DSCI 565: MODERN RECURRENT NEURAL NETWORKS

Ke-Thia Yao

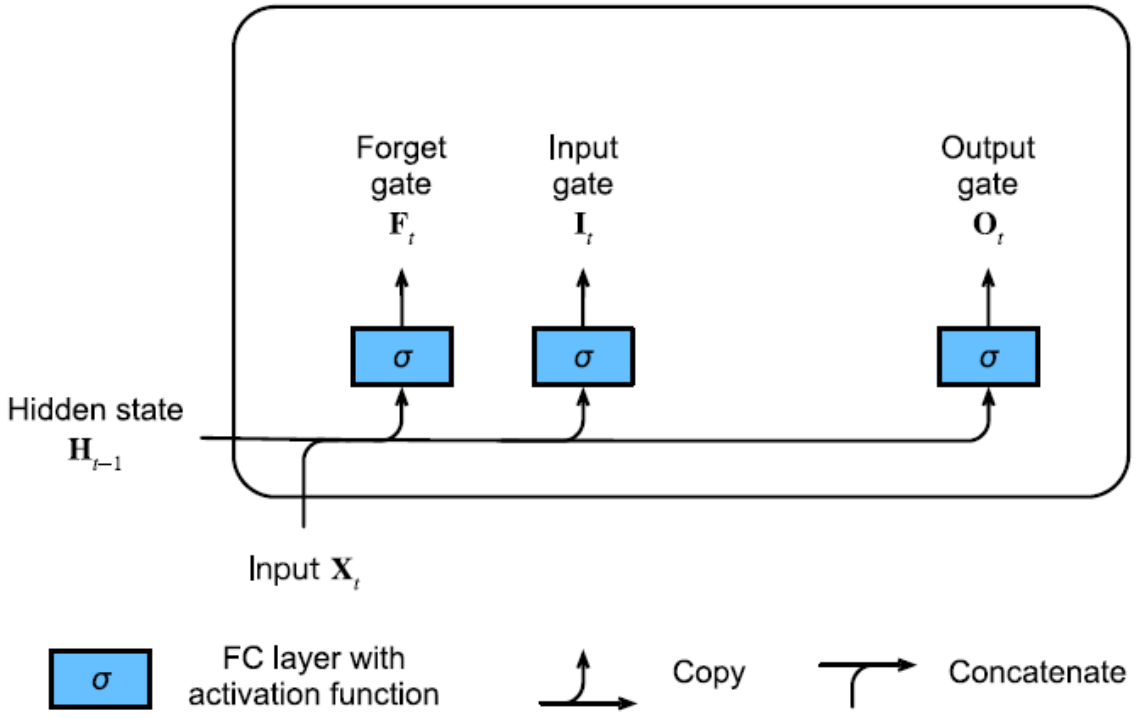
Lecture 13: 2025 October 8

Long Short-Term Memory (LSTM)

- Idea: in addition to the hidden state, each node should also hold an *internal state* for storing **short-term memory**
- This internal state is passed from through a self-connected recurrent edge with fixed weight 1 to avoid vanishing or exploding gradients
- The **long-term memory** are the weight parameters. And they only changed only during training

Memory Cell Gates

- Nodes in LSTM model are called memory cells
- Memory cells contain three types of gates:
  - Forget gate determines which parts of the internal state should be forgotten
  - Input gate determines which parts of the "input" should be added to the internal state
  - Output gate determines which parts of the input should contribute to the hidden state



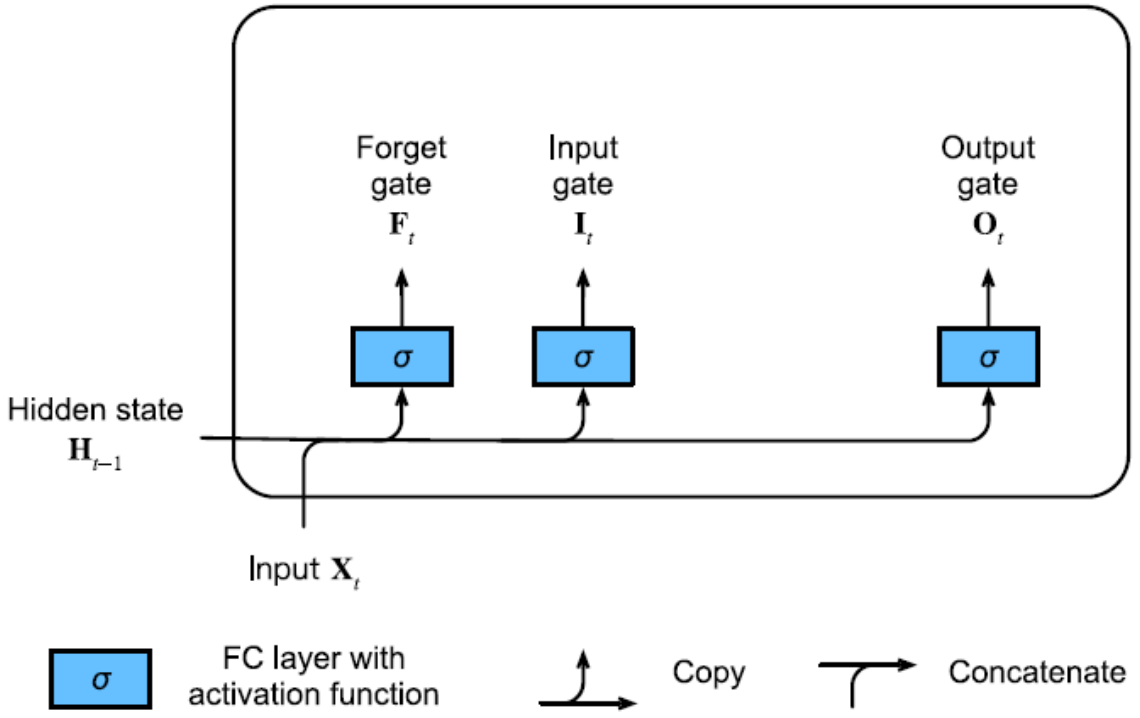
Memory Cell Gates

- $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$
- $\mathbf{F}_t, \mathbf{I}_t, \mathbf{O}_t \in \mathbb{R}^{n\times h}$
- σ is sigmoid function with range (0,1)

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),$$

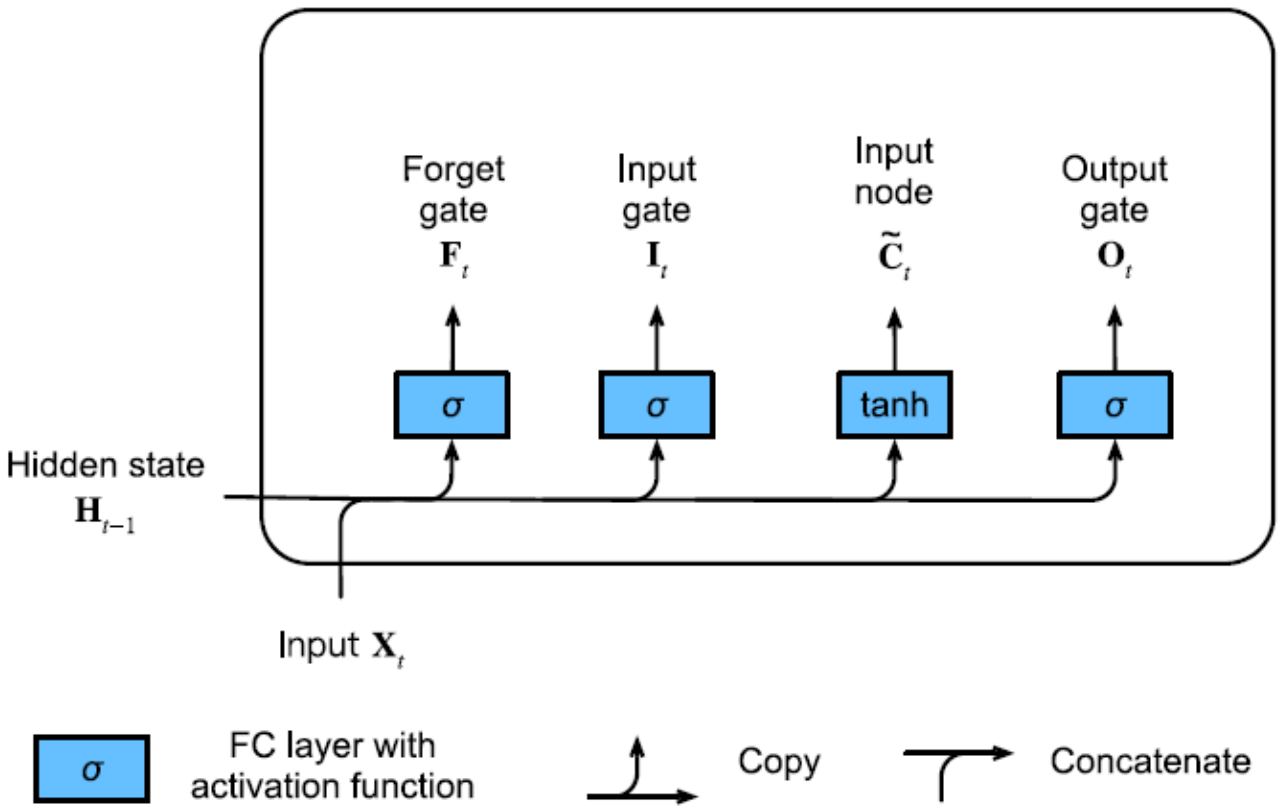


Input Node

- The input node proposes which parts of the input should be added to the internal state
- $\mathbf{X}_t \in \mathbb{R}^{n \times d}$, $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$
- $\tilde{\mathbf{C}} \in \mathbb{R}^{n \times h}$

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

- Where tanh has similar shape as sigmoid, except its range is (-1,1)



Updating Internal and Hidden State

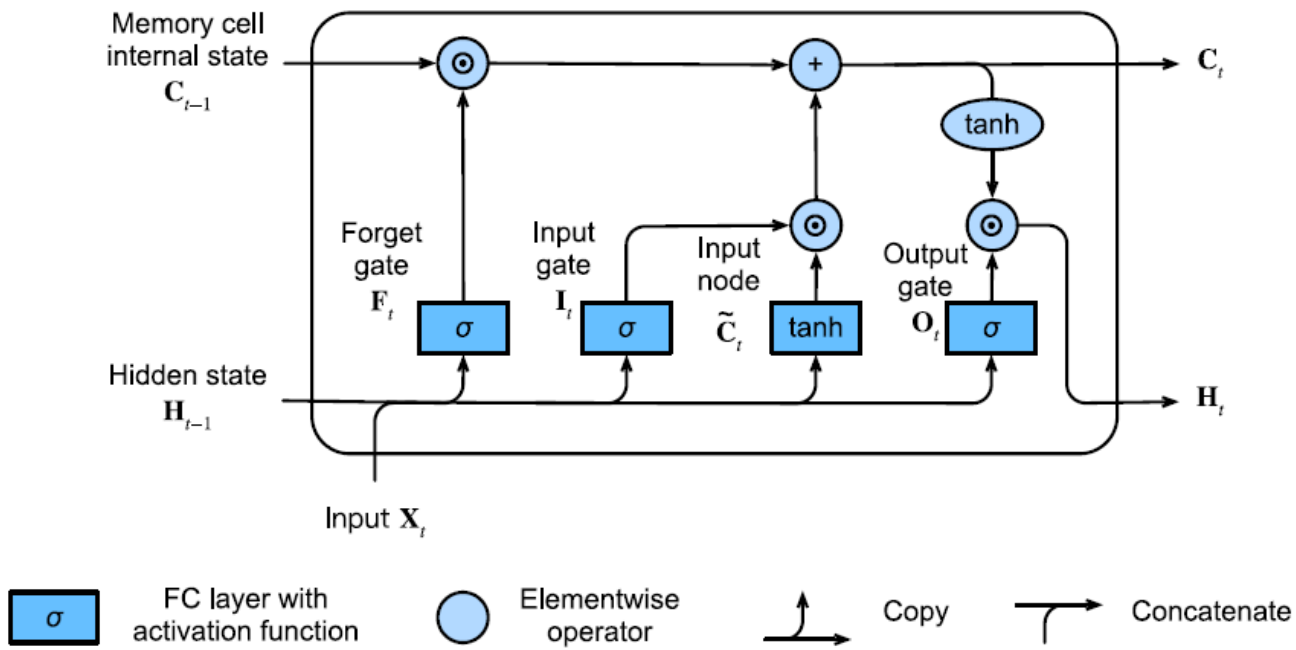
- Internal State $\mathbf{C}_t$

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

- Hidden State $\mathbf{H}_t$

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

- Where $\odot$ is elementwise multiplication



Notebook

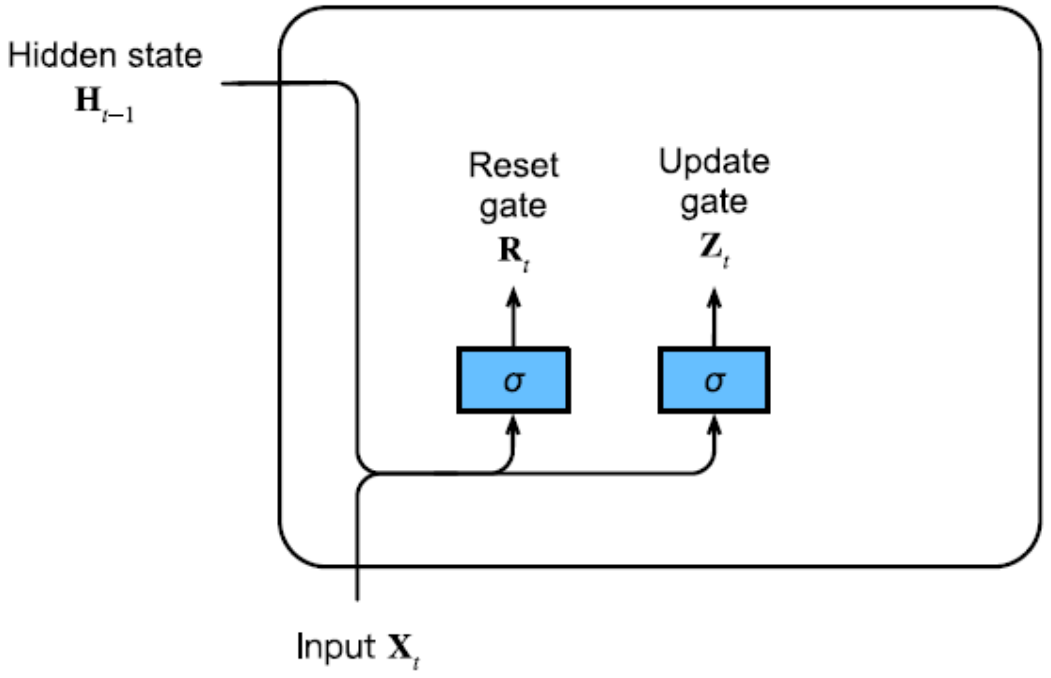
- chapter_recurrent-modern/lstm.ipynb

Gated Recurrent Unit (GRU)

- Simplified version of LSTM with the aim of speeding up computation and providing comparable performance
- Retains the multiplicative gating mechanism
- Merges LSTM internal state and hidden state into one vector

Reset Gate and Update Gate

- Update gate controls which parts of hidden state to forget and to add
- Reset gate controls which parts of hidden state to use for next candidate hidden state
- $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$
- $\mathbf{R}_t, \mathbf{Z}_t \in \mathbb{R}^{n\times h}$

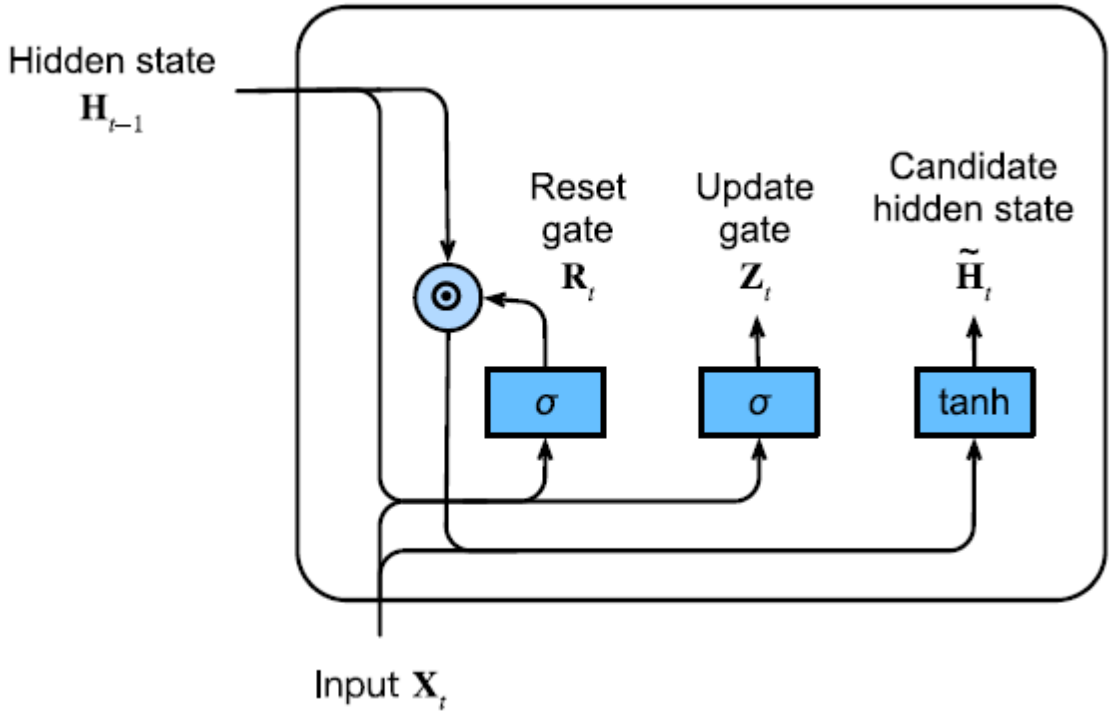


$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\mathrm{xr}} + \mathbf{H}_{t-1} \mathbf{W}_{\mathrm{hr}} + \mathbf{b}_{\mathrm{r}}),$$

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{\mathrm{xz}} + \mathbf{H}_{t-1} \mathbf{W}_{\mathrm{hz}} + \mathbf{b}_{\mathrm{z}}),$$

Candidate Hidden State

- Proposes candidate hidden state $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ by combining
  - MLP with Input $\mathbf{X}_t$
  - Parts of previous hidden state
- If $\mathbf{R}_t$ outputs all 1s, then same as regular RNN
- Any position $\mathbf{R}_t$ outputs 0s is replaced with MLP with Input $\mathbf{X}_t$

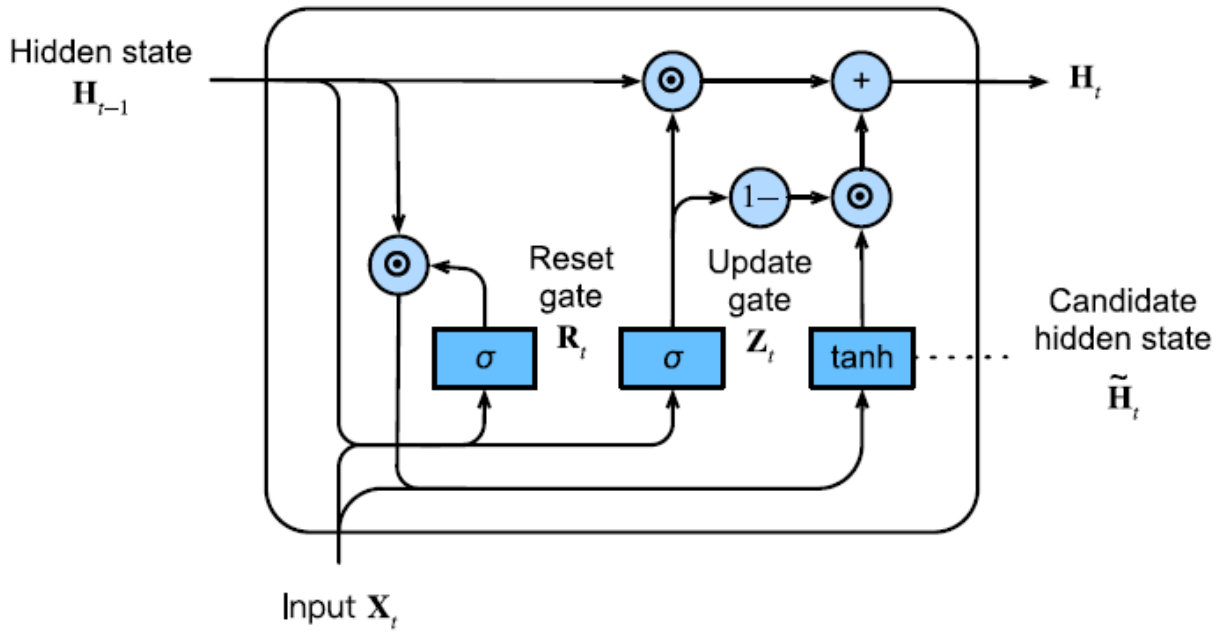


$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{\mathrm{xh}} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{\mathrm{hh}} + \mathbf{b}_{\mathrm{h}}).$$

Hidden State

- Compute the next hidden state by constructing it from
  - Parts of the previous hidden state
  - Parts of the candidate hidden state
- If $\mathbf{Z}_t$ is close to 1 use $\mathbf{H}_{t-1}$
- If $\mathbf{Z}_t$ is close to 0 use $\tilde{\mathbf{H}}_t$

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



e.g. $Z_t = (0.1, 0.1)$

$1 - Z_t = (1.0, 1.0)$

$H_{t-1} = (h_0, h_1, h_2, h_3)$

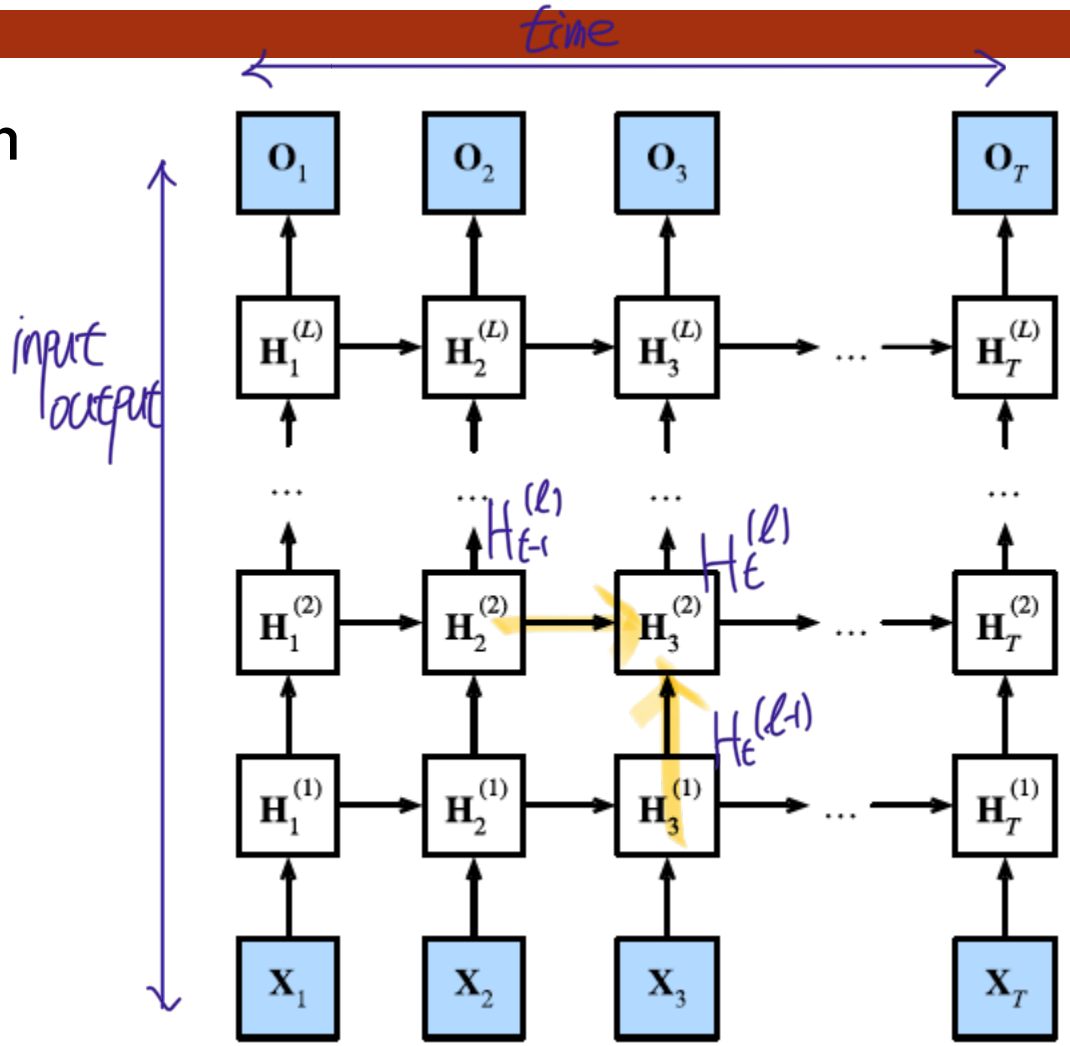
$H_t = (\tilde{h}_0, h_1, \tilde{h}_2, h_3)$

Deep Recurrent neural Networks

- Deep both in the time direction and in the input-output direction
- Let $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n\times h}, l = 1, \ldots, L,$ $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, and $\mathbf{O}_t \in \mathbb{R}^{n\times q}$

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{\text{xh}}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{\text{hh}}^{(l)} + \mathbf{b}_{\text{h}}^{(l)}),$$

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}},$$

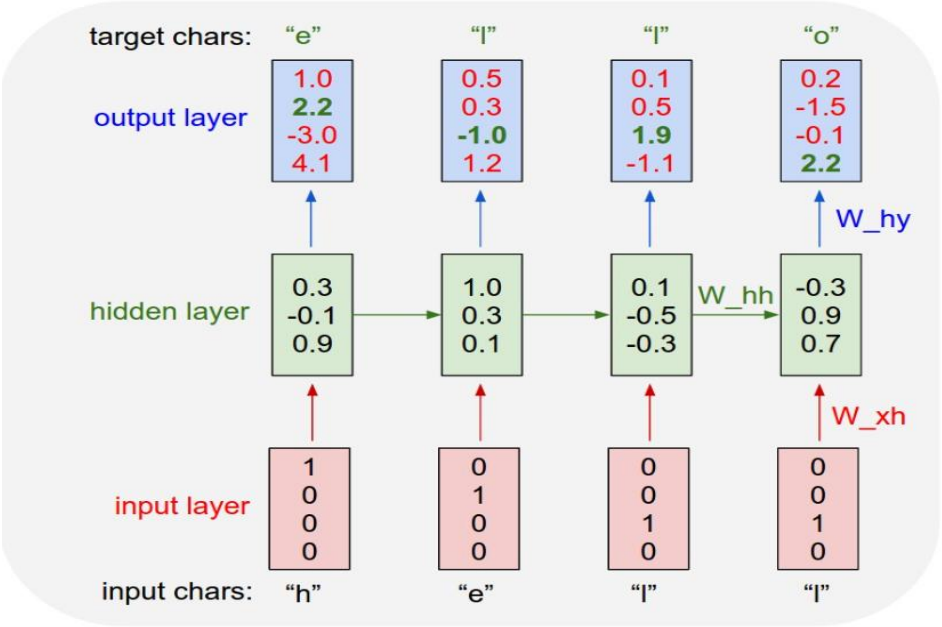


Notebook

- chapter_recurrent-modern/gru.ipynb
- chapter_recurrent-modern/deep-rnn.ipynb

RNN Application: Text Generation

- Character RNN – train a RNN to predicate the next character
- "Shakespeare" charRNN
  - 3-layer RNN with 512 hidden nodes
  - Trained on entire work of Shakespeare 4.4 MB




```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.
```

https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Bidirectional RNN

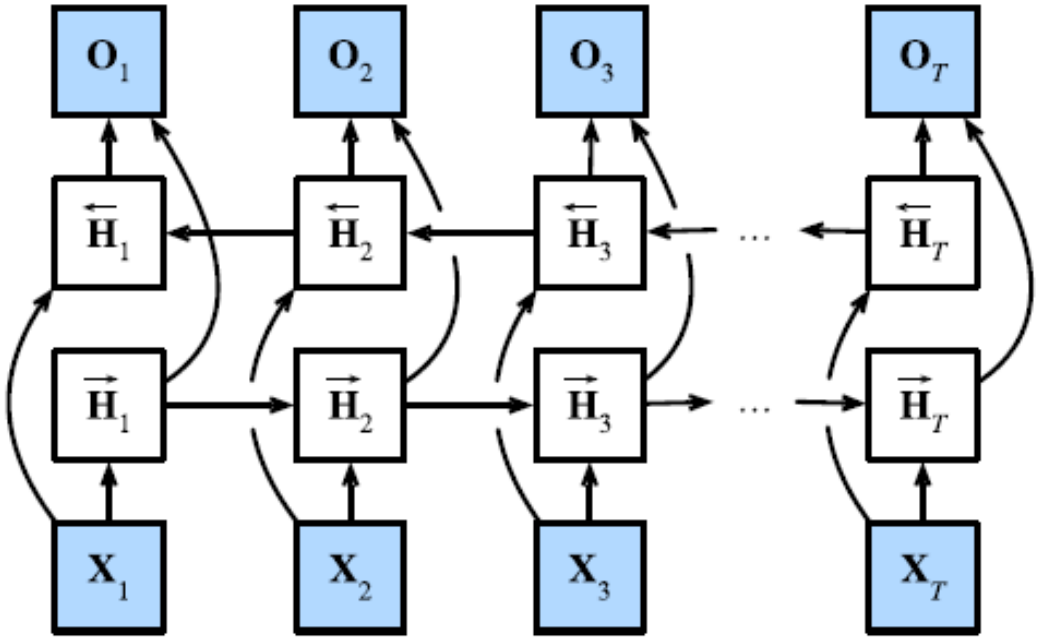
- In some sequence learning tasks, the entire sequence is available
- For examples, tasks like natural language translation, speech recognition, handwritten recognition and part-of-speech tagging
- In stead of just scanning the sequence from left to right, we can take advantage of context in the both directions

Predicting Masked Tokens

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a pig.

Bidirectional RNN

- Create two RNNs
- One RNN receives inputs from left-to-right $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$
  - Generating $\overrightarrow{\mathbf{H}}_1, \ldots, \overrightarrow{\mathbf{H}}_T$
- The other receives inputs from right-to-left $\mathbf{x}_T, \mathbf{x}_{T-1}, \ldots, \mathbf{x}_1$
  - Generating $\overrightarrow{\mathbf{H}}_T, \ldots, \overrightarrow{\mathbf{H}}_1$



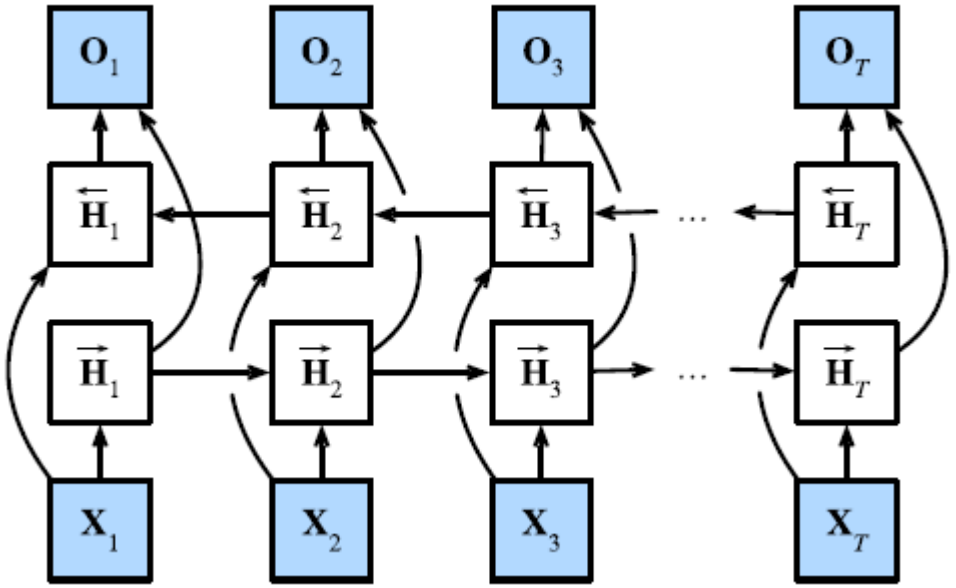
$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{\text{hh}}^{(f)} + \mathbf{b}_{\text{h}}^{(f)}),$$

$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{\text{hh}}^{(b)} + \mathbf{b}_{\text{h}}^{(b)}),$$

Bidirectional RNN Output

- $\mathbf{H}_t = \text{concat}(\overrightarrow{\mathbf{H}}_t, \overleftarrow{\mathbf{H}}_t) \in \mathbb{R}^{n \times 2h}$
- $\mathbf{O}_t \in \mathbb{R}^{n \times h}$

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

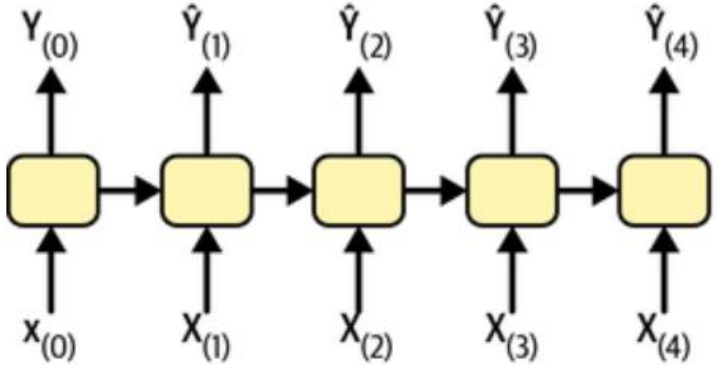


Notebooks

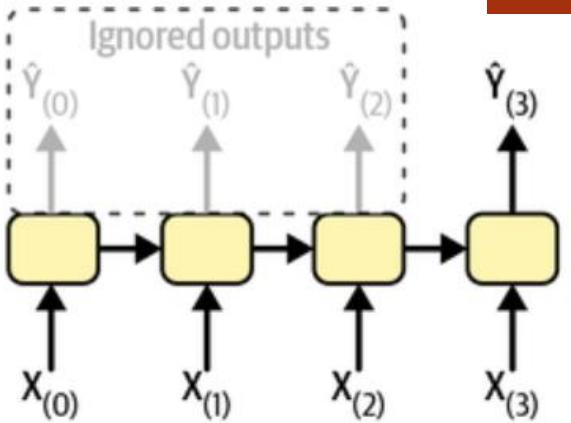
- chapter_recurrent-modern/bi-rnn.ipynb
- chapter_recurrent-modern/machine-translation-and-dataset.ipynb

Various Ways of Using RNNs

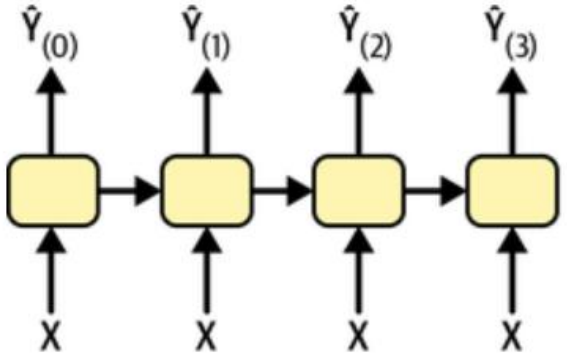
Sequence-to-Sequence



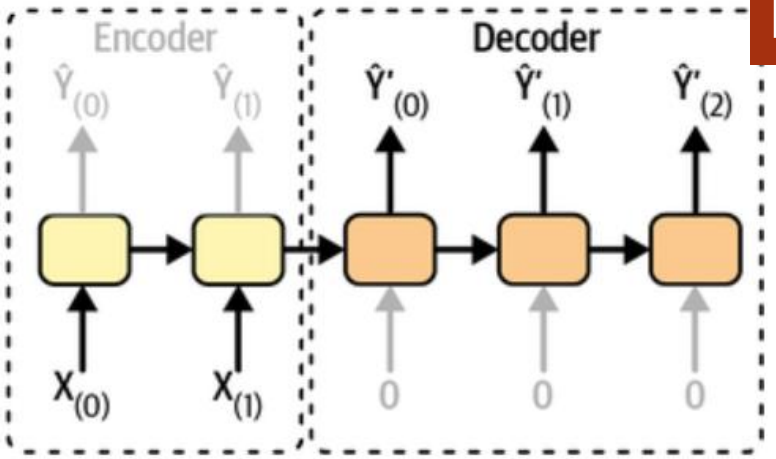
Sequence-to-Vector



Vector-to-Sequence



Encoder-Decoder

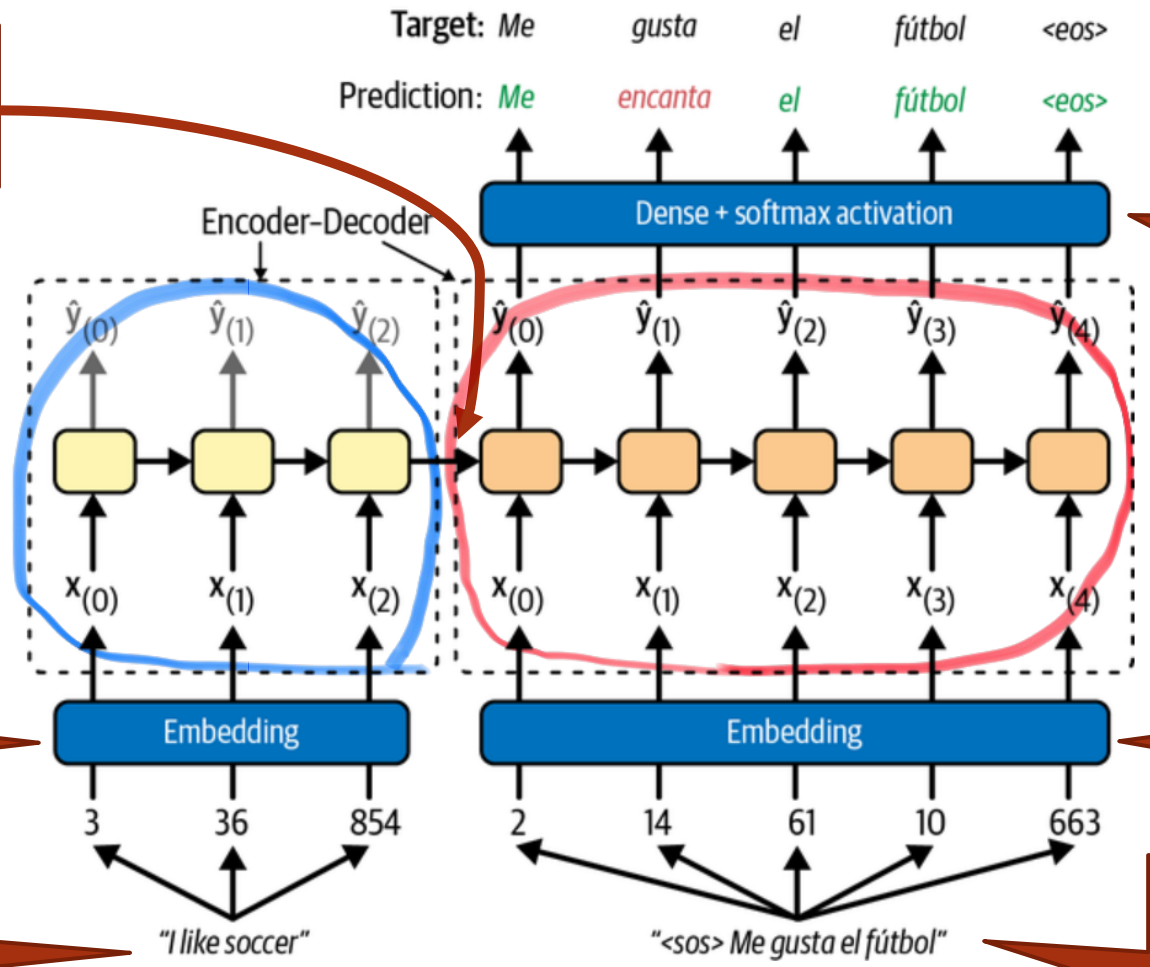


The Encoder-Decoder Architecture

- The encoder-decoder architecture has two main parts
- Encoder takes a variable length input and outputs a fixed length state
- Decoder takes a fixed length state and generates a variable length output
- For example, the input is a sentence in English,
  and the output is the translation of the sentence in French

Encoder-Decoder RNN for Language Translation

Transfer Encoder State Vector



Probability for each word

English Embedding

Spanish Embedding

Input: English sentence

Input: <sos> Spanish sentence

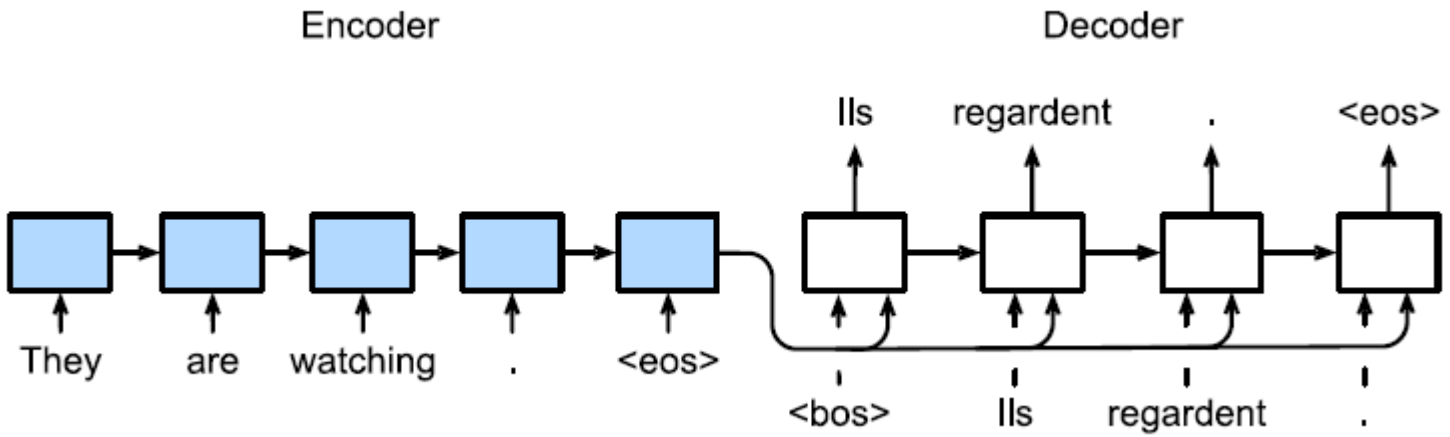
Figure 16-3. A simple machine translation model

Notebook

- chapter_recurrent-modern/encoder-decoder.ipynb

Sequence-to-sequence Learning for Machine Translation

- Add <eos> symbol to tell the encoder the sentence is complete
- Add <bos> symbol to the decoder to begin translation
- Teach forcing to train the decoder
  - Regardless the token generated by the decoder for the previous time step, for the next step always feed the decoder with the correct target token



Encoder

- From the input sequence the encoder generates a sequence of hidden states

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

- We can just pass $\mathbf{h}_T$ to the decoder
- More generally, we can construct a customized context variable

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

Encoder

```
class Seq2SeqEncoder(d2l.Encoder):  #@save
    """The RNN encoder for sequence-to-sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size, num_hiddens, num_layers, dropout)
        self.apply(init_seq2seq)

    def forward(self, X, *args):
        # X shape: (batch_size, num_steps)
        embs = self.embedding(X.t().type(torch.int64))
        # embs shape: (num_steps, batch_size, embed_size)
        outputs, state = self.rnn(embs)
        # outputs shape: (num_steps, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens)
        return outputs, state
```

Maps integers to random vectors

Process entire sequences in batches

Transpose X to have time as the first index

Outputs of top layer across all time

State of the layered RNN at last time T

Decoder

- Given a target output sequence $y_1, y_2, \ldots, y_{T'}$ for time step t' the decoder assigns a predicted probability to each possible token at next step

$$P(y_{t'+1} | y_1, \ldots, y_{t'}, \mathbf{c})$$

- We can represent the hidden state of the decoder as

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

- Note: the practice of including the context c at every time step differs
  - Some designs includes It at every time step, Cho et al., 2014
  - Others, only at the first-time step, Sutskever et al., 2014

```python
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = d2l.GRU(embed_size+num_hiddens, num_hiddens,
                           num_layers, dropout)
        self.dense = nn.LazyLinear(vocab_size)
        self.apply(init_seq2seq)

    def init_state(self, enc_all_outputs, *args):
        return enc_all_outputs

    def forward(self, X, state):
        # X shape: (batch_size, num_steps)
        # embs shape: (num_steps, batch_size, embed_size)
        embs = self.embedding(X.t().type(torch.int32))
        enc_output, hidden_state = state
        # context shape: (batch_size, num_hiddens)
        context = enc_output[-1]
        # Broadcast context to (num_steps, batch_size, num_hiddens)
        context = context.repeat(embs.shape[0], 1, 1)
        # Concat at the feature dimension
        embs_and_context = torch.cat((embs, context), -1)
        outputs, hidden_state = self.rnn(embs_and_context, hidden_state)
        outputs = self.dense(outputs).swapaxes(0, 1)
        # outputs shape: (batch_size, num_steps, vocab_size)
        # hidden_state shape: (num_layers, batch_size, num_hiddens)
        return outputs, [enc_output, hidden_state]
```

Input is word $\mathbf{x}_t$ and context $\mathbf{c}$

Dense layer to assign prob to all possible tokens

Transpose X to have time as the first index

Use output at last time step T as context

Combine words and context

Swap axes to get back having batch size first axis

Check Shape


```
vocab_size, embed_size, num_hiddens, num_layers = 10, 8, 16, 2
batch_size, num_steps = 4, 9
encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)
X = torch.zeros((batch_size, num_steps))
enc_outputs, enc_state = encoder(X)
d2l.check_shape(enc_outputs, (num_steps, batch_size, num_hiddens))
```

```
d2l.check_shape(enc_state, (num_layers, batch_size, num_hiddens))
```

```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)
state = decoder.init_state(encoder(X))
dec_outputs, state = decoder(X, state)
d2l.check_shape(dec_outputs, (batch_size, num_steps, vocab_size))
d2l.check_shape(state[1], (num_layers, batch_size, num_hiddens))
```

Abstract Encoder-Decoder Class

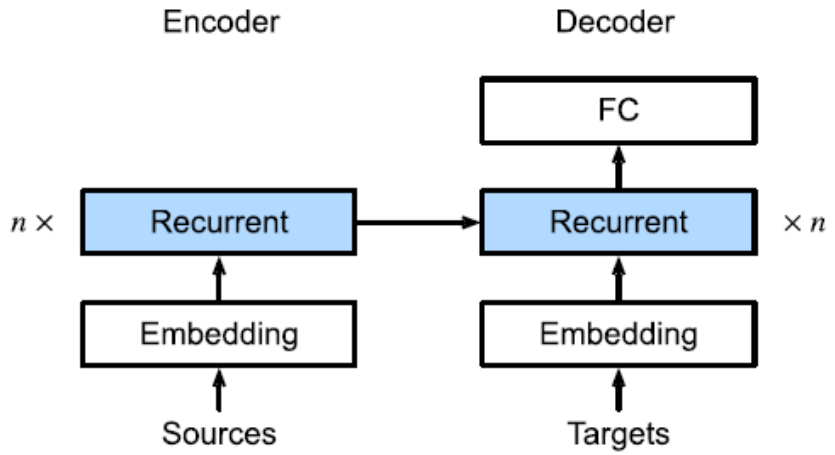

```
class EncoderDecoder(d2l.Classifier):  #@save
    """The base class for the encoder--decoder architecture."""
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_all_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_all_outputs, *args)
        # Return decoder output only
        return self.decoder(dec_X, dec_state)[0]
```

Generate encoding

Returns just the output sequence

Encoder-Decoder for Sequence-to-Sequence Learning




```python
class Seq2Seq(d2l.EncoderDecoder):  #@save
    """The RNN encoder--decoder for sequence to sequence learning."""
    def __init__(self, encoder, decoder, tgt_pad, lr):
        super().__init__(encoder, decoder)
        self.save_hyperparameters()

    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)

    def configure_optimizers(self):
        # Adam optimizer is used here
        return torch.optim.Adam(self.parameters(), lr=self.lr)
```

Loss defined on next slide

Adam optimizer, SDG with momentum

Loss Function with Masking

Cross Entropy loss inherited from d2l.Classifier

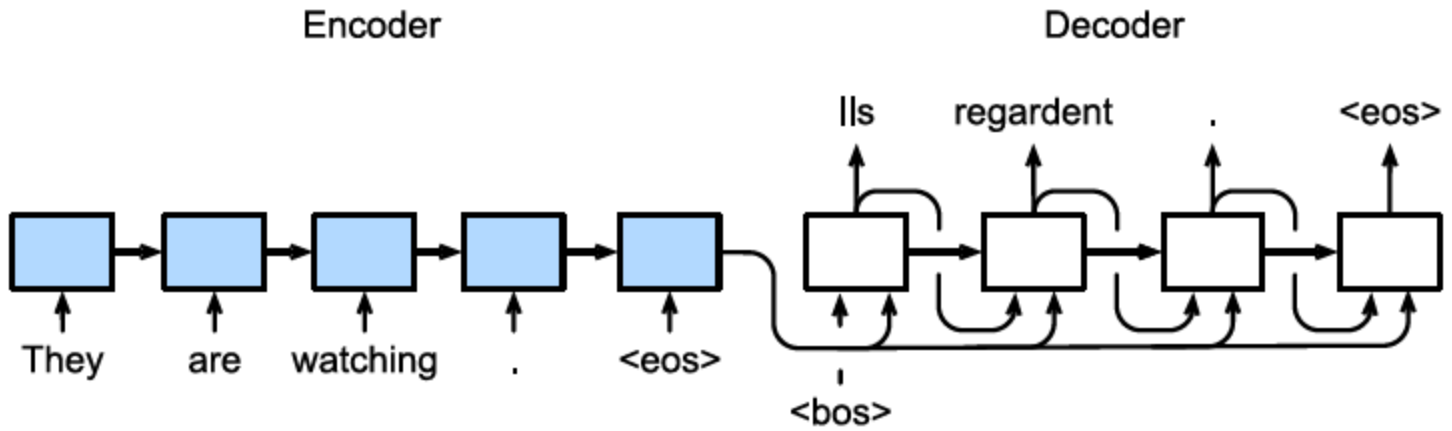
Skip <pad> tokens

```
@d2l.add_to_class(Seq2Seq)
def loss(self, Y_hat, Y):
    l = super(Seq2Seq, self).loss(Y_hat, Y, averaged=False)
    mask = (Y.reshape(-1) != self.tgt_pad).type(torch.float32)
    return (l * mask).sum() / mask.sum()
```

```
src, tgt, _,  _ = data.build(['hi .'], ['salut .'])
print('source:', data.src_vocab.to_tokens(src[0].type(torch.int32)))
print('target:', data.tgt_vocab.to_tokens(tgt[0].type(torch.int32)))
```

```
source: ['hi', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '<pad>', '
↪<pad>']
target: ['<bos>', 'salut', '.', '<eos>', '<pad>', '<pad>', '<pad>', '<pad>', '
↪<pad>']
```

Prediction




```python
@d2l.add_to_class(d2l.EncoderDecoder)  #@save
def predict_step(self, batch, device, num_steps,
                 save_attention_weights=False):
    batch = [a.to(device) for a in batch]
    src, tgt, src_valid_len, _ = batch
    enc_all_outputs = self.encoder(src, src_valid_len)
    dec_state = self.decoder.init_state(enc_all_outputs, src_valid_len)
    outputs, attention_weights = [tgt[:, (0)].unsqueeze(1), ], []
    for _ in range(num_steps):
        Y, dec_state = self.decoder(outputs[-1], dec_state)
        outputs.append(Y.argmax(2))
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weights.append(self.decoder.attention_weights)
    return torch.cat(outputs[1:], 1), attention_weights
```

Generate encoding for source sequences

Initialize output 2D tensor with <bos>

Greedy strategy: Output tokens associated with highest activation