queries*



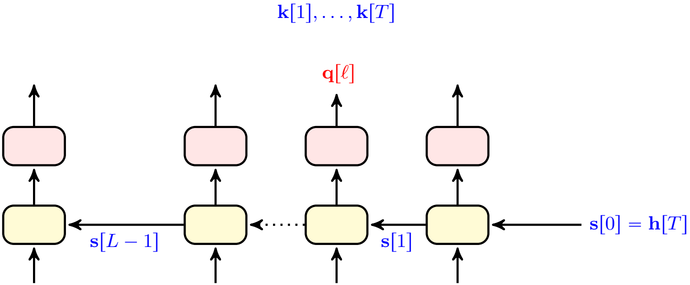

*We next generate a queries for each time step at decoding*

- *Queries are again learned by an arbitrary layer, e.g.,*

$$\mathbf{q}[t] = \sigma\left(\mathbf{W}_{\mathrm{q}}\mathbf{s}[t]\right)$$

- *Note that it's in general a new learnable layer*

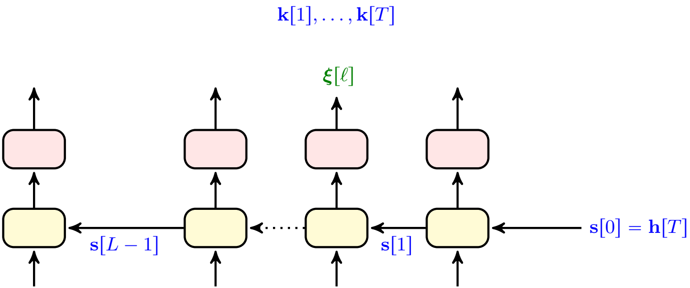
# Attention: *Score at Time $\ell$*




*During decoding, we can find score of query at time $\ell$ by comparing it to all keys*

$$\xi_t[\ell] = \mathbf{k}^{\mathsf{T}}[t]\mathbf{q}[\ell] \rightsquigarrow \boldsymbol{\xi}[\ell] = \begin{bmatrix} \xi_1[\ell] \\ \vdots \\ \xi_T[\ell] \end{bmatrix}$$

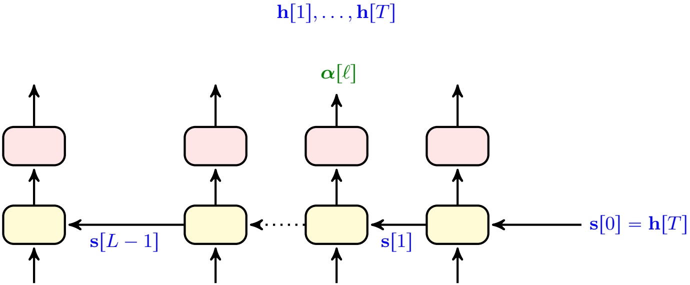
# Attention: *Score at Time $\ell$*




*We then convert this score to the chance of $\mathbf{v}[\ell]$ being related to input entry $\mathbf{x}[t]$: we could use softmax*

$$\boldsymbol{\alpha}[\ell] = \mathrm{Soft}_{\max}\left(\boldsymbol{\xi}_\ell\right) = \begin{bmatrix} \alpha_1[\ell] \\ \vdots \\ \alpha_T[\ell] \end{bmatrix} \rightsquigarrow \alpha_t[\ell] \equiv \text{chance of } \mathbf{v}[\ell] \text{ being related to } \mathbf{x}[t]$$

# Attention: *Attention Feature*

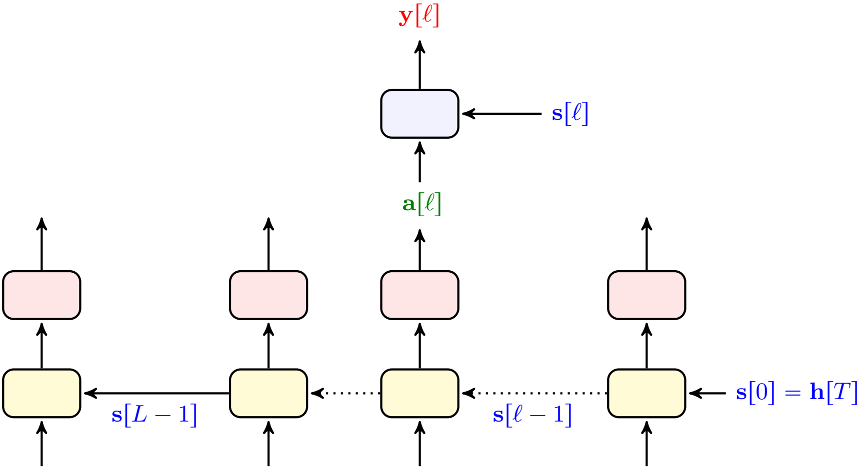



*We now use these probabilities to make a vector of attention features*

$$\mathbf{a}[\ell] = \sum_{t=1}^{T} \alpha_t[\ell]\mathbf{h}[t]$$

# Attention: *Computing Output*

*We now use attention features to build the output sequence*



# Attention: *Computing Output*

*We use attention features to build the output sequence*

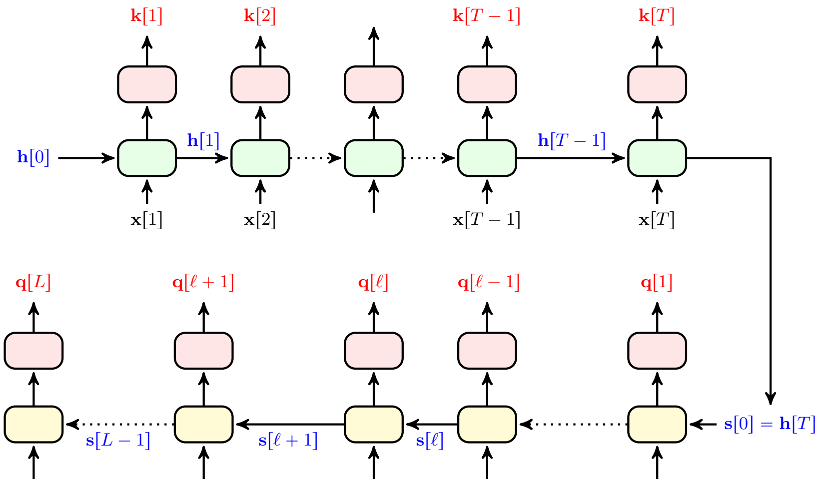
- *This can be any layer, as in standard RNN, e.g.,*

$$\mathbf{y}[\ell] = \mathit{Soft}_{\max}\left(\mathbf{W}_{\rm out}\mathbf{s}[\ell] + \mathbf{W}_{\rm att}\mathbf{a}[\ell]\right)$$

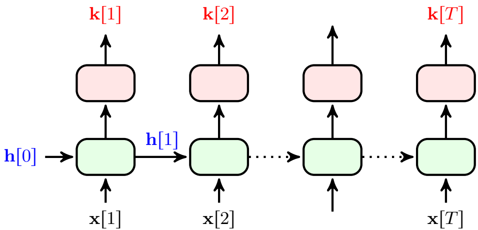
- *We could also use $\mathbf{a}[t]$ as a new state*
  - ↳ *We could combine it with the current state*
  - ↳ *We can pass it to higher layer*

*It turns out that attention can hugely help in practice!*

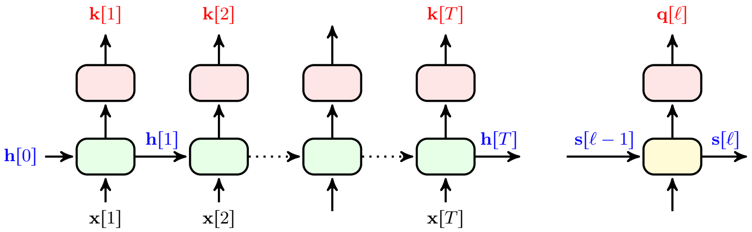
# Attention: *End to End Architecture*



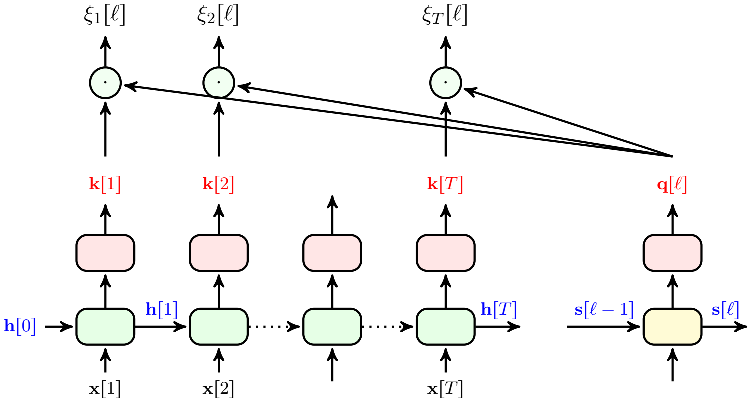
# Attention: *End to End Architecture*



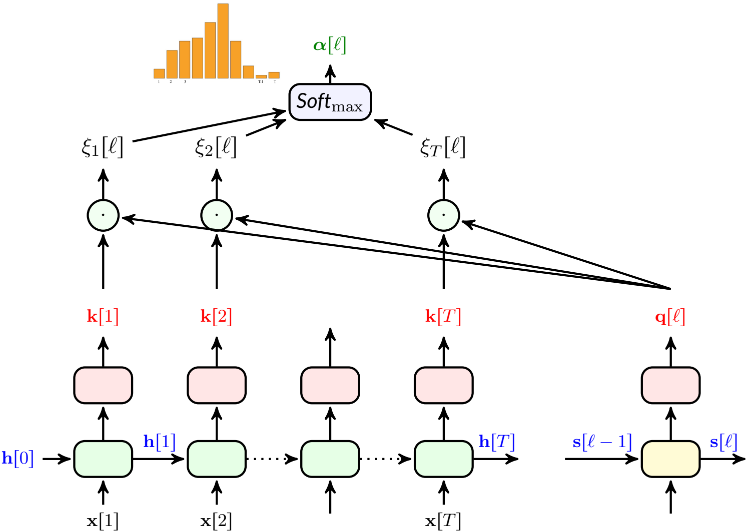
# Attention: *End to End Architecture*



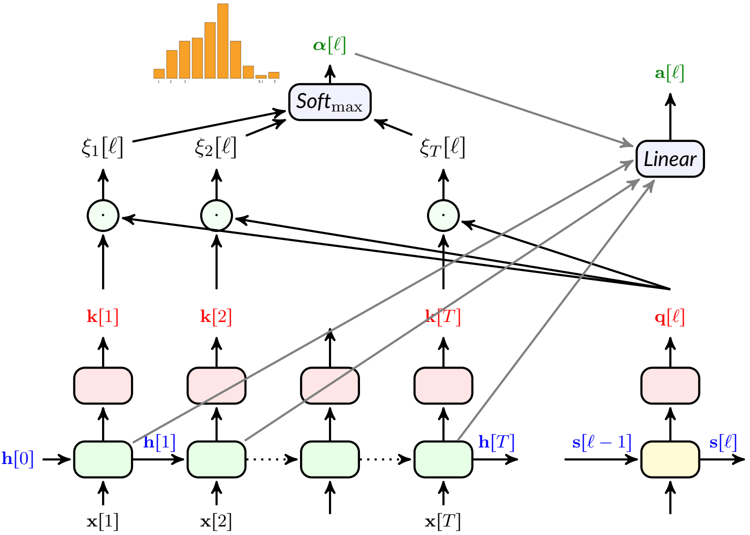
# Attention: *End to End Architecture*



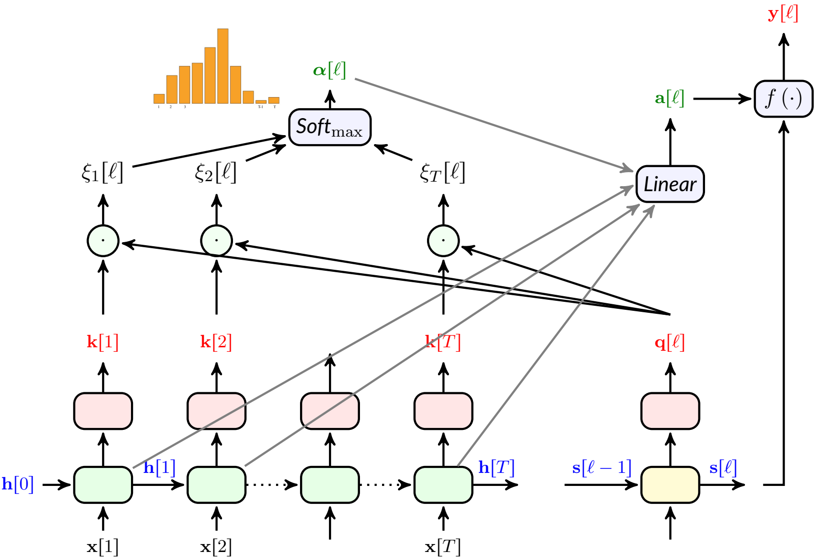
# Attention: *End to End Architecture*



# Attention: *End to End Architecture*



# Attention: *End to End Architecture*



# Attention: *Training*

Let's look at training: *assume we want to train it over a single pair of sequences*

- *We have a sequence of inputs* $\mathbf{x}[t]$
  - ↳ *For instance a German sentence*
- *We have a sequence of labels* $\mathbf{v}[\ell]$
  - ↳ *For instance the English translation*
  - ↳ *We can compare each output* $\mathbf{y}[\ell]$ *with its label*

# Attention: *Training*

Let's look at training: *assume we want to train it over a single pair of sequences*

- *We have a sequence of inputs* $\mathbf{x}[t]$
  - ↳ *For instance a German sentence*
- *We have a sequence of labels* $\mathbf{v}[\ell]$
  - ↳ *For instance the English translation*
  - ↳ *We can compare each output* $\mathbf{y}[\ell]$ *with its label*

*We start with forward pass*

- *Pass forward through the encoder*
  - ↳ *Also generate the keys*
- *Pass the encoder's state and its keys to the decoder*
- *Pass forward through the decoder*
  - ↳ *Generate queries and compare them to the keys*
  - ↳ *Compute attention and the outputs*

# Attention: *Training*

Let's look at training: *assume we want to train it over a single pair of sequences*

- *We have a sequence of inputs* $\mathbf{x}[t]$
  - ↳ *For instance a German sentence*
- *We have a sequence of labels* $\mathbf{v}[\ell]$
  - ↳ *For instance the English translation*
  - ↳ *We can compare each output* $\mathbf{y}[\ell]$ *with its label*

*We start with forward pass*

- *Pass forward through the encoder*
  - ↳ *Also generate the keys*
- *Pass the encoder's state and its keys to the decoder*
- *Pass forward through the decoder*
  - ↳ *Generate queries and compare them to the keys*
  - ↳ *Compute attention and the outputs*
- *Compute loss by aggregating* $\mathcal{L}(\mathbf{y}[\ell], \mathbf{v}[\ell])$

# Attention: *Training*

*Now we should pass backward*

- *Pass backward through the output layer*
  - ↳ *Compute* $\nabla_{\mathbf{a}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{s}[\ell]} \hat{R}[\ell]$

# Attention: *Training*

*Now we should pass backward*

- *Pass backward through the output layer*
  ↳ *Compute* $\nabla_{\mathbf{a}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{s}[\ell]} \hat{R}[\ell]$
- *Pass backward through the attention layer*
  ↳ *Compute* $\nabla_{\boldsymbol{\alpha}[\ell]} \hat{R}[\ell]$, $\nabla_{\boldsymbol{\xi}[\ell]} \hat{R}[\ell]$, $\nabla_{\mathbf{k}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{q}[\ell]} \hat{R}[\ell]$

# Attention: *Training*

*Now we should pass backward*

- *Pass backward through the output layer*
  - ↳ *Compute* $\nabla_{\mathbf{a}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{s}[\ell]} \hat{R}[\ell]$
- *Pass backward through the attention layer*
  - ↳ *Compute* $\nabla_{\boldsymbol{\alpha}[\ell]} \hat{R}[\ell]$, $\nabla_{\boldsymbol{\xi}[\ell]} \hat{R}[\ell]$, $\nabla_{\mathbf{k}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{q}[\ell]} \hat{R}[\ell]$
- *Pass backward through time at the decoder*

$$\nabla_{\mathbf{s}[\ell-1]} \hat{R}[\ell] = \nabla_{\mathbf{s}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{s}[\ell] + \nabla_{\mathbf{q}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{q}[\ell]$$

# Attention: *Training*

*Now we should pass backward*

- *Pass backward through the output layer*
  - ↳ *Compute* $\nabla_{\mathbf{a}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{s}[\ell]} \hat{R}[\ell]$
- *Pass backward through the attention layer*
  - ↳ *Compute* $\nabla_{\boldsymbol{\alpha}[\ell]} \hat{R}[\ell]$, $\nabla_{\boldsymbol{\xi}[\ell]} \hat{R}[\ell]$, $\nabla_{\mathbf{k}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{q}[\ell]} \hat{R}[\ell]$
- *Pass backward through time at the decoder*

$$\nabla_{\mathbf{s}[\ell-1]} \hat{R}[\ell] = \nabla_{\mathbf{s}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{s}[\ell] + \nabla_{\mathbf{q}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{q}[\ell]$$

- *Pass backward through time at the encoder*

$$\begin{aligned} \nabla_{\mathbf{h}[t-1]} \hat{R}[\ell] = \nabla_{\mathbf{h}[t]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{h}[t] + \nabla_{\mathbf{k}[t]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{q}[t] \\ + \nabla_{\mathbf{a}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{a}[\ell] \end{aligned}$$

# Attention: *Training*

*Now we should pass backward*

- *Pass backward through the output layer*
  - ↳ *Compute* $\nabla_{\mathbf{a}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{s}[\ell]} \hat{R}[\ell]$
- *Pass backward through the attention layer*
  - ↳ *Compute* $\nabla_{\boldsymbol{\alpha}[\ell]} \hat{R}[\ell]$, $\nabla_{\boldsymbol{\xi}[\ell]} \hat{R}[\ell]$, $\nabla_{\mathbf{k}[\ell]} \hat{R}[\ell]$ *and* $\nabla_{\mathbf{q}[\ell]} \hat{R}[\ell]$
- *Pass backward through time at the decoder*

$$\nabla_{\mathbf{s}[\ell-1]} \hat{R}[\ell] = \nabla_{\mathbf{s}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{s}[\ell] + \nabla_{\mathbf{q}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{s}[\ell-1]} \mathbf{q}[\ell]$$

- *Pass backward through time at the encoder*

$$\begin{aligned}\nabla_{\mathbf{h}[t-1]} \hat{R}[\ell] = \nabla_{\mathbf{h}[t]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{h}[t] &+ \nabla_{\mathbf{k}[t]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{q}[t] \\ &+ \nabla_{\mathbf{a}[\ell]} \hat{R}[\ell] \circ \nabla_{\mathbf{h}[t-1]} \mathbf{a}[\ell]\end{aligned}$$

- *Aggregate over* $\ell$*, update all weights and go for the next round*

# Attention: *Final Remarks*

Attention is a strong mechanism: *it is the key building block of transformers*

- *The described architecture is the vanilla form*
  - ↳ *We can advance each step to handle various issues*

# Attention: *Final Remarks*

Attention is a strong mechanism: *it is the key building block of transformers*

- *The described architecture is the vanilla form*
  - ↳ *We can advance each step to handle various issues*
  - ↳ *Multi-head attention is a parallelized version of vanilla attention*
    - ↳ *Transformers are built based on multi-head attention*
    - ↳ *Pre-implementation is accessed in* `torch.nn.MultiheadAttention()`

# Attention: *Final Remarks*

Attention is a strong mechanism: *it is the key building block of transformers*

- *The described architecture is the vanilla form*
  - ↳ *We can advance each step to handle various issues*
  - ↳ *Multi-head attention is a parallelized version of vanilla attention*
    - ↳ *Transformers are built based on multi-head attention*
    - ↳ *Pre-implementation is accessed in* `torch.nn.MultiheadAttention()`
- *In general, we can play with keys, queries and output layer*
  - ↳ *We can use more advanced layers to compute keys and queries*
    - ↳ *Linear layers with activation is often enough*

# Attention: *Final Remarks*

Attention is a strong mechanism: *it is the key building block of transformers*

- *The described architecture is the vanilla form*
  - ↳ *We can advance each step to handle various issues*
  - ↳ *Multi-head attention is a parallelized version of vanilla attention*
    - ↳ *Transformers are built based on multi-head attention*
    - ↳ *Pre-implementation is accessed in* `torch.nn.MultiheadAttention()`
- *In general, we can play with keys, queries and output layer*
  - ↳ *We can use more advanced layers to compute keys and queries*
    - ↳ *Linear layers with activation is often enough*
  - ↳ *There are various approaches to use the attention variable* $\mathbf{a}[\ell]$ *at decoder*
    - ↳ *We could simply pass it through an output layer*
    - ↳ *We could concatenate it to the decoder's state*
    - ↳ ...