Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 8

November 29th, 2022

4.3 Gated recurrent units (GRU)

- we have introduced the basics of RNNs, which can better handle sequence data
- however, such techniques may not be sufficient for a wide range of sequence learning problems from nowadays
- for instance, a notable issue in practice is the numerical instability of RNNs
- although we have discussed implementation tricks such as gradient clipping, this issue can be alleviated further with more sophisticated designs of sequence models
- specifically, gated RNNs are much more common in practice

4.3 Gated recurrent units (GRU)

- we will introduce two such widely-used networks, namely gated recurrent units (GRUs) and long short-term memory (LSTM)
- furthermore, we will expand the RNN architecture with a single hidden layer, which has been discussed so far, and describe deep architectures, with multiple hidden layers
- such expansions are frequently adopted in modern recurrent networks

4.3 Gated recurrent units (GRU)

- in Chapter 2, we discussed that long products of matrices can lead to vanishing or exploding gradients
- let us briefly think about what such gradient anomalies mean in practice:
  - We might encounter a situation where an early observation is highly significant for predicting all future observations. Consider the case where the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence. In this case, the influence of the first token is vital. We would like to have some mechanisms for storing vital early information in a *memory cell*. Without such a mechanism, we will have to assign a very large gradient to this observation, since it affects all the subsequent observations.
  - We might encounter situations where some tokens carry no pertinent observation. For instance, when parsing a web page, there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for *skipping* such tokens in the latent state representation.
  - We might encounter situations where there is a logical break between parts of a sequence. For instance, there might be a transition between chapters in a book, or a transition between a bear and a bull market for stocks. In this case, it would be nice to have a means of *resetting* our internal state representation.

- a number of methods have been proposed to address this
- one of the earliest is long short-term memory, proposed in 1997, which we will discuss in Section 4.4
- the gated recurrent unit (GRU), proposed in 2014, is a slightly lighter variant that often offers comparable performance and is significantly faster to compute
- due to its simplicity, we will start with the GRU

4.3 Gated recurrent units (GRU)

- the key distinction between vanilla RNNs and GRUs is that the latter support gating of the hidden state
- this means that we have dedicated mechanisms for when a hidden state should be *updated*, and also when it should be *reset*
- these mechanisms are learned, and they address the concerns listed above
- for instance, if the first token is of great importance, we will learn not to update the hidden state after the first observation
- likewise, we will learn to skip irrelevant temporary observations
- lastly, we will learn to reset the latent state whenever needed; we discuss this in detail below

4.3 Gated recurrent units (GRU)

- the first thing we need to introduce are the *reset gate* and the *update gate*
- we engineer them to be vectors with entries in $(0, 1)$, such that we can perform convex combinations
- for instance, a reset gate would allow us to control how much of the previous state we might still want to remember
- likewise, an update gate would allow us to control how much of the new state is just a copy of the old state

4.3 Gated recurrent units (GRU)

- we begin by engineering these gates
- Figure 5 illustrates the inputs for both the reset and update gates in a GRU, given the input of the current time step and the hidden state of the previous time step
- the outputs of the two gates are given by two fully-connected layers with a sigmoid activation function

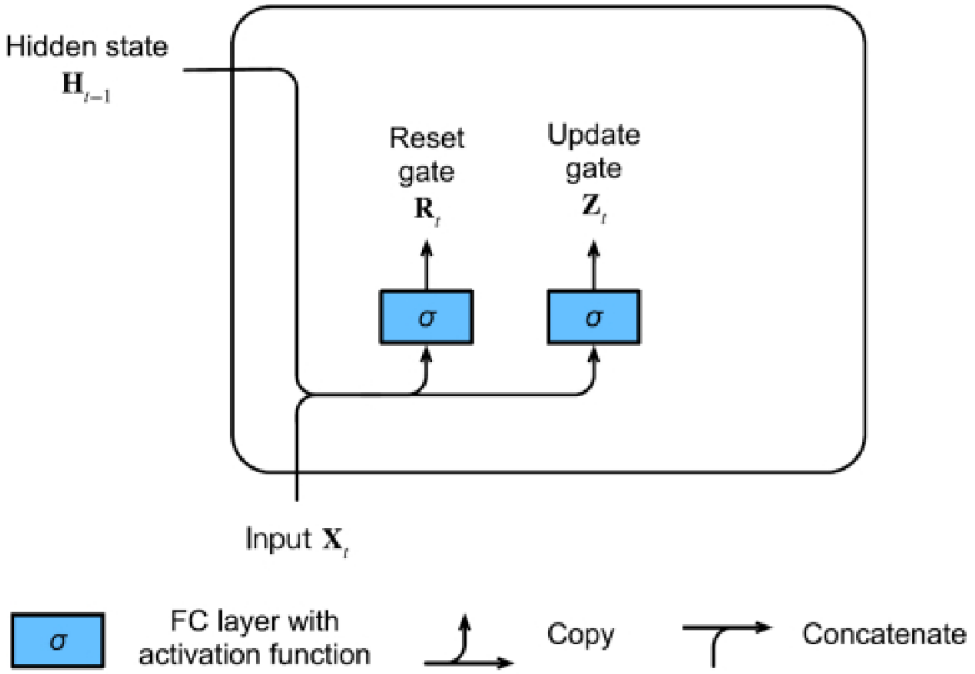

Figure 5: Computing the reset gate and the update gate in a GRU model.

- mathematically, for a given time step t, suppose that the input is a mini-batch $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d) and the hidden state of the previous time step is $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$ (number of hidden units: h)
- then, the *reset gate* $\boldsymbol{R}_t \in \mathbb{R}^{n \times h}$ and *update gate* $\boldsymbol{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\begin{aligned} \boldsymbol{R}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hr} + \boldsymbol{b}_r), \\ \boldsymbol{Z}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xz} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hz} + \boldsymbol{b}_z), \end{aligned}$$

  where $\boldsymbol{W}_{xr}, \boldsymbol{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hr}, \boldsymbol{W}_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters, and $\boldsymbol{b}_r, \boldsymbol{b}_z \in \mathbb{R}^{1 \times h}$ are biases
- note that broadcasting is triggered during the summation
- we use the sigmoid function σ to transform input values to the interval $(0, 1)$

- next, let us integrate the reset gate $\boldsymbol{R}_t$ with the regular latent state updating mechanism in (3)
- it leads to the following *candidate hidden state* $\tilde{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$ at time step t:

$$\tilde{\boldsymbol{H}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xh} + (\boldsymbol{R}_t \odot \boldsymbol{H}_{t-1}) \boldsymbol{W}_{hh} + \boldsymbol{b}_h), \tag{5}$$

  where $\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ are weight parameters, $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$ is the bias, and the symbol $\odot$ is the Hadamard (element-wise) product operator
- here, we use a nonlinearity in the form of tanh to ensure that the values in the candidate hidden state remain in the interval $(-1, 1)$

- the result is a *candidate*, since we still need to incorporate the action of the update gate
- comparing with (3), now, the influence of the previous states can be reduced with the element-wise multiplication of $\boldsymbol{R}_t$ and $\boldsymbol{H}_{t-1}$ in (5)
- whenever the entries in the reset gate $\boldsymbol{R}_t$ are close to 1, we recover a vanilla RNN, such as in (3)
- for all entries of the reset gate $\boldsymbol{R}_t$ that are close to 0, the candidate hidden state is the result of an MLP with $\boldsymbol{X}_t$ as the input
- any pre-existing hidden state is thus *reset* to defaults
- Figure 6 illustrates the computational flow after applying the reset gate

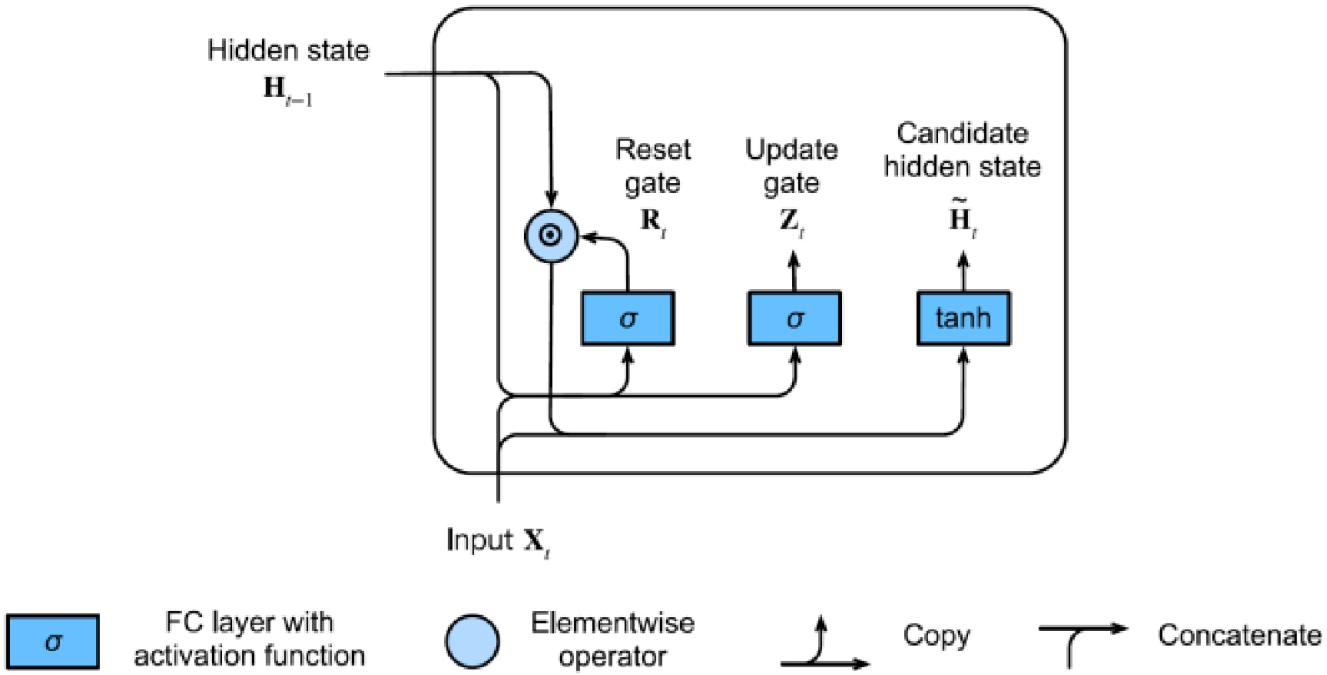

Figure 6: Computing the candidate hidden state in a GRU model.

- finally, we need to incorporate the effect of the update gate $\boldsymbol{Z}_t$
- this determines the extent to which the new hidden state $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ is just the old state $\boldsymbol{H}_{t-1}$, and by how much the new candidate state $\tilde{\boldsymbol{H}}_t$ is used
- the update gate $\boldsymbol{Z}_t$ can be used for this purpose, simply by taking element-wise convex combinations between both $\boldsymbol{H}_{t-1}$ and $\tilde{\boldsymbol{H}}_t$
- this leads to the final update equation for the GRU:

$$\boldsymbol{H}_t = \boldsymbol{Z}_t \odot \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) \odot \tilde{\boldsymbol{H}}_t.$$

- whenever the update gate $\boldsymbol{Z}_t$ is close to 1, we simply retain the old state
- in this case, the information from $\boldsymbol{X}_t$ is essentially ignored, effectively skipping time step t in the dependency chain

- in contrast, whenever $\boldsymbol{Z}_t$ is close to 0, the new latent state $\boldsymbol{H}_t$ approaches the candidate latent state $\tilde{\boldsymbol{H}}_t$
- these designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances
- for instance, if the update gate has been close to 1 for all the time steps of an entire subsequence, the old hidden state at the time step of its beginning will be easily retained and passed to its end, regardless of the length of the subsequence
- Figure 7 illustrates the computational flow after the update gate is in action
- in summary, GRUs have the following two distinguishing features:
  - Reset gates help capture short-term dependencies in sequences.
  - Update gates help capture long-term dependencies in sequences.

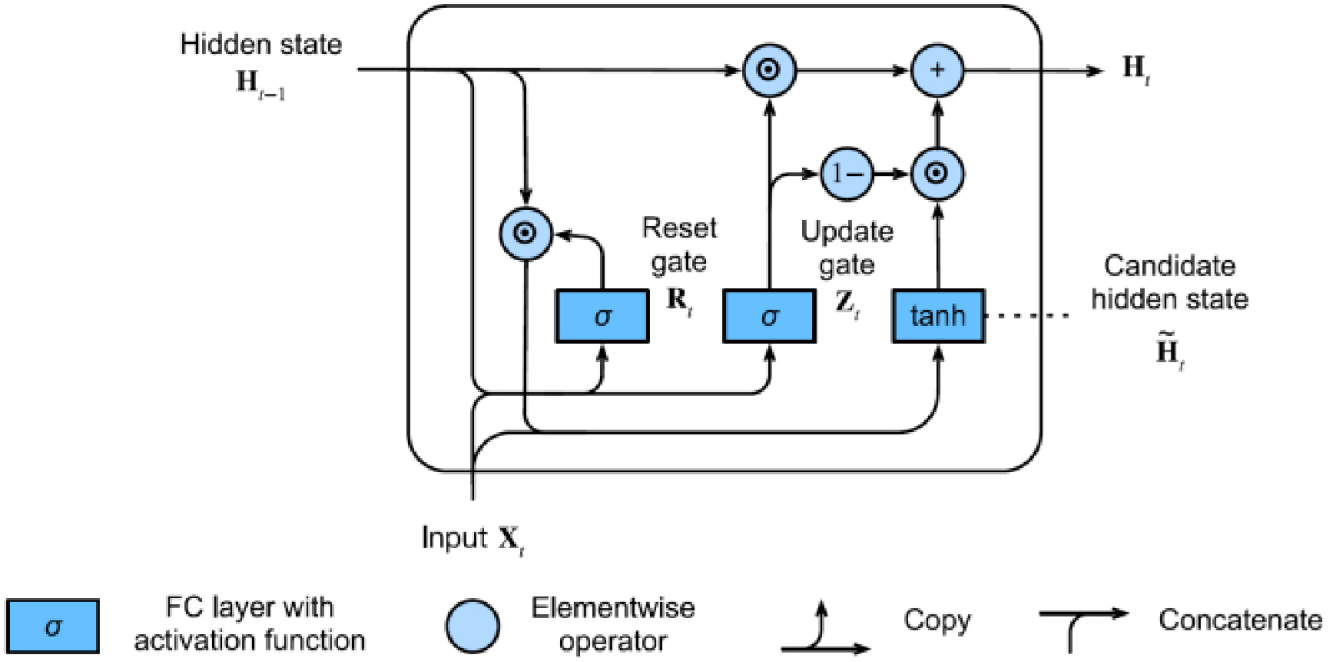

Figure 7: Computing the hidden state in a GRU model.

4.4 Long short-term memory (LSTM)

- the challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time
- one of the earliest approaches to address this was the *long short-term memory* (*LSTM*)
- it shares many of the properties of the GRU
- interestingly, LSTMs have a slightly more complex design than GRUs, but predate GRUs by almost two decades

4.4 Long short-term memory (LSTM)

- arguably LSTM's design is inspired by logic gates of a computer
- LSTM introduces a *memory cell* (or *cell* for short) that has the same shape as the hidden state (some papers consider the memory cell as a special type of hidden state), engineered to record additional information
- to control the memory cell, we need a number of gates
- one gate is needed to read out the entries from the cell; we will refer to this as the *output gate*

4.4 Long short-term memory (LSTM)

- a second gate is needed to decide when to read data into the cell; we refer to this as the *input gate*
- lastly, we need a mechanism to reset the content of the cell, governed by a *forget gate*
- the motivation for such a design is the same as that of GRUs, namely to be able to decide when to remember and when to ignore inputs in the hidden state, via a dedicated mechanism
- let us see how this works in practice

4.4 Long short-term memory (LSTM)

- just like in GRUs, the data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Figure 8
- they are processed by three fully-connected layers with a sigmoid activation function to compute the values of the input, forget, and output gates
- as a result, values of the three gates are in the range of $(0, 1)$

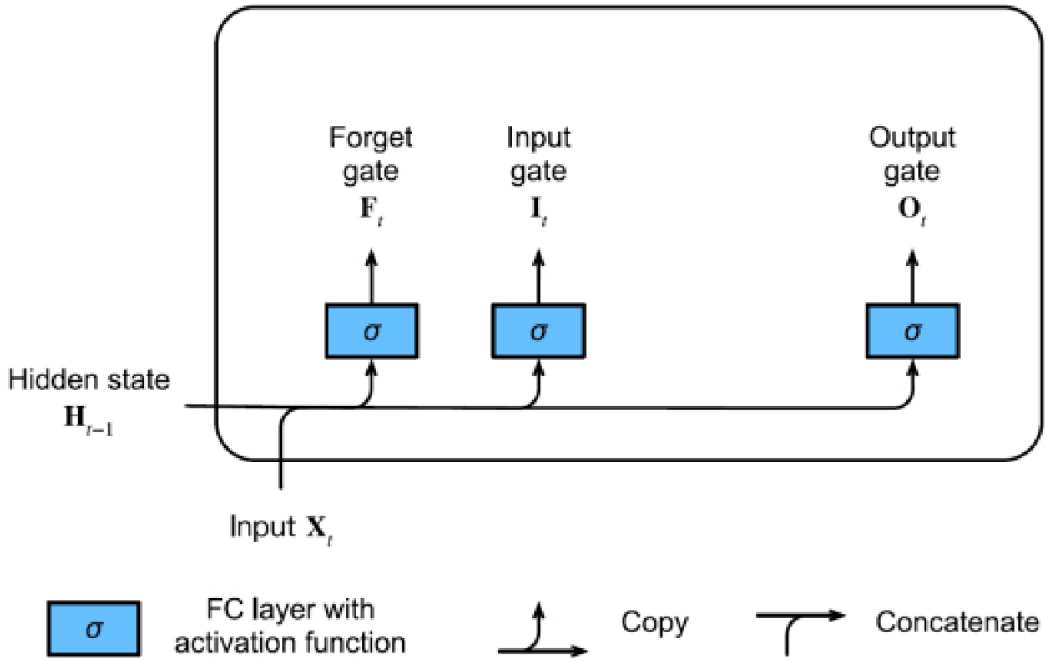

Figure 8: Computing the input gate, the forget gate, and the output gate in an LSTM model.

- mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d
- thus, the input is $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ and the hidden state of the previous time step is $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$
- correspondingly, the gates at time step t are defined as follows: the input gate is $\boldsymbol{I}_t \in \mathbb{R}^{n \times h}$, the forget gate is $\boldsymbol{F}_t \in \mathbb{R}^{n \times h}$, and the output gate is $\boldsymbol{O}_t \in \mathbb{R}^{n \times h}$
- they are calculated as follows:

$$\begin{aligned} \boldsymbol{I}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hi} + \boldsymbol{b}_i), \\ \boldsymbol{F}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xf} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hf} + \boldsymbol{b}_f), \\ \boldsymbol{O}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{ho} + \boldsymbol{b}_o), \end{aligned}$$

where $\boldsymbol{W}_{xi}, \boldsymbol{W}_{xf}, \boldsymbol{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hi}, \boldsymbol{W}_{hf}, \boldsymbol{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters, and $\boldsymbol{b}_i$, $\boldsymbol{b}_f, \boldsymbol{b}_o \in \mathbb{R}^{1 \times h}$ are bias parameters

- next, we design the memory cell
- since we have not specified the action of the various gates yet, we first introduce the *candidate* memory cell $\tilde{\boldsymbol{C}}_t \in \mathbb{R}^{n \times h}$
- its computation is similar to that of the three gates described above, but using a tanh function, with a value range of $(-1, 1)$ as the activation function
- this leads to the following equation at time step t:

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xc} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c),$$

  where $\boldsymbol{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters, and $\boldsymbol{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter
- a quick illustration of the candidate memory cell is shown in Figure 9

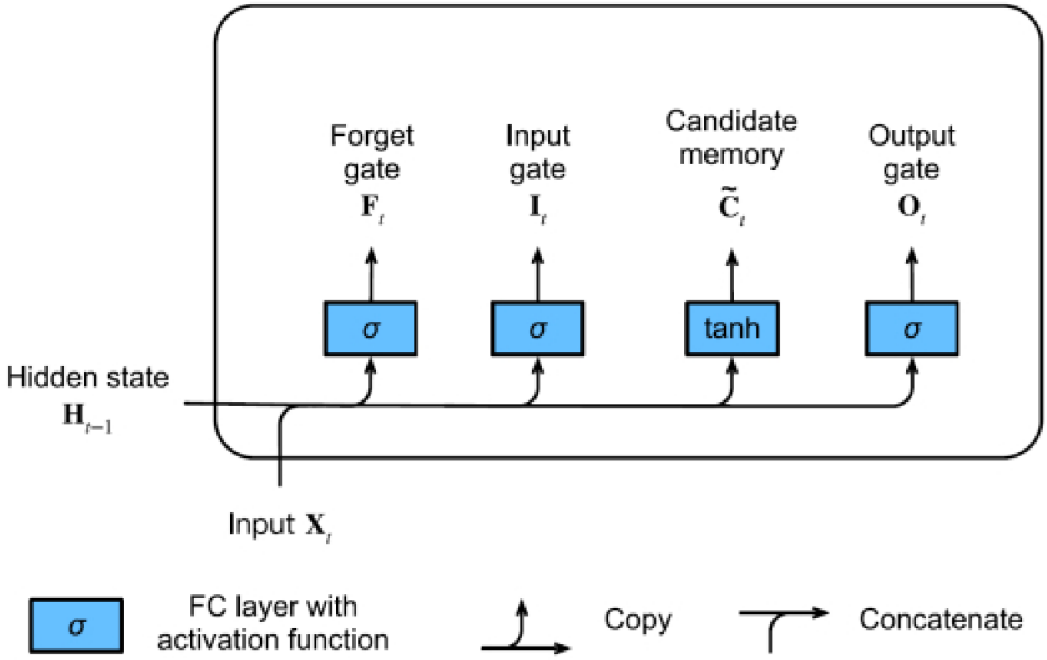

Figure 9: Computing the candidate memory cell in an LSTM model.

- in GRUs, we have a mechanism to govern input and forgetting (or skipping)
- similarly, in LSTMs we have two dedicated gates for such purposes: the input gate $\boldsymbol{I}_t$ governs how much we take new data into account via $\tilde{\boldsymbol{C}}_t$, and the forget gate $\boldsymbol{F}_t$ addresses how much of the old memory cell content $\boldsymbol{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain
- using the same element-wise multiplication trick as before, we arrive at the following update equation:

$$\boldsymbol{C}_t = \boldsymbol{F}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \odot \tilde{\boldsymbol{C}}_t.$$

- if the forget gate is always approximately 1, and the input gate is always approximately 0, the past memory cells $\boldsymbol{C}_{t-1}$ will be saved over time and passed to the current time step
- this design is introduced to alleviate the vanishing gradient problem, and to better capture long range dependencies within sequences
- we thus arrive at the flow diagram in Figure 10

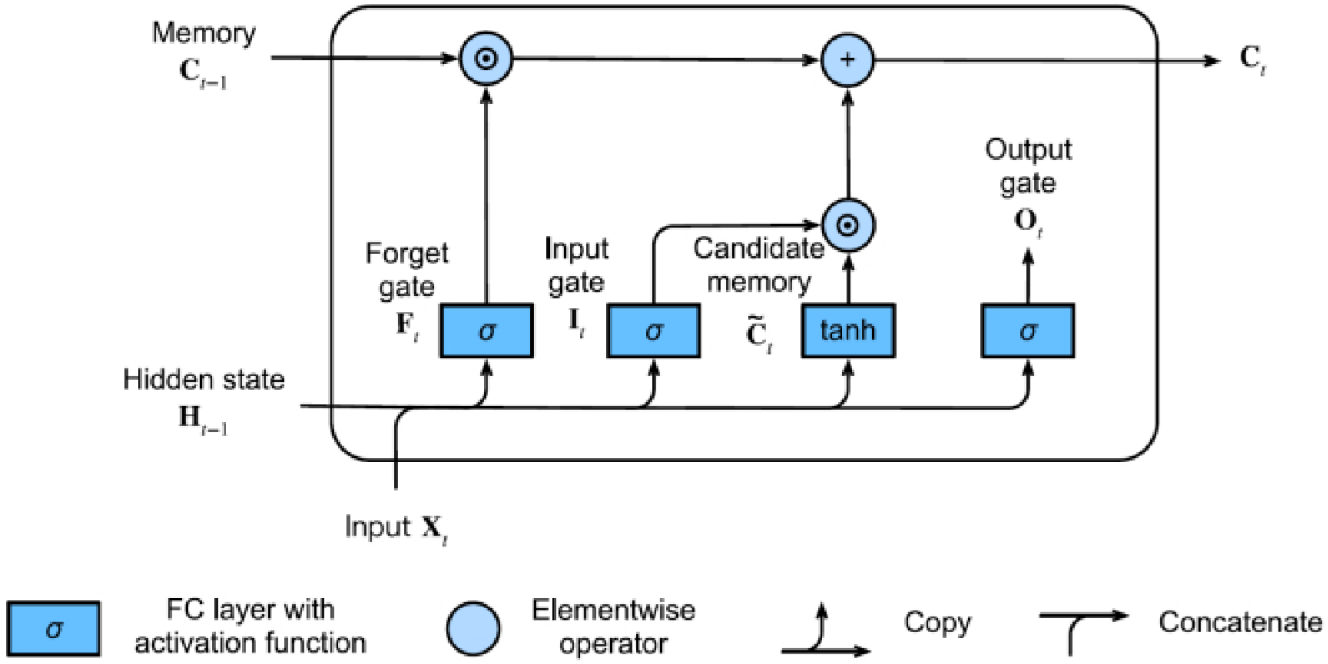

Figure 10: Computing the memory cell in an LSTM model.

- lastly, we need to define how to compute the hidden state $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$; this is where the output gate comes into play
- in LSTM, it is simply a gated version of the tanh of the memory cell
- this ensures that the values of $\boldsymbol{H}_t$ are always in the interval $(-1, 1)$:

$$\boldsymbol{H}_t = \boldsymbol{O}_t \odot \tanh(\boldsymbol{C}_t).$$

- whenever the output gate approximates 1, we effectively pass all memory information through to the predictor, whereas for the output gate close to 0, we retain all the information only within the memory cell, and perform no further processing
- Figure 11 gives a graphical illustration of the data flow

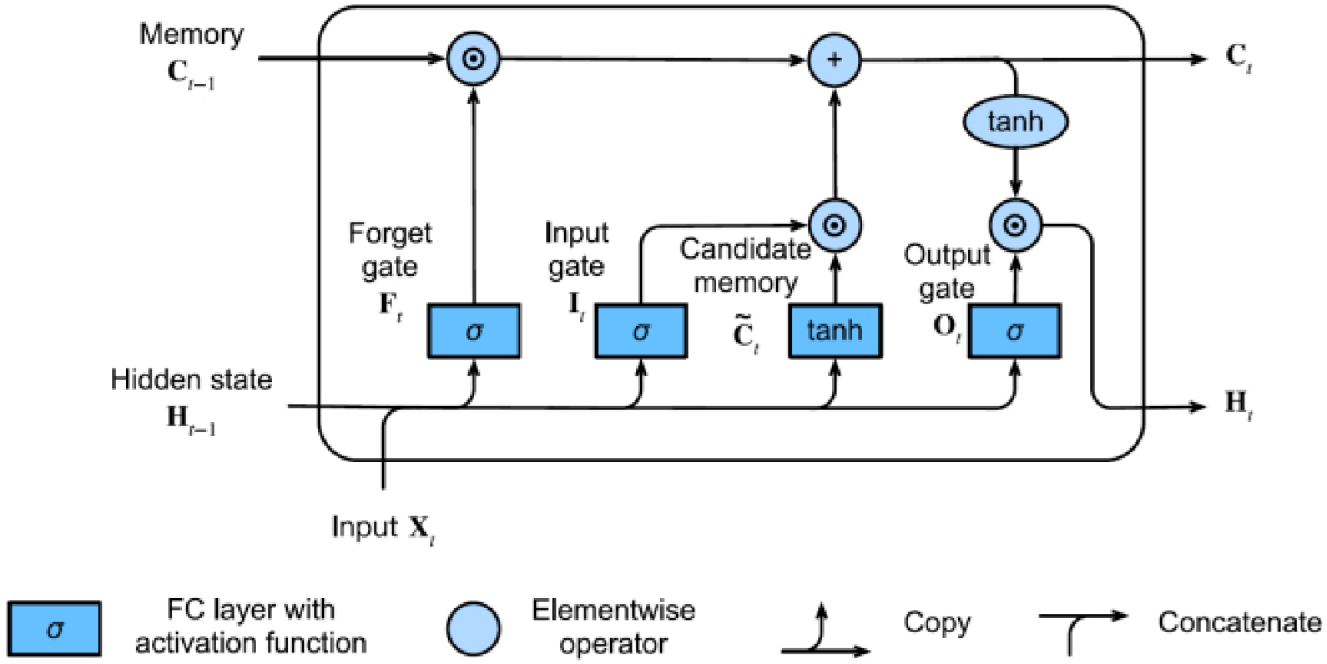

Figure 11: Computing the hidden state in an LSTM model.

4.5 Deep recurrent neural networks

- up to now, we only discussed RNNs with a single hidden layer
- in it, the specific functional form of how latent variables and observations interact is rather arbitrary
- this is not a big problem, as long as we have enough flexibility to model different types of interactions
- with a single layer, however, this can be quite challenging
- in the case of the linear models, we fixed this problem by adding more layers
- within RNNs, this is a bit trickier, since we first need to decide how and where to add extra nonlinearity

4.5 Deep recurrent neural networks

- in fact, we could stack multiple layers of RNNs on top of each other
- this results in a flexible mechanism, due to the combination of several simple layers
- in particular, data might be relevant at different levels of the stack
- for instance, we might want to keep high-level data about financial market conditions (bear or bull market) available, whereas, at a lower level, we only record shorter-term temporal dynamics
- beyond all the above abstract discussion, it is probably easiest to understand the family of models we are interested in by reviewing Figure 12; it describes a deep RNN with L hidden layers
- each hidden state is continuously passed to both the next time step of the current layer, and the current time step of the next layer

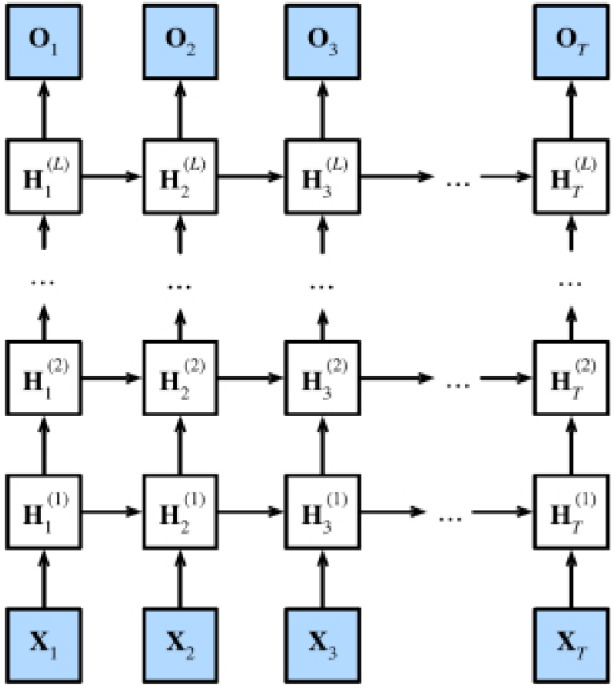

Figure 12: Architecture of a deep RNN.

- we can formalize the functional dependencies within the deep architecture of L hidden layers depicted in Figure 12
- our following discussion focuses primarily on the vanilla RNN model, but it applies to other sequence models, too
- suppose that we have a mini-batch input $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d) at time step t
- at the same time step, let the hidden state of the lth hidden layer ($l = 1, \ldots, L$) be $\boldsymbol{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (number of hidden units: h) and the output layer variable be $\boldsymbol{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q)
- setting $\boldsymbol{H}_t^{(0)} = \boldsymbol{X}_t$, the hidden state of the lth hidden layer that uses the activation function ϕ_l is expressed as follows:

$$\boldsymbol{H}_t^{(l)} = \phi_l(\boldsymbol{H}_t^{(l-1)} \boldsymbol{W}_{xh}^{(l)} + \boldsymbol{H}_{t-1}^{(l)} \boldsymbol{W}_{hh}^{(l)} + \boldsymbol{b}_h^{(l)}), \tag{6}$$

where the weights $\boldsymbol{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ and $\boldsymbol{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, together with the bias $\boldsymbol{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, are the model parameters of the lth hidden layer

- in the end, the calculation of the output layer is only based on the hidden state of the final Lth hidden layer:

$$\boldsymbol{O}_t = \boldsymbol{H}_t^{(L)} \boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

  where the weight $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$ and the bias $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer
- just as with MLPs, the number of hidden layers L and the number of hidden units h are hyperparameters
- in other words, they can be tuned or specified by us
- in addition, we can easily get a deep gated RNN by replacing the hidden state computation in (6) with that from a GRU or an LSTM

4.6 Sequence to sequence learning

- RNNs are used to design language models, which are key to natural language processing
- another important benchmark is *machine translation*, a central problem domain for *sequence transduction* models that transform input sequences into output sequences
- playing a crucial role in various modern AI applications, sequence transduction models will form the focus of the remainder of this chapter and Chapter 5
- thus, we introduce the machine translation problem

4.6 Sequence to sequence learning

- *machine translation* refers to the automatic translation of a sequence from one language to another
- for decades, statistical approaches had been dominant in this field, before the rise of end-to-end learning using neural networks
- the latter is often called *neural machine translation* to distinguish itself from *statistical machine translation* that involves statistical analysis in components such as the translation model and the language model
- we will focus on neural machine translation methods
- different from language models, for which the corpus is in one single language, machine translation datasets are composed of pairs of text sequences that are in the *source language* and the *target language*, respectively

- thus, machine translation is a major problem domain for sequence transduction models, whose input and output are both variable-length sequences
- to handle this type of inputs and outputs, we can design an architecture with two major components
- the first component is an *encoder*: it takes a variable-length sequence as the input and transforms it into a state with a fixed shape
- the second component is a *decoder*: it maps the encoded state of a fixed shape to a variable-length sequence
- this is called an *encoder-decoder* architecture, which is depicted in Figure 13

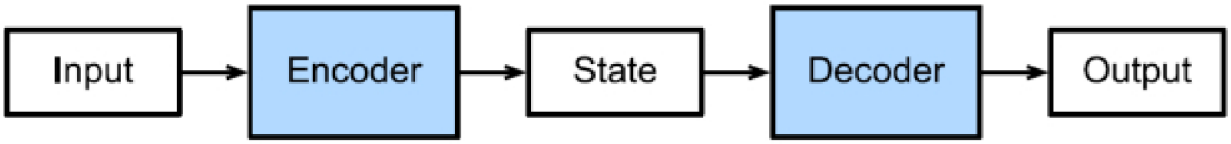

Figure 13: The encoder-decoder architecture.

4.6 Sequence to sequence learning

- let us take machine translation from English to French as an example
- given an input sequence in English: “They”, “are”, “watching”, “.”, this encoder-decoder architecture first encodes the variable-length input into a state, then decodes the state to generate the translated sequence token by token as the output: “Ils”, “regardent”, “.”
- the term “state” in the encoder-decoder architecture has probably inspired us to think about neural networks with states, when instantiating this architecture
- next, we will see how to apply RNNs to design sequence transduction models based on this encoder-decoder architecture

4.6 Sequence to sequence learning

- in machine translation, both the input and output are a variable-length sequence
- to address this type of problem, we have designed a general encoder-decoder architecture
- now, we will use two RNNs to design the encoder and the decoder of this architecture, and apply it to sequence to sequence learning for machine translation

4.6 Sequence to sequence learning

- following the design principle of the encoder-decoder architecture, the RNN encoder can take a variable-length sequence as the input, and transform it into a fixed-shape hidden state
- in other words, information of the input (source) sequence is *encoded* in the hidden state of the RNN encoder
- to generate the output sequence token by token, a separate RNN decoder can predict the next token based on what tokens have been seen (such as in language modeling) or generated, together with the encoded information of the input sequence
- Figure 14 illustrates how to use two RNNs for sequence to sequence learning in machine translation

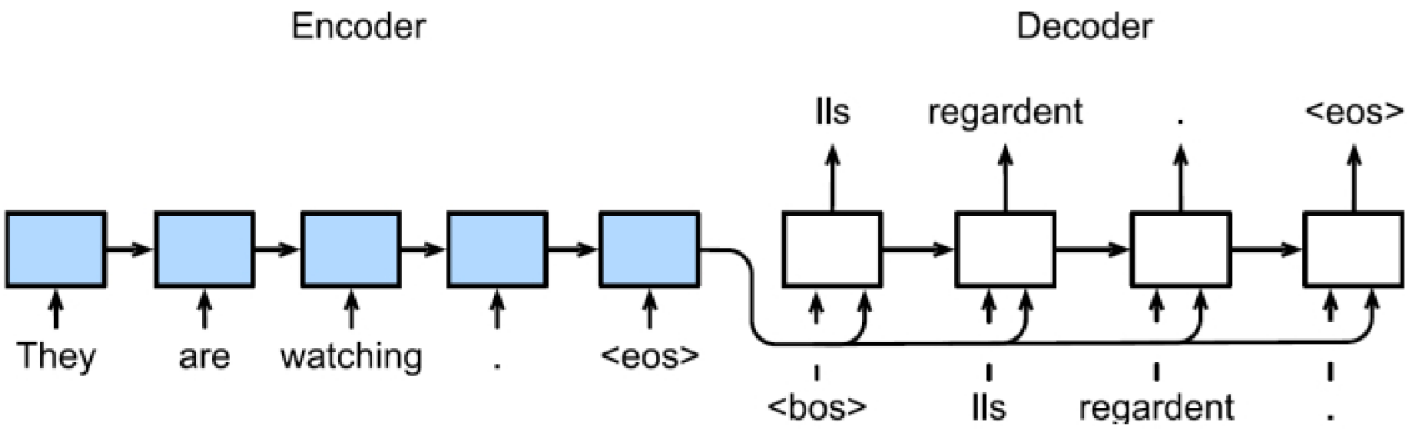

Figure 14: Sequence to sequence learning with an RNN encoder and an RNN decoder.

- in Figure 14, the special “<eos>” token marks the end of the sequence
- the model can stop making predictions once this token is generated
- at the initial time step of the RNN decoder, there are two special design decisions
- first, the special beginning-of-sequence “<bos>” token is an input
- second, the final hidden state of the RNN encoder is used to initialize the hidden state of the decoder

- in some designs, this is exactly how the encoded input sequence information is fed into the decoder for generating the output (target) sequence
- in some other designs, the final hidden state of the encoder is also fed into the decoder as part of the inputs at every time step, as shown in Figure 14
- similar to the training of language models, we can allow the labels to be the original output sequence, shifted by one token: “<bos>”, “Ils”, “regardent”, “.” $\rightarrow$ “Ils”, “regardent”, “.”, “<eos>”
- in the following, we will explain the design of Figure 14 in greater detail

- technically speaking, the encoder transforms an input sequence of variable length into a fixed-shape *context variable* $\boldsymbol{c}$, and encodes the input sequence information in this context variable
- as depicted in Figure 14, we can use an RNN to design the encoder
- let us consider a sequence example (batch size: 1)
- suppose that the input sequence is $x_1, \ldots, x_T$, such that x_t is the tth token in the input text sequence
- at time step t, the RNN transforms the input feature vector $\boldsymbol{x}_t$ for x_t and the hidden state $\boldsymbol{h}_{t-1}$ from the previous time step into the current hidden state $\boldsymbol{h}_t$
- we can use a function f to express the transformation of the RNN's recurrent layer:

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}).$$

- in general, the encoder transforms the hidden states at all the time steps into the context variable through a customized function q:

$$\boldsymbol{c} = q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T).$$

- for example, when choosing $q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T) = \boldsymbol{h}_T$, such as in Figure 14, the context variable is just the hidden state $\boldsymbol{h}_T$ of the input sequence at the final time step
- so far, we have used an RNN to design the encoder, where a hidden state only depends on the input subsequence at, and before, the time step of the hidden state

- as we just mentioned, the context variable $\boldsymbol{c}$ of the encoder's output encodes the entire input sequence $x_1, \ldots, x_T$
- given the output sequence $y_1, y_2, \ldots, y_{T'}$ from the training dataset, for each time step t' (the symbol differs from the time step t of input sequences or encoders), the probability of the decoder output $y_{t'}$ is conditional on the previous output subsequence $y_1, \ldots, y_{t'-1}$, and the context variable $\boldsymbol{c}$, i.e., $P(y_{t'}|y_1, \ldots, y_{t'-1}, \boldsymbol{c})$

- to model this conditional probability on sequences, we can use another RNN as the decoder
- at any time step t' on the output sequence, the RNN takes the output $y_{t'-1}$ from the previous time step and the context variable $\boldsymbol{c}$ as its input, then transforms them and the previous hidden state $\boldsymbol{s}_{t'-1}$ into the hidden state $\boldsymbol{s}_{t'}$ at the current time step
- as a result, we can use a function g to express the transformation of the decoder's hidden layer:

$$\boldsymbol{s}_{t'} = g(y_{t'-1}, \boldsymbol{c}, \boldsymbol{s}_{t'-1}).$$

- after obtaining the hidden state of the decoder, we can use an output layer and the softmax operation to compute the conditional probability distribution $P(y_{t'}|y_1, \ldots, y_{t'-1}, \boldsymbol{c})$ for the output at time step t'
- to predict the output sequence token by token, at each decoder time step, the predicted token from the previous time step is fed into the decoder as an input
- similar to training, at the initial time step, the beginning-of-sequence (“<bos>”) token is fed into the decoder
- this prediction process is illustrated in Figure 15
- when the end-of-sequence (“<eos>”) token is predicted, the prediction of the output sequence is complete

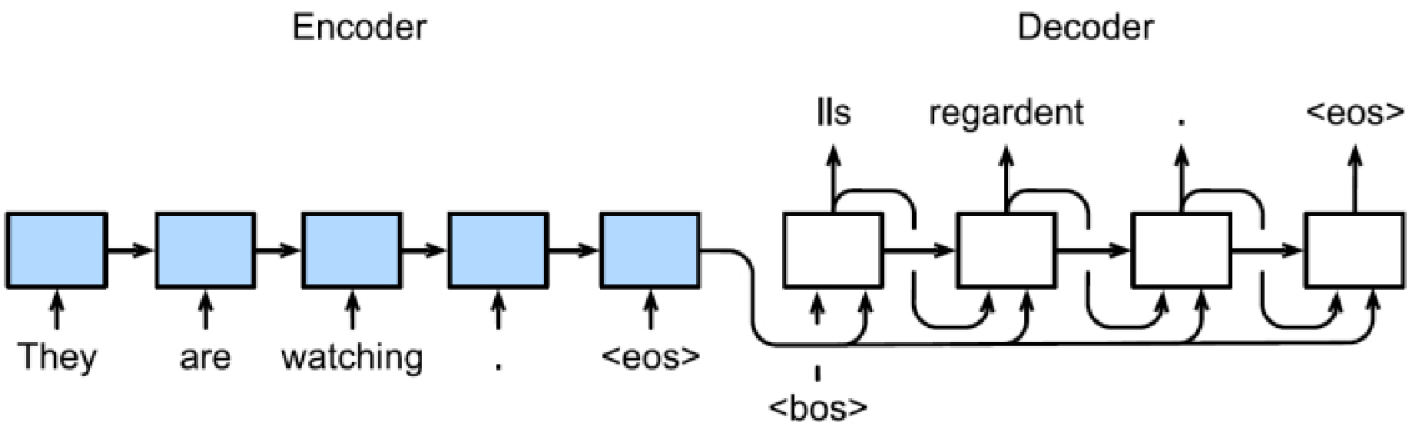

Figure 15: Predicting the output sequence token by token using an RNN encoder-decoder.

- we can evaluate a predicted sequence by comparing it with the label sequence (the ground truth)
- *BLEU* (*Bilingual Evaluation Understudy*), though originally proposed for evaluating machine translation results, has been extensively used in measuring the quality of output sequences for different applications
- in principle, for any *n*-grams in the predicted sequence, BLEU evaluates whether these *n*-grams appear in the label sequence

- denote by p_n the precision of n-grams, which is the ratio of the number of matched n-grams in the predicted and label sequences to the number of n-grams in the predicted sequence
- to explain, given a label sequence A, B, C, D, E, F, and a predicted sequence A, B, B, C, D, we have $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$, and $p_4 = 0$
- besides, let $\text{len}_{\text{label}}$ and len_{pred} be the number of tokens in the label sequence and the predicted sequence, respectively
- then, BLEU is defined as:

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}, \tag{7}$$

where k is the longest n-grams for matching

- based on the definition of BLEU in (7), whenever the predicted sequence is the same as the label sequence, BLEU is 1
- moreover, since matching longer n-grams is more difficult, BLEU assigns a greater weight to a longer n-gram precision
- specifically, when p_n is fixed, $p_n^{1/2^n}$ increases as n grows (the original paper uses $p_n^{1/n}$)
- furthermore, since predicting shorter sequences tends to obtain a higher p_n value, the coefficient before the multiplication term in (7) penalizes shorter predicted sequences
- for example, when $k = 2$, given the label sequence A, B, C, D, E, F and the predicted sequence A, B, although $p_1 = p_2 = 1$, the penalty factor $\exp(1 - 6/2) \approx 0.14$ lowers the BLEU

Thank you!