EE 628
Deep Learning
Spring 2020

Lecture 10

04/02/2020

Sergul Aydore

Applied Scientist

Amazon Web Services

Announcements

- Deadline for Project Proposals: **04/03/2020 Friday at 5pm ET**
  - This includes creation of a github repository with READ.md file that contains the summary of the project.
  - Late submissions or repositories with empty READ.md file will lose 30 points from their grade for the project.
  - Email me the link to your github repository before deadline
- Deadline for Projects **04/27/2020 Monday at 5pm ET**
  - project presentation on **04/30/2020** (40 %)
  - Details on grading: https://github.com/sergulaydore/EE-628-Spring-2020

Overview

- Last lecture we covered
  - Data Preparation for RNNs
  - Implementing RNNs from scratch

- Today, we will cover
  - GRUs and LSTMs
  - Attention Mechanism
  - Transformers

- Source material:
  - Dive into Deep Learning (https://d2l.ai/)
  - https://github.com/sergulaydore/EE-628-Spring-2020

Backpropagation Through Time

- Now we will delve a bit more deeply into the details of backpropagation for sequence models and why (and how) the math works.
- Forward propagation in a recurrent neural network is relatively straightforward.
- Back-propagation through time is actually a specific application of back propagation in recurrent neural networks.

The computation graph

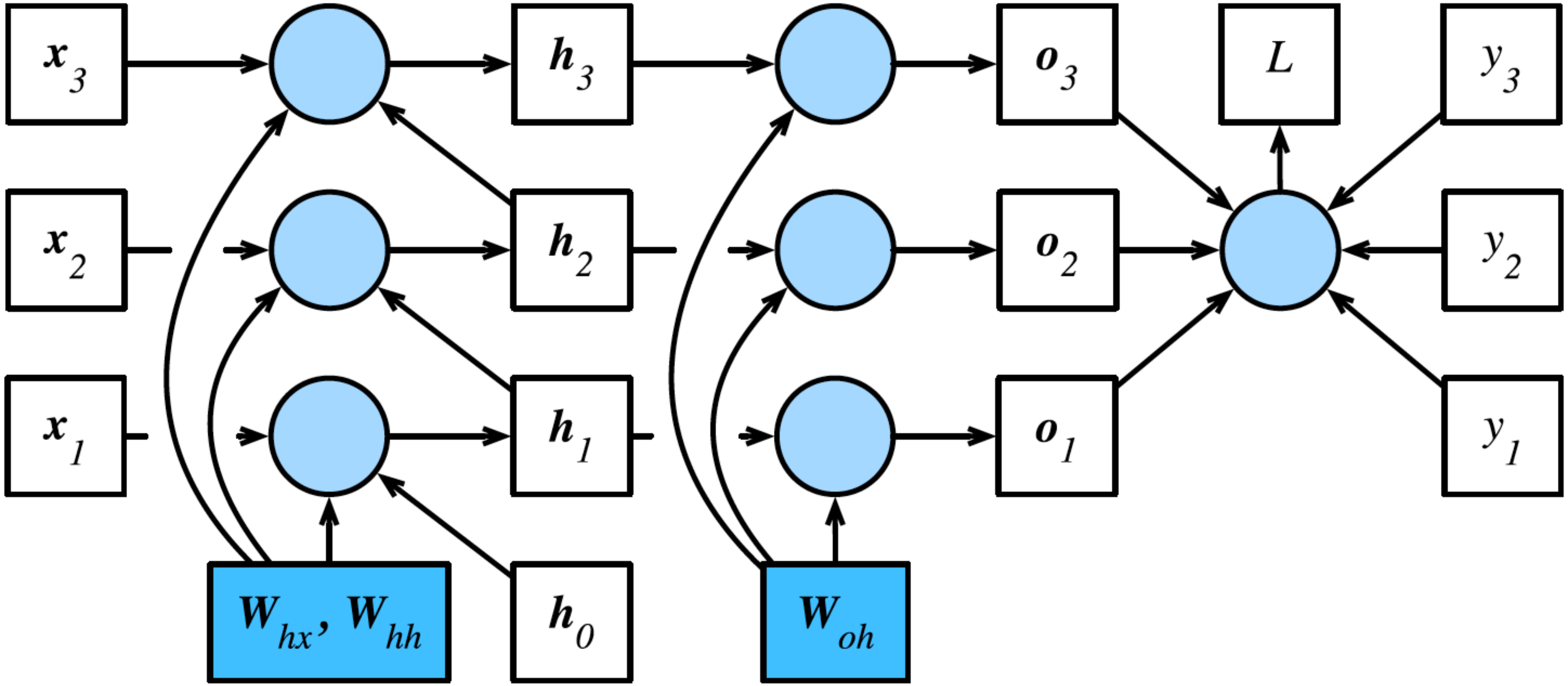


Fig. 10.7.2: Computational dependencies for a recurrent neural network model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

BPTT in detail

- In a simple linear latent variable model, we have:

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} \text{ and } \mathbf{o}_t = \mathbf{W}_{oh}\mathbf{h}_t$$

- Given an objective function $L(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{t=1}^{T} l(\mathbf{o}_t, y_t)$
- Derivatives wrt $\mathbf{W}_{oh}$ is straightforward:

$$\partial_{\mathbf{W}_{oh}} L = \sum_{t=1}^{T} \operatorname{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{h}_t\right)$$

- The dependency on $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$ is a bit tricky

$$\partial_{\mathbf{W}_{hh}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^{\top}\right)^{t-j} \mathbf{h}_j$$

$$\partial_{\mathbf{W}_{hh}} L = \sum_{t=1}^{T} \operatorname{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hh}} \mathbf{h}_t\right)$$

$$\partial_{\mathbf{W}_{hx}} L = \sum_{t=1}^{T} \operatorname{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hx}} \mathbf{h}_t\right)$$

$$\partial_{\mathbf{W}_{hx}} \mathbf{h}_t = \sum_{j=1}^{t} \left(\mathbf{W}_{hh}^{\top}\right)^{t-j} \mathbf{x}_j.$$

Problems with these gradients

- It pays to store intermediate results
  - i.e. powers of $\mathbf{W}_{hh}$
- Even this simple linear example involves potentially very large powers of $\mathbf{W}_{hh}^{j}$.
  - In it, eigenvalues smaller than 1 vanish for large j and eigenvalues larger than 1 diverge.
  - This is numerically unstable and gives undue importance to potentially irrelevant past detail.
- One way to address this is to truncate the sum at a computationally convenient size.
  - That is why we detached the gradients in the code

Gated Recurrent Units (GRU)

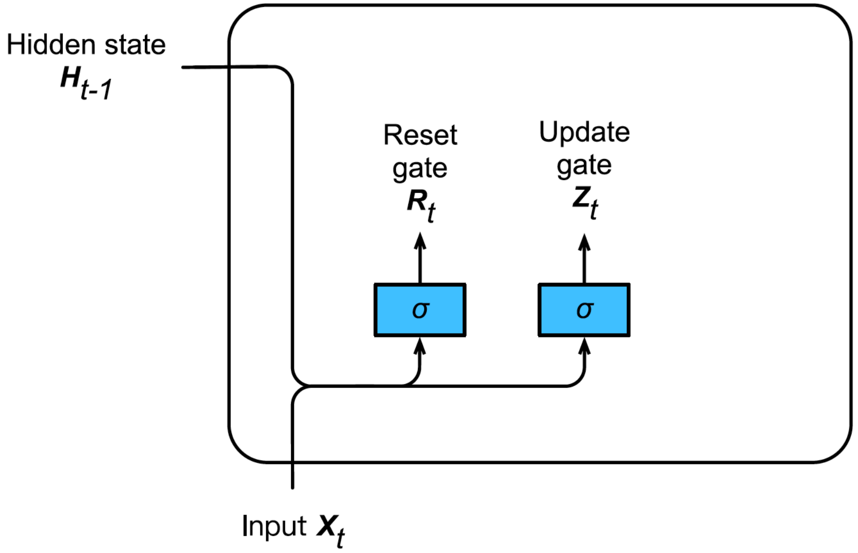
- We found that long products of matrices can lead to vanishing or divergent gradients.
- What such gradient anomalies mean in practice:
  - We might encounter a situation where an early observation is highly significant for predicting all future observations -> memory cell
  - We might encounter situations where some symbols carry no pertinent observation -> skipping such symbols
  - We might encounter situations where there is a logical break between parts of a sequence -> resetting our internal state
- A number of methods have been proposed to address this.
  - One of the earliest is the Long Short Term Memory (LSTM).
- The Gated Recurrent Unit (GRU) is a slightly more streamlined variant that often offers comparable performance and is significantly faster to compute.

Gating the Hidden State

- In GRUs, we have dedicated mechanisms for when the hidden state should be updated and also when it should be reset.
- These mechanisms are learned.
  - For instance, if the first symbol 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.

Reset Gates and Update Gates

- We engineer them to be vectors with entries in (0, 1) such that we can perform convex combinations.
- For instance, a reset variable would allow us to control how much of the previous state we might still want to remember.
- Likewise, an update variable would allow us to control how much of the new state is just a copy of the old state.



$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r)$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)$$

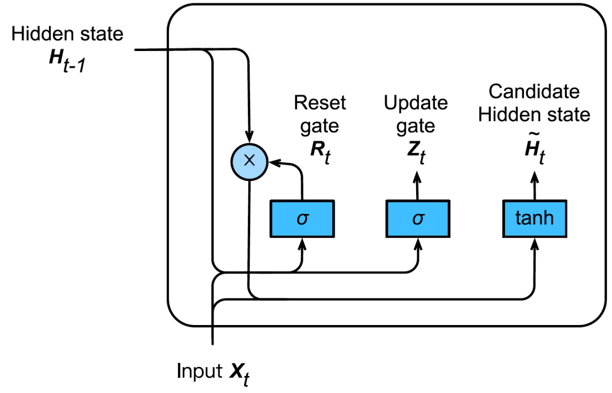
Reset Gate in action

- In a conventional RNN we would have an update of the form

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

- If we want to be able to reduce the influence of the previous states we can multiply $\mathbf{H}_{t-1}$ with $\mathbf{R}_t$ elementwise.
  - Whenever the entries in $\mathbf{R}_t$ are close to 1 we recover a conventional deep RNN.
  - For all entries of $\mathbf{R}_t$ that are close to 0 the hidden state is the result of an MLP with $\mathbf{X}_t$ as input.
- This leads to the following candidate for a new hidden state

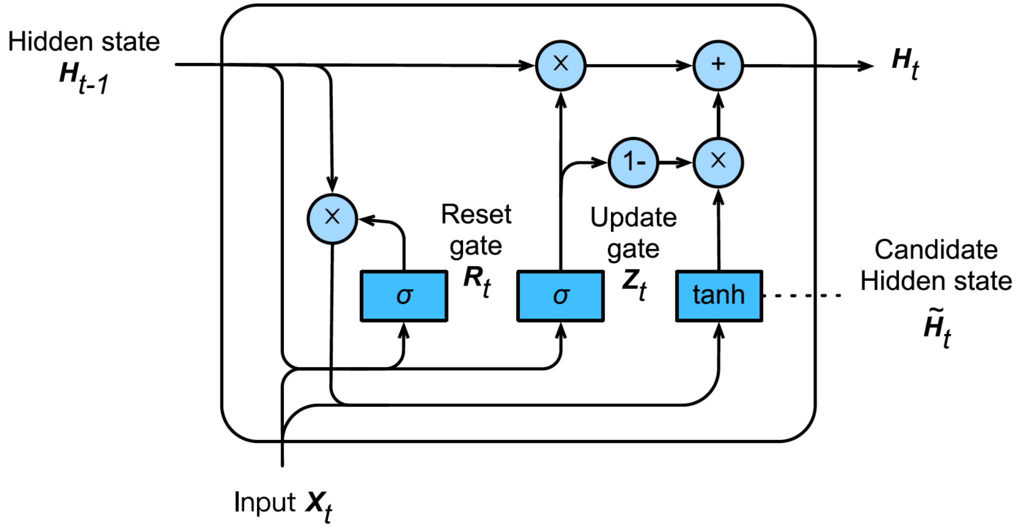
$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h)$$



Update Gate in action

- This determines the extent to which the new state $\mathbf{H}_t$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used.
- The gating variable $\mathbf{Z}_t$ can be used for this purpose leads to the final update equation for the GRU

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



In summary

- GRUs have the following two distinguishing features:
  - Reset gates help capture short-term dependencies in time series.
  - Update gates help capture long-term dependencies in time series

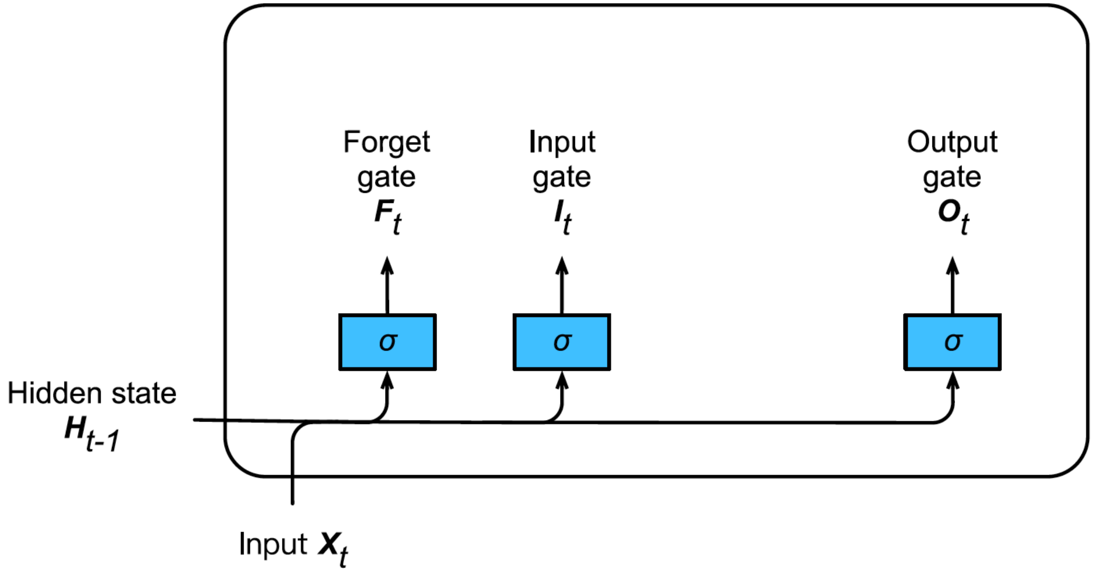
Long Short Term Memory (LSTM)

- It shares many of the properties of the Gated Recurrent Unit (GRU).
- Its design is slightly more complex.
- Arguably it is inspired by logic gates of a computer. To control a memory cell we need a number of gates.
  - One gate is needed to read out the entries from the cell -> output gate
  - A second gate is needed to decide when to read data into the cell -> input gate
  - Lastly, we need a mechanism to reset the contents of the cell -> forget gate

Gated Memory Cells - input gates, forget gates and output gates

- Three gates are introduced in LSTMs: the input gate, the forget gate, and the output gate.
- The gates are defined as follows: the input gate $\mathbf{I}_t \in R^{n \times h}$, the forget gate is $\mathbf{F}_t \in R^{n \times h}$, and the output gate is $\mathbf{O}_t \in R^{n \times h}$.

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



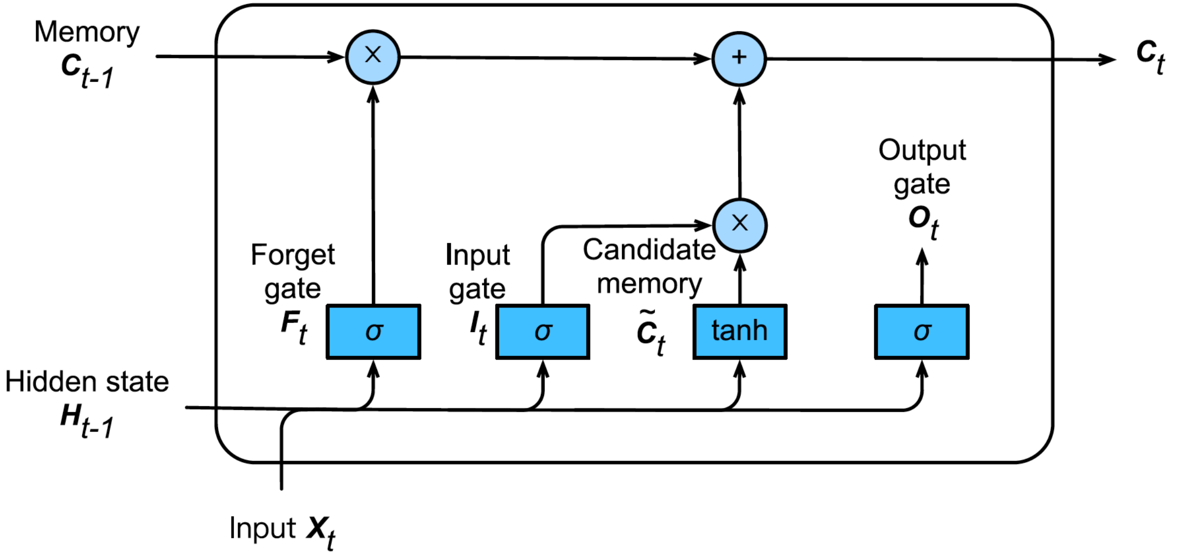
Gated Memory Cells - Memory Cells

- Candidate memory cell is defined as

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

- In GRUs we had a single mechanism to govern input and forgetting.
- Here we have two parameters, $\mathbf{I}_t$ which governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget parameter $\mathbf{F}_t$ which addresses how much we of the old memory cell content $\mathbf{C}_{t-1}$ we retain.

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

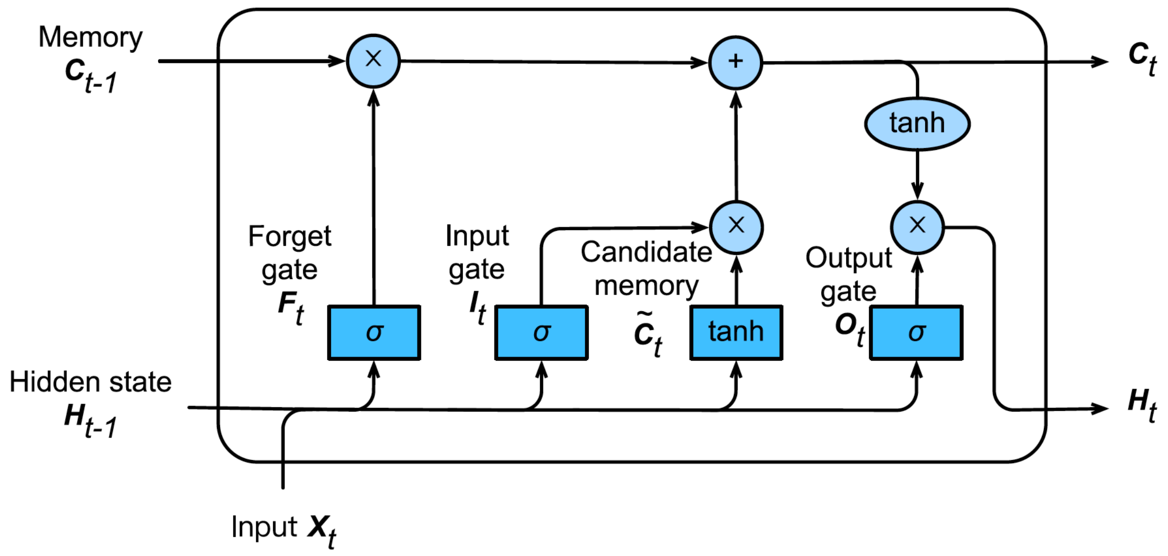


Gated Memory Cells – hidden states

- Lastly we need to define how to compute the hidden state $\mathbf{H}_t$
- This is where the output gate comes into play.

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

  - Whenever the output gate is 1 we effectively pass all memory information through to the predictor
  - whereas for output 0 we retain all information only within the memory cell and perform no further processing.



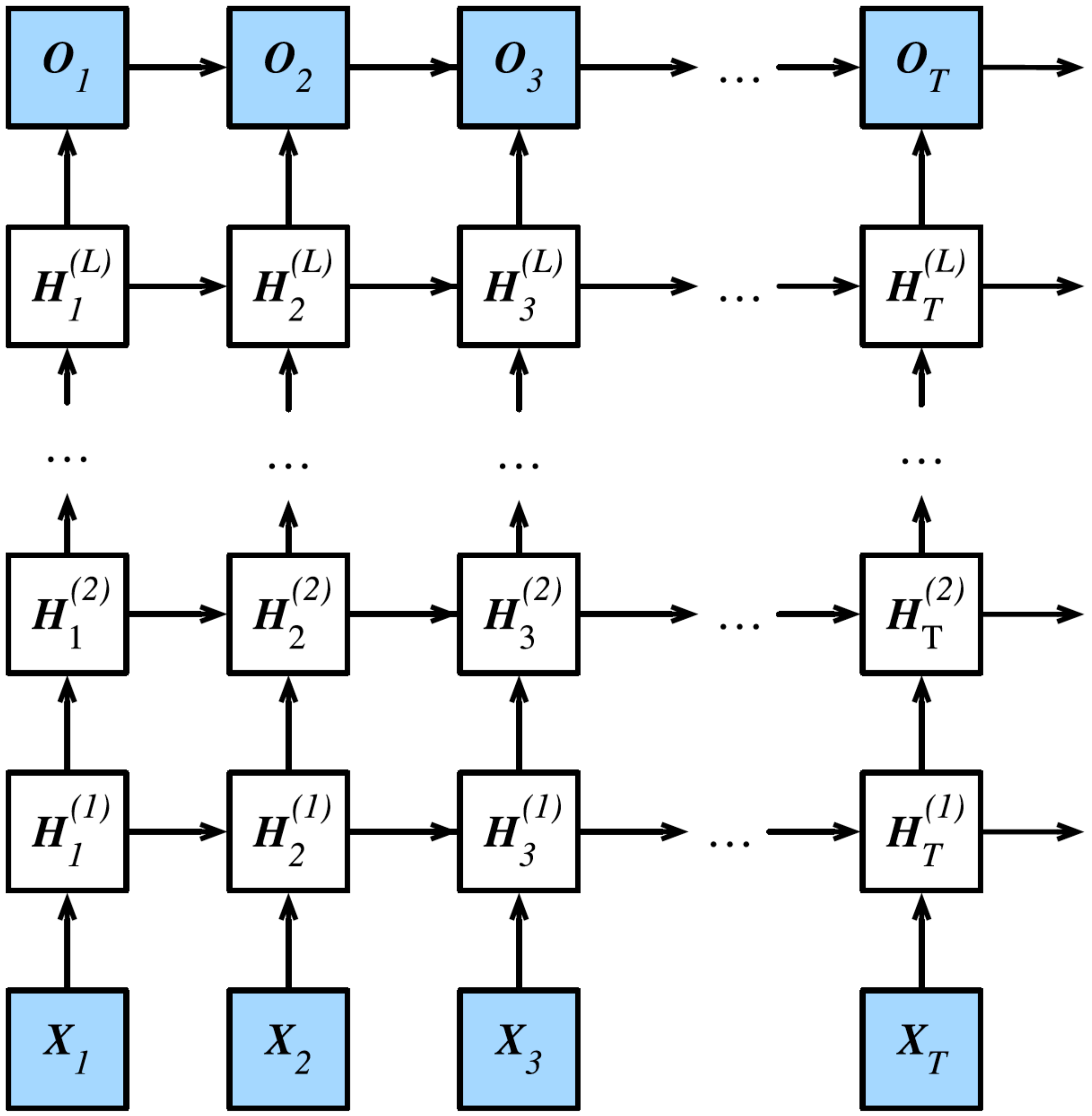
Deep Recurrent Neural Networks

- Consider a deep recurrent neural network with L hidden layers.
- Each hidden state is continuously passed to the next time step of the current layer and the next layer of the current time step.

$$\mathbf{H}_t^{(1)} = f_1\left(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}\right)$$

$$\mathbf{H}_t^{(l)} = f_l\left(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}\right)$$

$$\mathbf{O}_t = g\left(\mathbf{H}_t^{(L)}\right)$$



Bidirectional Recurrent Neural Networks

- So far we assumed that our goal is to model the next word given what we've seen so far.
- While this is a typical scenario, it is not the only one we might encounter.
- Consider the following three tasks of filling in the blanks in a text:

1. I am _____
2. I am _____ very hungry.
3. I am _____ very hungry, I could eat half a pig.

- Clearly the end of the phrase (if available) conveys significant information about which word to pick.

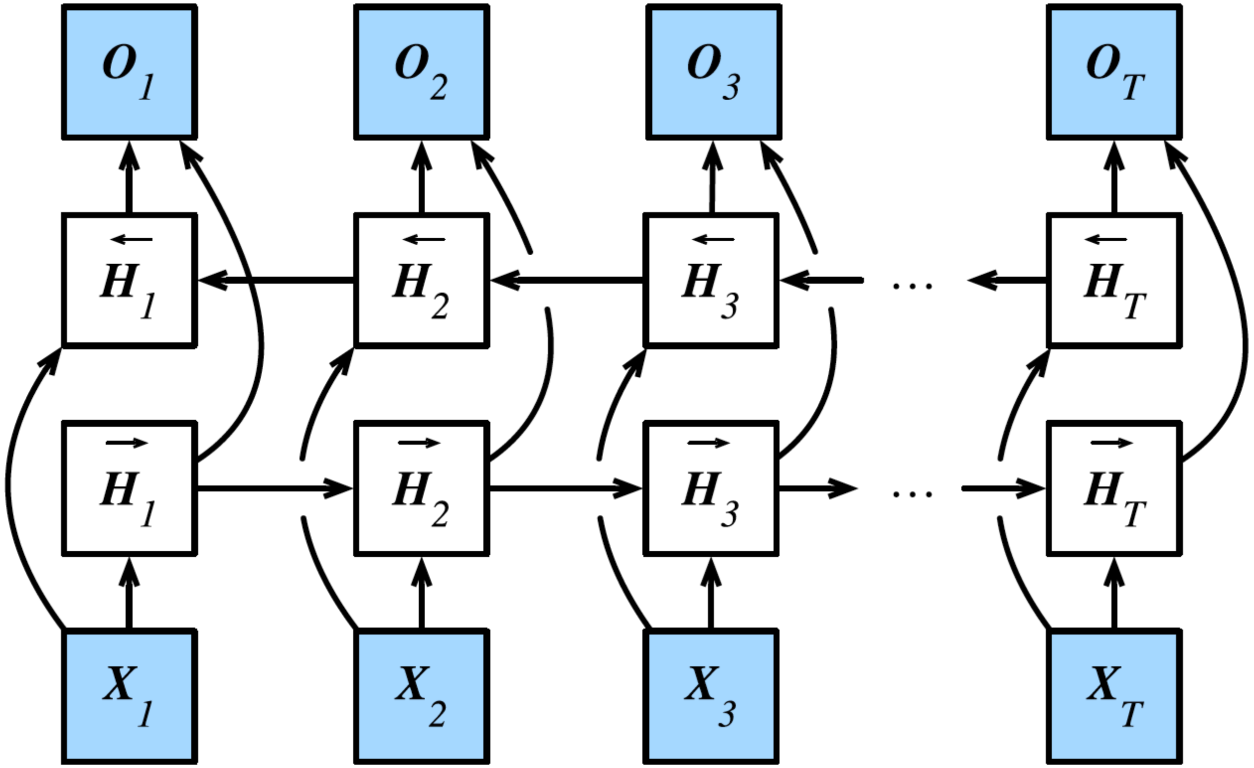
Bidirectional Model

- Instead of running an RNN only in forward mode starting from the first symbol we start another one from the last symbol running back to front.
- Bidirectional recurrent neural networks add a hidden layer that passes

information in a backward direction to more flexibly process such information.

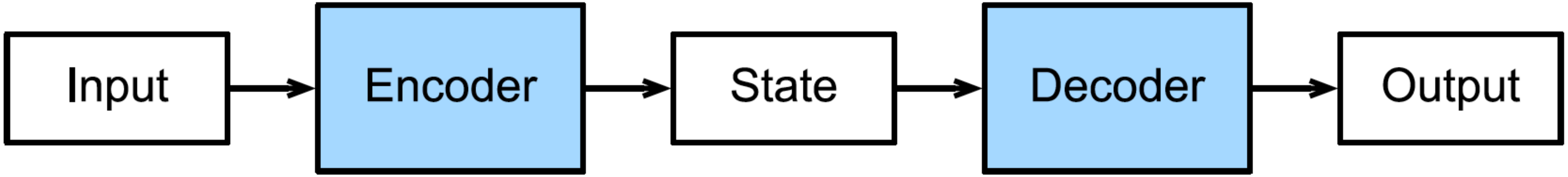
$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),$$

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



Encoder-Decoder architecture

- The encoder-decoder architecture is a neural network design pattern.
- The encoder’s role is encoding the inputs into state, which often contains several tensors.
- Then the state is passed into the decoder to generate the outputs.
- In machine translation, the encoder transforms a source sentence, e.g. “Hello world.”, into state, e.g. a vector, that captures its semantic information.
- The decoder then uses this state to generate the translated target sentence, e.g. “Bonjour le monde.”.



Sequence to Sequence

- The sequence to sequence (seq2seq) model is based on the encoder-decoder architecture to generate a sequence output for a sequence input.
- Both the encoder and the decoder use recurrent neural networks to handle sequence inputs.
- The hidden state of the encoder is used directly to initialize the decoder hidden state to pass information from the encoder to the decoder.

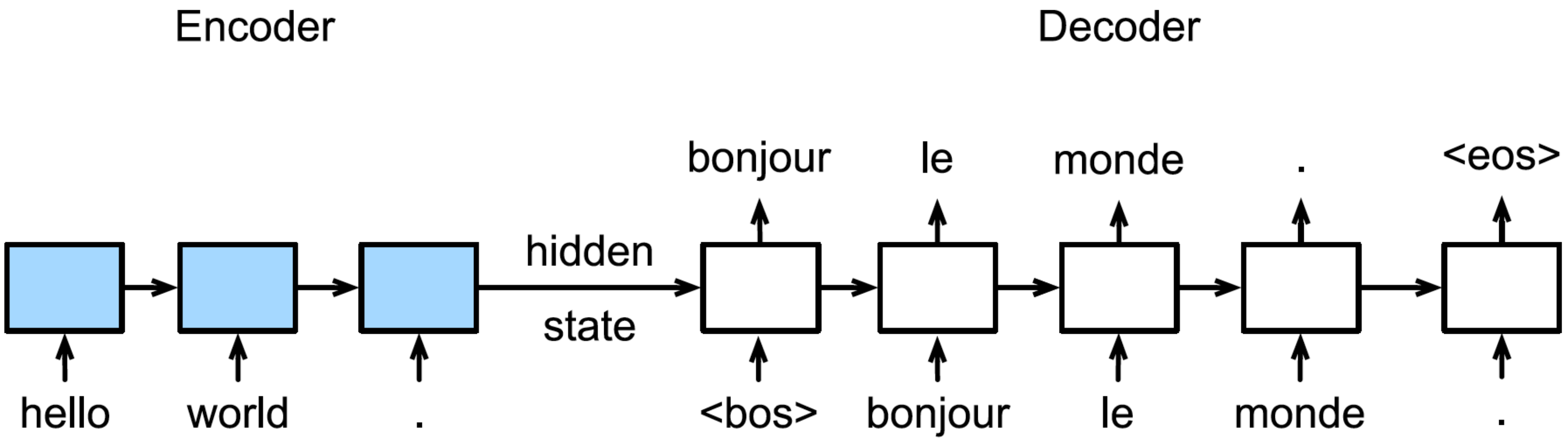


Fig. 10.14.1: The sequence to sequence model architecture.

Seq2SeqEncoder

```python
# Saved in the d2l package for later use
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)

    def forward(self, X, *args):
        X = self.embedding(X) # X shape: (batch_size, seq_len, embed_size)
        X = X.swapaxes(0, 1)  # RNN needs first axes to be time
        state = self.rnn.begin_state(batch_size=X.shape[1], ctx=X.context)
        out, state = self.rnn(X, state)
        # The shape of out is (seq_len, batch_size, num_hiddens).
        # state contains the hidden state and the memory cell
        # of the last time step, the shape is (num_layers, batch_size, num_hiddens)
        return out, state
```

Seq2SeqEncoder

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8,
                         num_hiddens=16, num_layers=2)
encoder.initialize()
X = np.zeros((4, 7))
output, state = encoder(X)
output.shape, len(state), state[0].shape, state[1].shape
```

```
((7, 4, 16), 2, (2, 4, 16), (2, 4, 16))
```

Seq2SeqDecoder

```
# Saved in the d2l package for later use
class Seq2SeqDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        X = self.embedding(X).swapaxes(0, 1)
        out, state = self.rnn(X, state)
        # Make the batch to be the first dimension to simplify loss computation.
        out = self.dense(out).swapaxes(0, 1)
        return out, state
```

Add dense layer with the hidden size to be the vocabulary size

Seq2SeqDecoder

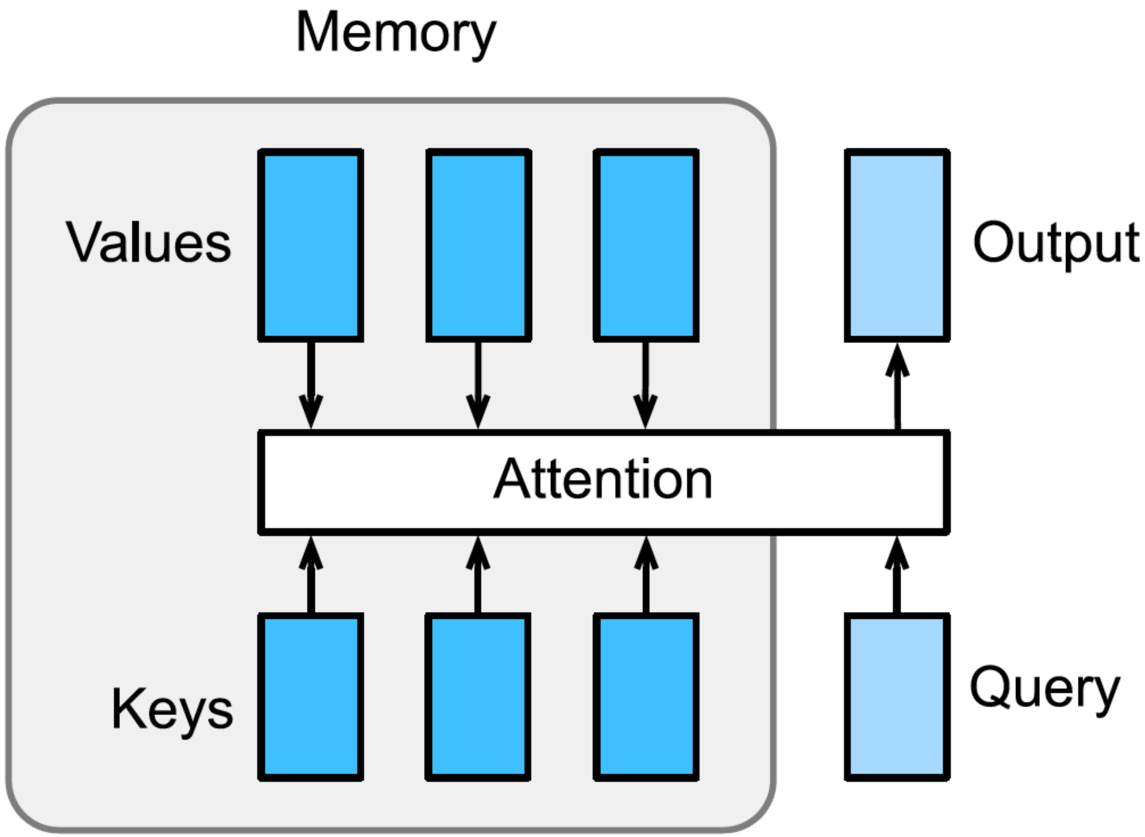
```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8,
                         num_hiddens=16, num_layers=2)
decoder.initialize()
state = decoder.init_state(encoder(X))
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, state[1].shape
```

```
((4, 7, 10), 2, (2, 4, 16), (2, 4, 16))
```

Attention Mechanism

- So far, we encode the source sequence input information in the recurrent unit state and then pass it to the decoder to generate the target sequence.
- A token in the target sequence may closely relate to some tokens in the source sequence instead of the whole source sequence.
  - For example, when translating "Hello world." to "Bonjour le monde.", "Bonjour" maps to "Hello" and "monde" maps to "world".
- In the seq2seq model, the decoder may implicitly select the corresponding information from the state passed by the decoder.
- The attention mechanism, however, makes this selection explicit.

Attention Mechanism



- Attention is a generalized pooling method with bias alignment over inputs.
- The core component in the attention mechanism is the attention layer.
- An input of the attention layer is called a query.
- For a query, the attention layer returns the output based on its memory, which is a set of key-value pairs.
- Assume a query $\mathbf{q} \in \mathbb{R}^{d_q}$
- The memory contains n key-value pairs $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)$ with $\mathbf{k}_i \in \mathbb{R}^{d_k}$, $\mathbf{v}_i \in \mathbb{R}^{d_v}$
- The attention layer then returns an output $\mathbf{o} \in \mathbb{R}^{d_v}$

Attention Mechanism

- To compute the output, we first assume there is a score function α which measures the similarity between the query and a key. Then we compute all n scores $a_1, \dots, a_n$ by

$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i).$$

- Next we use softmax to obtain the attention weights

$$b_1, \dots, b_n = \mathrm{softmax}(a_1, \dots, a_n).$$

- Then the output is the weighted sum of the values.

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i.$$

- Different choices of the score function lead to different attention layers.

Attention Mechanism

- To compute the output, we first assume there is a score function α which measures the similarity between the query and a key. Then we compute all n scores $a_1, \ldots, a_n$ by

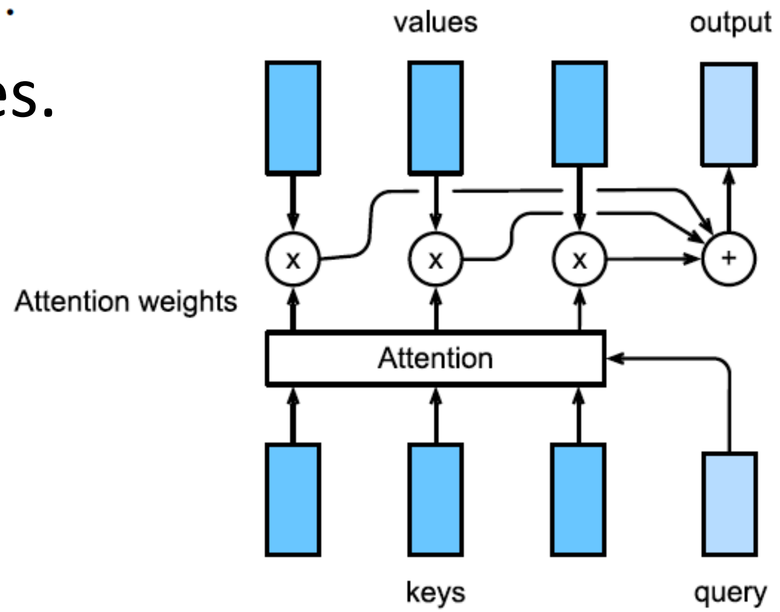
$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i).$$

- Next we use softmax to obtain the attention weights

$$b_1, \ldots, b_n = \mathrm{softmax}(a_1, \ldots, a_n).$$

- Then the output is the weighted sum of the values.

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i.$$



Dot Product Attention

- The dot product assumes the query has the same dimension as the keys, namely $\mathbf{q}, \mathbf{k}_i \in \mathbb{R}^d$
- It computes the score by an inner product between the query and a key, often then divided by $\sqrt{d}$ to make the scores less sensitive to the dimension d.

$$\alpha(\mathbf{q}, \mathbf{k}) = \langle \mathbf{q}, \mathbf{k} \rangle / \sqrt{d}.$$

- Assume $\mathbf{Q} \in \mathbb{R}^{m \times d}$ contains m queries and $\mathbf{K} \in \mathbb{R}^{n \times d}$ has all n keys. We can compute all mn scores by

$$\alpha(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}\mathbf{K}^T / \sqrt{d}.$$

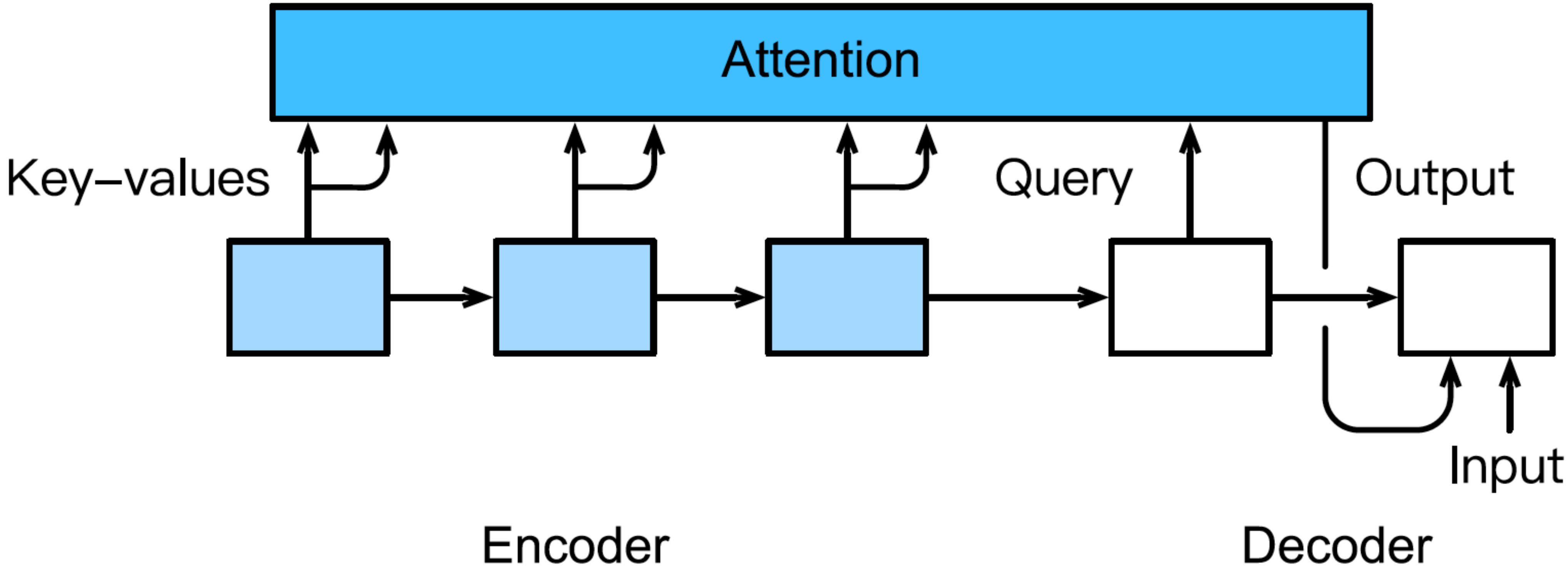
Multilayer Perceptron Attention

- In multilayer perceptron attention, we first project both query and keys into $\mathbb{R}^h$.
- Given learnable parameters $\mathbf{W}_k \in \mathbb{R}^{h \times d_k}$, $\mathbf{W}_q \in \mathbb{R}^{h \times d_q}$, and $\mathbf{v} \in \mathbb{R}^p$, the score function is defined by

$$\alpha(\mathbf{k}, \mathbf{q}) = \mathbf{v}^T \tanh(\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q}).$$

Sequence to Sequence with Attention Mechanism

- Now, we add the attention mechanism to the sequence to sequence model.
- The memory of the attention layer consists of the encoder outputs of each time step.
- During decoding, the decoder output from the previous time step is used as the query, the attention output is then fed into the decoder.



Illustrate on white board

Sequence to Sequence with Attention Mechanism

- Read more on

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

Transformer

- So far, we covered CNNs and RNNs. Let’s recap pros and cons:
  - CNNs are easy to parallelize at a layer but cannot capture sequential dependency very well.
  - RNNs are able to capture the long-range, variable-length sequential information, but suffer from inability to parallelize within a sequence.
- To combine the advantages from both RNNs and CNNs, Vaswano et al designed a novel architecture using the attention mechanism.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

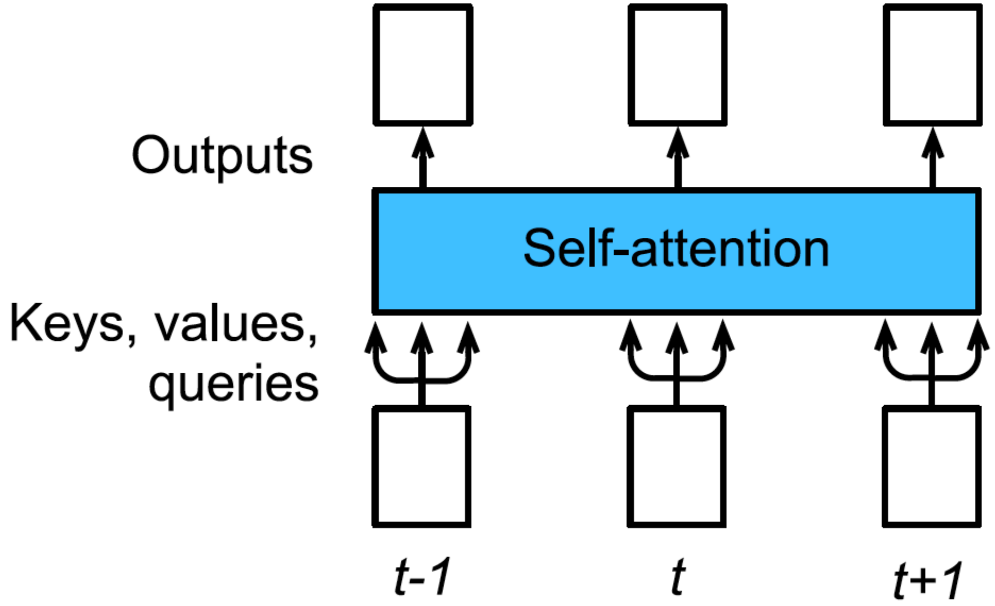
Aidan N. Gomez*† University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin*‡ illia.polosukhin@gmail.com

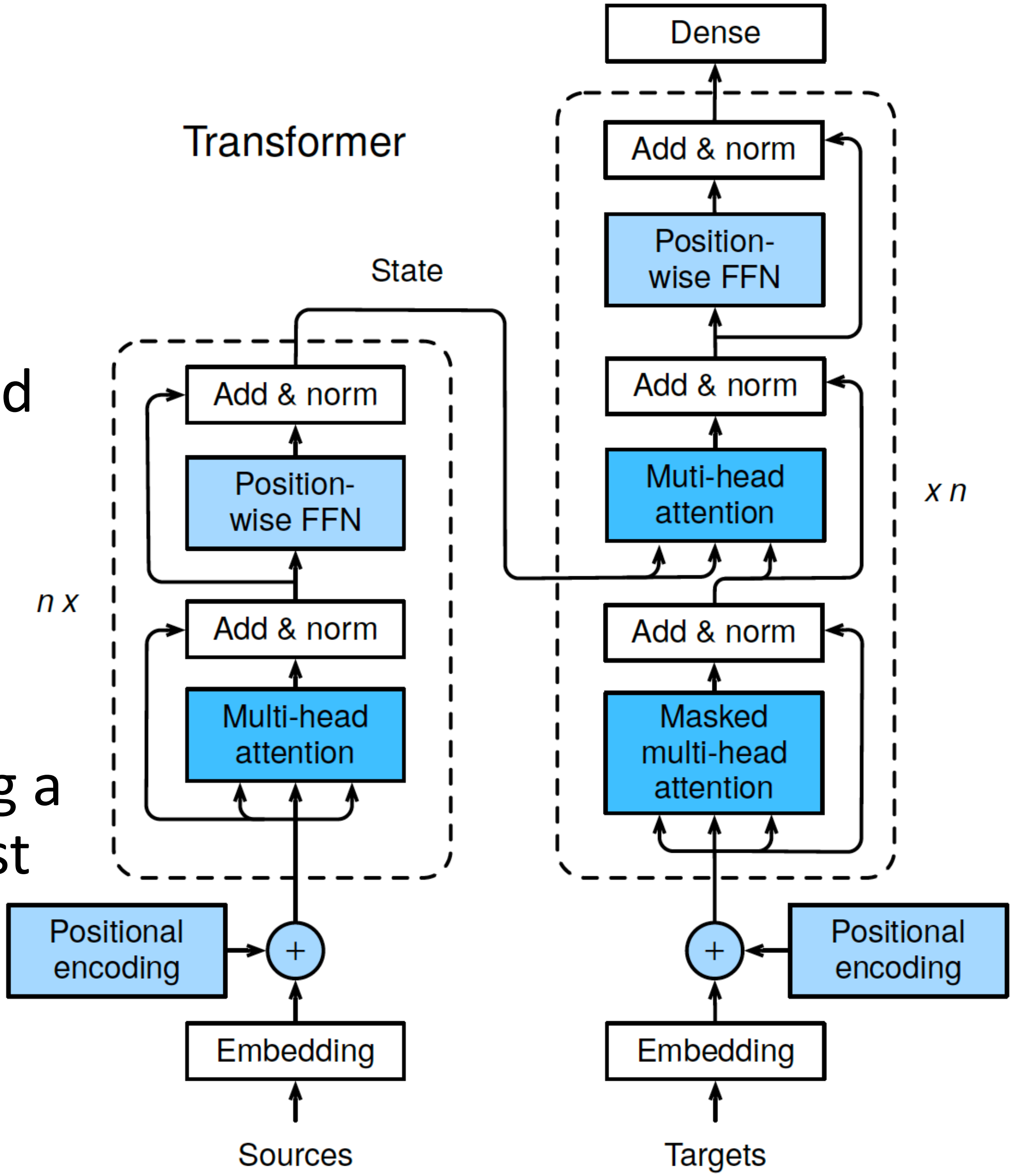
Transformer

- Transformer achieves parallelization by capturing recurrent sequence with attention and at the same time encodes each item's position in the sequence.
- The Transformer model is also based on the encoder-decoder architecture.
- The transformer replaces the recurrent layers in seq2seq with multi-head attention layers.
- Each item in the sequential is copied as the query, the key and the value.
- We call such an attention layer as a self-attention layer.



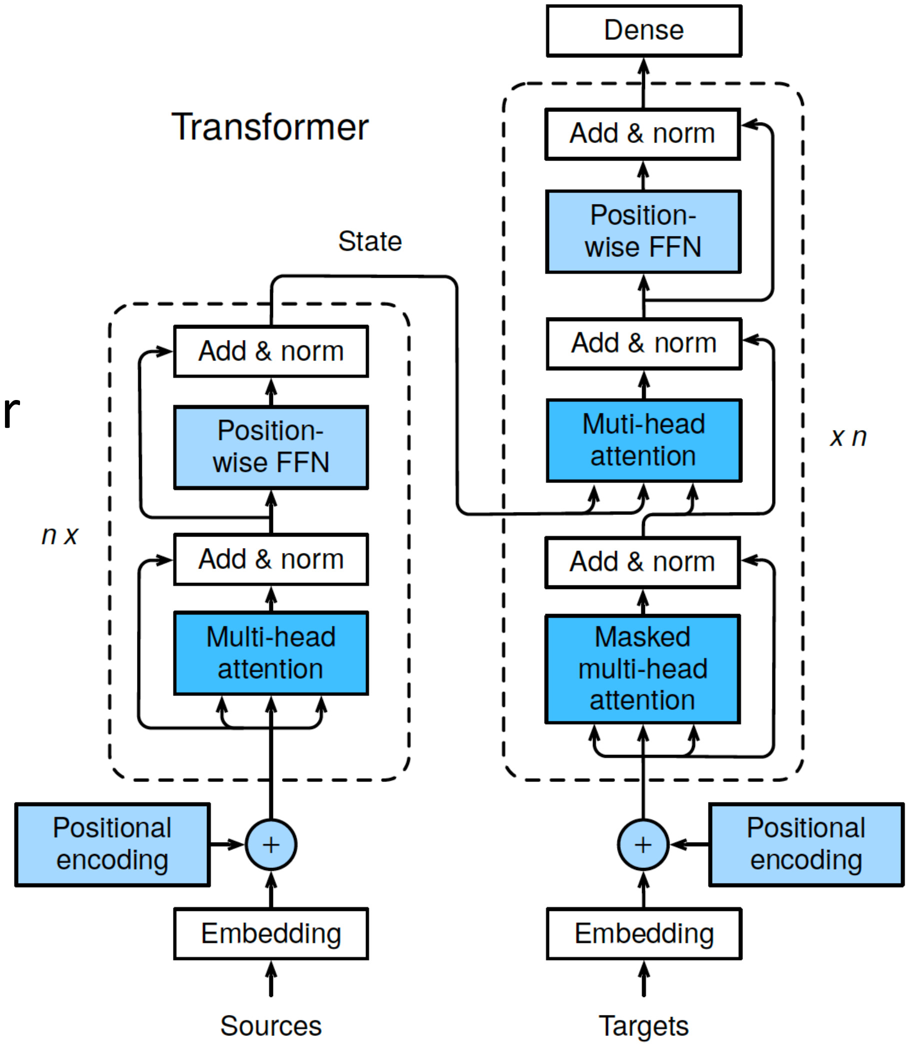
Transformer

- The source sequence embeddings are fed into n repeated blocks.
- The outputs of the last block are then used as attention memory for the decoder.
- The target sequence embeddings are similarly fed into n repeated blocks in the decoder.
- The final outputs are obtained by applying a dense layer with vocabulary size to the last block's outputs.



Transformer versus seq2seq with attention

1. **Transformer Block**: A recurrent layer in seq2seq is replaced by a Transformer block. This contains:
   - A multi-head attention layer (in encoder)
   - A network with position-wise feed-forward network layers (in encoder)
   - Another multi-head attention layer is used to take the encoder state in decoder
2. **Add and norm**: The inputs and outputs of both the multi-head attention layer or the position-wise feed forward network are processed by two "add and norm" layer that contains residual structure and a layer normalization layer.
3. **Position encoding**: Since the self-attention layer does not distinguish the item order in a sequence, a positional encoding layer is used to add sequential information into each sequence item.



Self-Attention

- The self-attention is a normal attention model, with its query, its key, and its value being copied exactly the same from each item of the sequential inputs.
- Self-attention outputs same-length sequential output for each input item.
- Compared to a recurrent layer, output items of a self-attention layer can be computed in parallel.

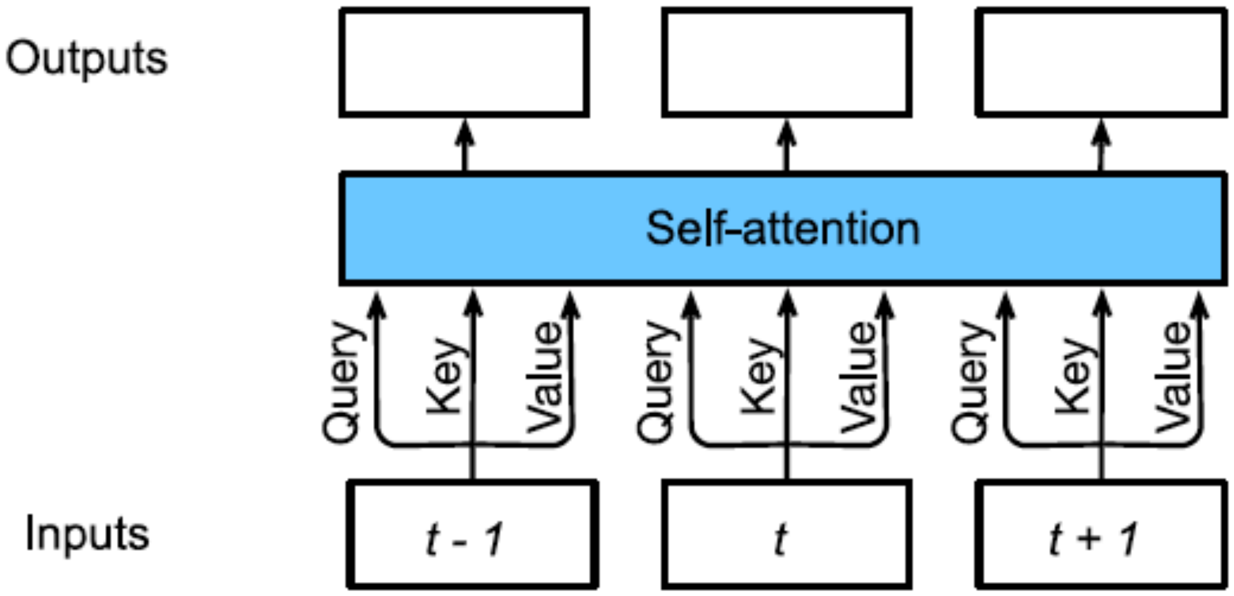
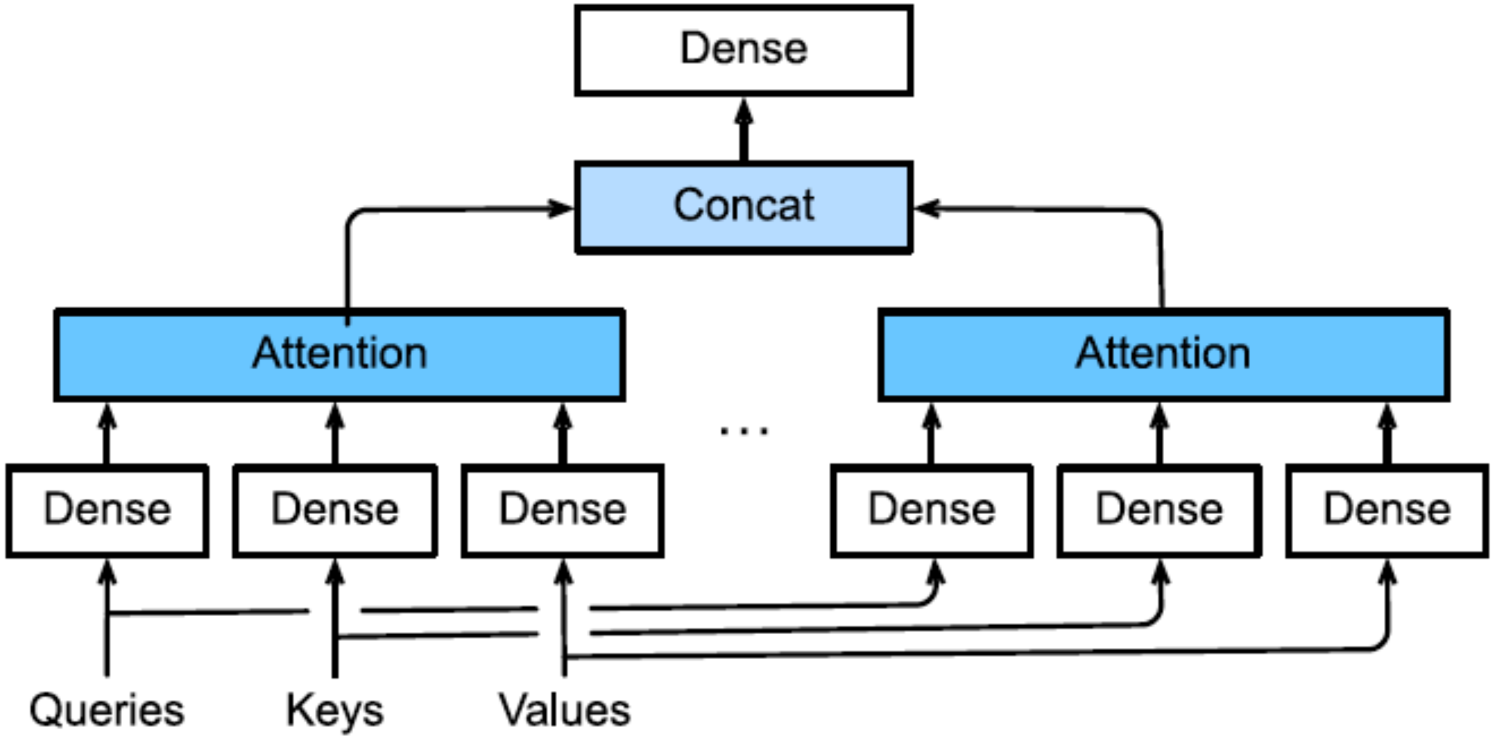


Fig. 10.3.2: Self-attention architecture.

Multi-head Attention

- The multi-head attention layer consists of h parallel self-attention layers, each one is called a head.
- The outputs of these h attention heads are concatenated and then processed by a final dense layer.



Position-wise Feed-Forward Networks (FFN)

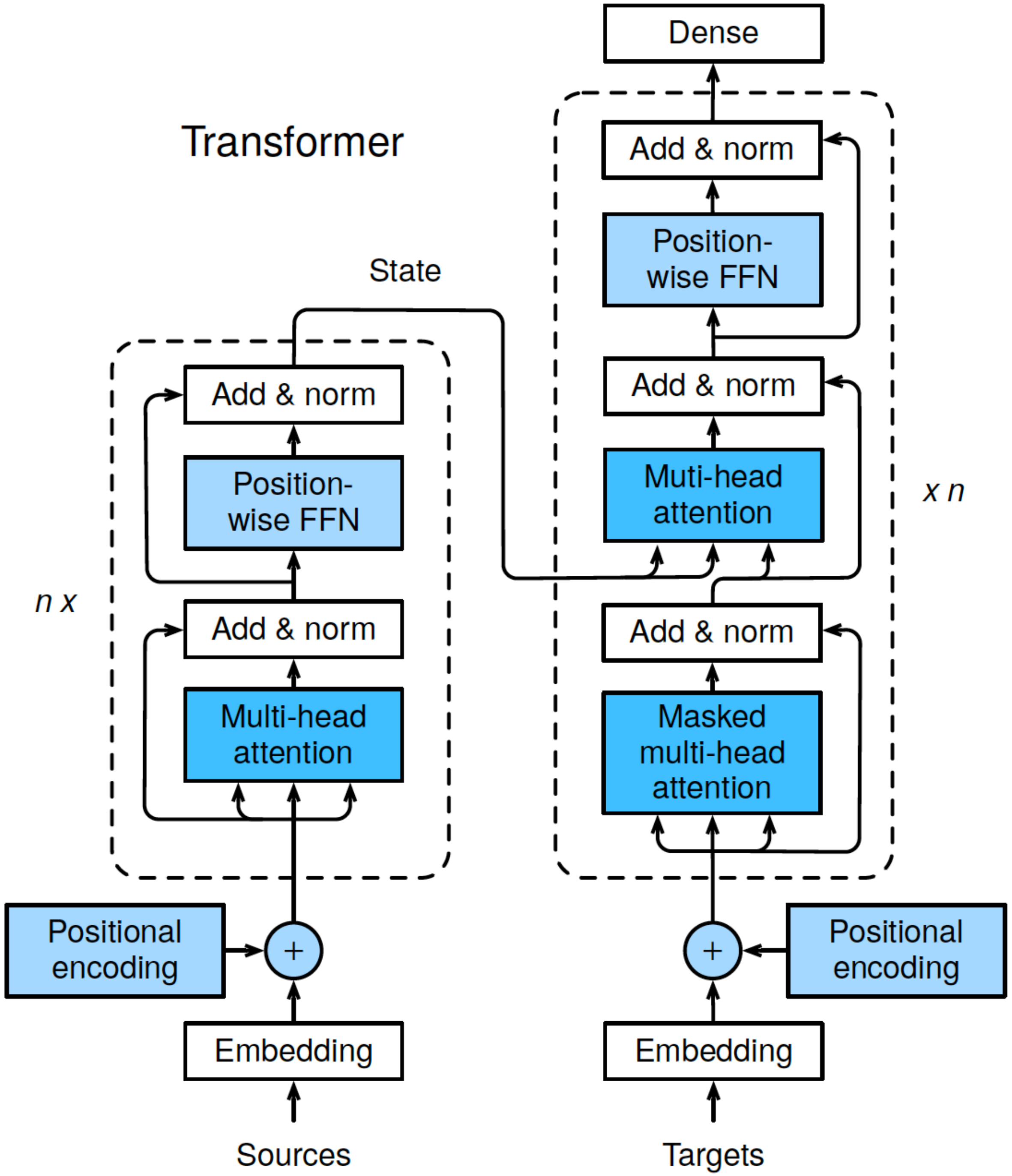
- Another key component in the Transformer block
- It accepts 3D input with shape batch_size x sequence_length x feature_size
- The position-wise FFN consists of two dense layers that applies to the last dimension.
- Since the same two dense layers are used for each position item in the sequence, we referred to it as position-wise.

```
ffn = PositionWiseFFN(4, 8)
ffn.initialize()
ffn(np.ones((2, 3, 4)))[0]
```

```
array([[ 9.15348239e-04, -7.27669394e-04,  1.14063594e-04,
        -8.76279722e-04, -1.02867256e-03,  8.02748313e-04,
        -4.53725770e-05,  2.15598906e-04],
       [ 9.15348239e-04, -7.27669394e-04,  1.14063594e-04,
        -8.76279722e-04, -1.02867256e-03,  8.02748313e-04,
        -4.53725770e-05,  2.15598906e-04],
       [ 9.15348239e-04, -7.27669394e-04,  1.14063594e-04,
        -8.76279722e-04, -1.02867256e-03,  8.02748313e-04,
        -4.53725770e-05,  2.15598906e-04]])
```

Add and Norm

- Add and norm within the block connects inputs and outputs of other layers smoothly.
- We add a layer that contains a residual structure and a layer normalization after both the multi-head attention layer and the position-wise FFN network.
- Layer normalization is similar to batch normalization.
- One difference is that the mean and variances for the layer normalization are calculated along the last dimension.



Positional Encoding

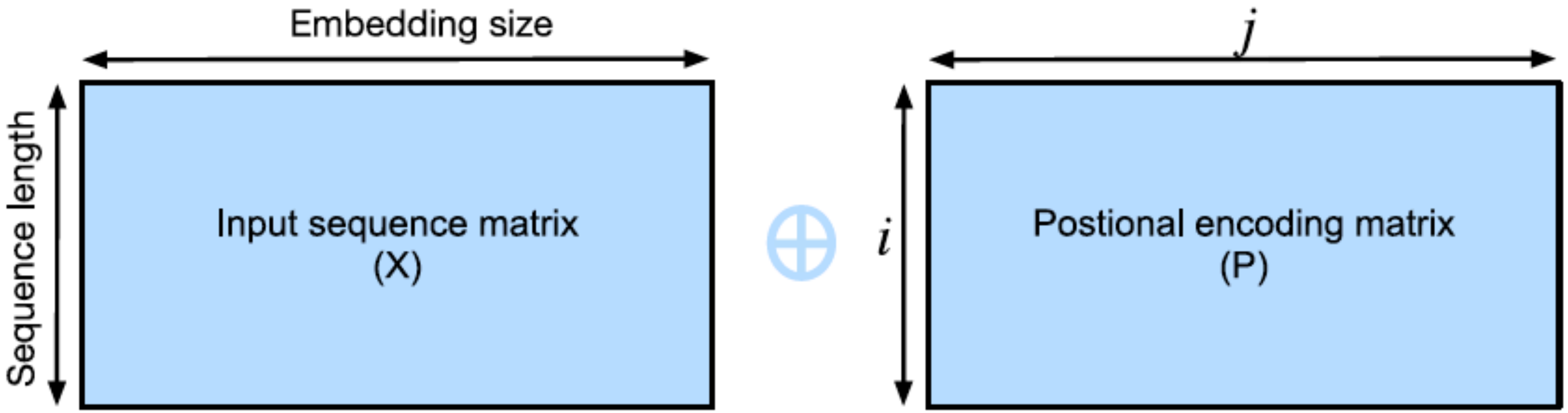
- Both the multi-head layer and position-wise feed-forward layer compute the output of each item in the sequence independently.
- It enables us to parallelize the computation, but it fails to model the sequential information for a given sequence.
- To capture the sequence, the Transformer model uses the positional encoding.
- Assume that $X \in R^{l \times d}$ is the embedding of a single example
  - l: sequence length
  - d: embedding size
- The positional encoding layer encodes X's position $P \in R^{l \times d}$ and outputs $P + X$

Positional Encoding

- The position P is a 2D matrix where
  - Each row refers to the position along the embedding vector dimension
  - Each column refers to the position along the embedding dimension
- P is obtained by using the equations below:

$$P_{i,2j} = \sin(i/10000^{2j/d}),$$

$$P_{i,2j+1} = \cos(i/10000^{2j/d}),$$



Positional Encoding

```
pe = PositionalEncoding(20, 0)
pe.initialize()
Y = pe(np.zeros((1, 100, 20)))
d2l.plot(np.arange(100), Y[0, :, 4:8].T, figsize=(6, 2.5),
         legend=["dim %d" % p for p in [4, 5, 6, 7]])
```