Deep Learning & Engineering Applications

8. Recurrent Neural Networks

JIDONG J. YANG

COLLEGE OF ENGINEERING

UNIVERSITY OF GEORGIA

Types of Data

Statistics/Econometrics

- Cross-sectional data
- Time-series data
- Panel data

Machine Learning

- Tabular data
- Image data
- Sequential data (e.g., language, audio, video, etc.)

RNN: Neural Nets with Hidden States

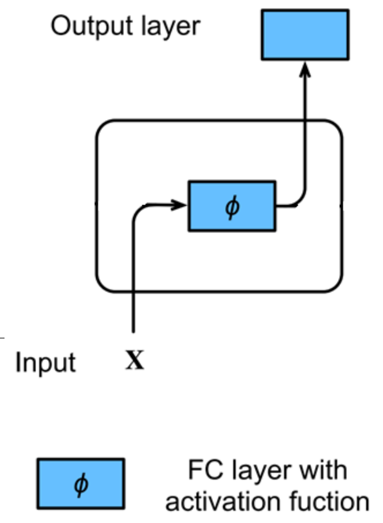
$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1})$$

h_{t-1} = hidden state (hidden variable) that stores the sequence information up to time step t−1 .

In general, the hidden state at any time step t could be computed:

$$\begin{tikzcd}[column sep=tiny] h_t \arrow[rrr, bend left] & = & f(x_t, & h_{t-1}) \end{tikzcd}$$

Neural Nets without Hidden States



Let us take a look at a MLP with a single hidden layer

Given a minibatch of examples $\mathbf{X} \in \mathbb{R}^{n\times d}$ with batch size n and d inputs and activation function ϕ

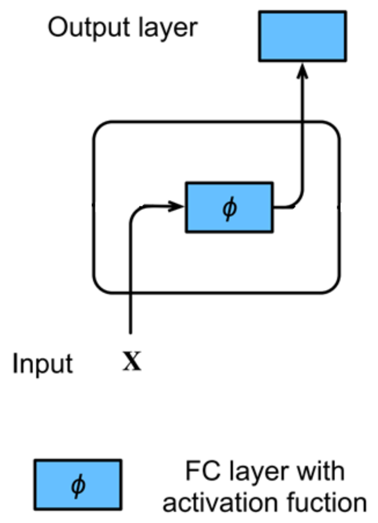
The hidden layer's output $\mathbf{H} \in \mathbb{R}^{n\times h}$ $\quad \mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h)$ $\quad \mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$ $\quad \mathbf{b}_h \in \mathbb{R}^{1\times h}$

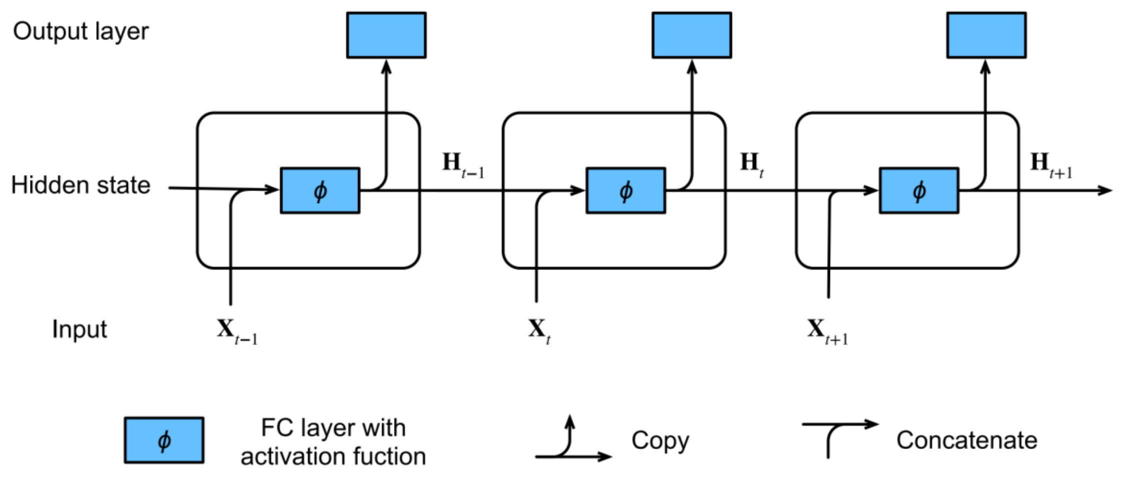
The output layer is given by $\quad \mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q$ $\quad \mathbf{O} \in \mathbb{R}^{n\times q}$ $\quad \mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ $\quad \mathbf{b}_q \in \mathbb{R}^{1\times q}$

Note: This is analogous to the regression problems for nonsequential data. We can arrange the data as feature-label pairs at random and learn the parameters of our network via stochastic gradient descent.

Neural nets without Hidden States

Neural nets with Hidden States





$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h)$$

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

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

$$\mathbf{H_t} = \boldsymbol{\phi}\left([\mathbf{X_t}\ \mathbf{H_{t-1}}]\begin{bmatrix}\mathbf{W_{xh}}\\ \mathbf{W_{hh}}\end{bmatrix} + \mathbf{b_h}\right)$$

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

RNN

RNNs use same model parameters across different time steps, often referred to as “**parameters sharing**”, which is similar to CNNs.

As such, the parameterization cost of a RNN does not grow as the number of time steps increases.

Backpropagation Through Time (BPTT)

$$h_t = f(x_t, h_{t-1}, w_h)$$

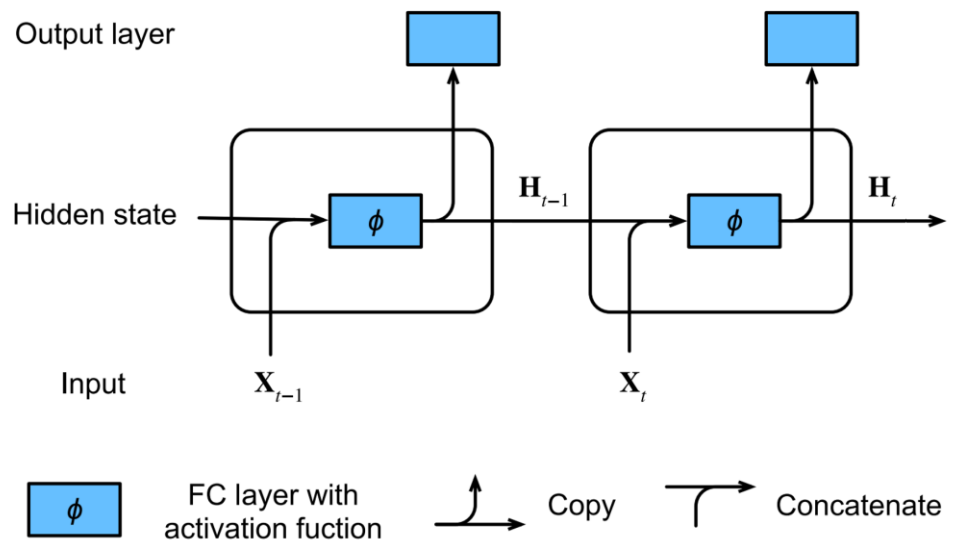
$$o_t = g(h_t, w_o)$$

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$$

$$\frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial g(h_t, w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}$$

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}$$

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}}\right)\frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h}$$



BPTT – a simple case

Let us consider an RNN without bias parameters and the activation function in the hidden layer being the identity mapping, i.e., $\phi(x) = x$.

For time step t, consider a single example input $x_t \in R^d$ and the corresponding label y_t.

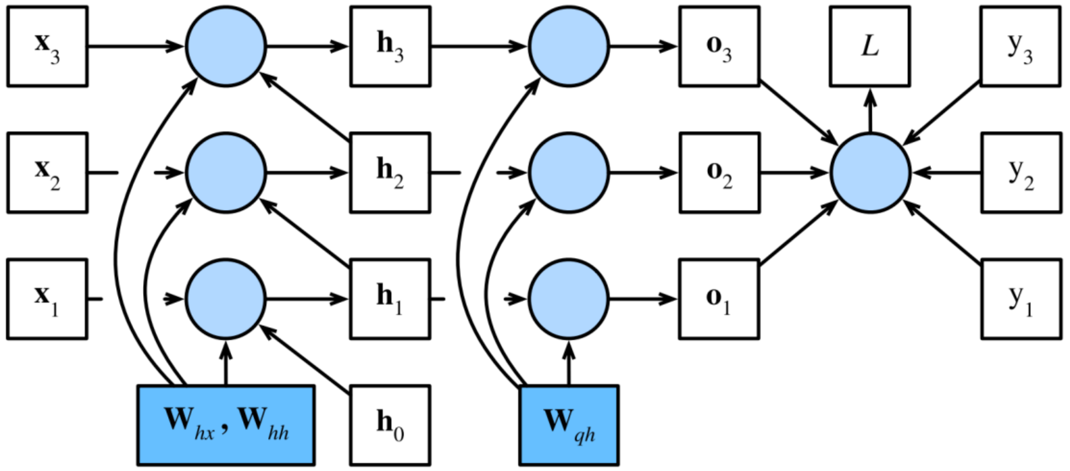
$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} \qquad L = \frac{1}{T}\sum_{t=1}^{T} l(\mathbf{o}_t, y_t)$$

$$\mathbf{o}_t = \mathbf{W}_{qh}\mathbf{h}_t$$

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{hh}^\top\right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}$$

$$\frac{\partial L}{\partial \mathbf{W}_{hx}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t}\mathbf{x}_t^\top$$

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t}\mathbf{h}_{t-1}^\top$$



Computational graph of RNN with three timesteps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

- Large powers of $\mathbf{W}_{\mathbf{hh}}^{\mathbf{T}}$: numerically unstable.
- Eigenvalues < 1 vanish while eigenvalues > 1 diverge (vanishing and exploding gradients).
- Truncation is needed, referred to as truncated backpropgation through time.

Modern RNNs

Gated Recurrent Units (GRU)

Long Short-Term Memory (LSTM)

Liquid Time-Constant (LTC) Networks

GRU Model

- Reset gates help capture short-term dependencies in sequences.
- Update gates help capture long-term dependencies in sequences

1. Compute reset gate and update gate.

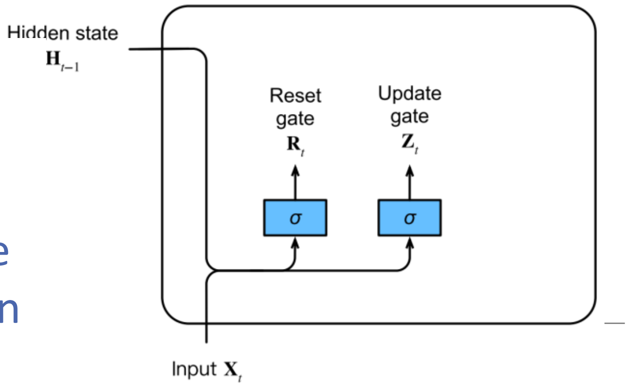
$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r)$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)$$

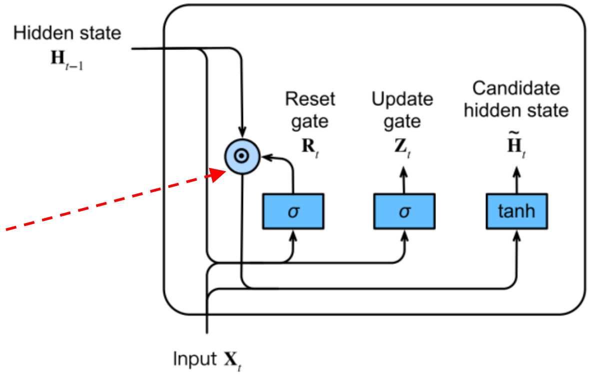
2. Compute the candidate hidden state.

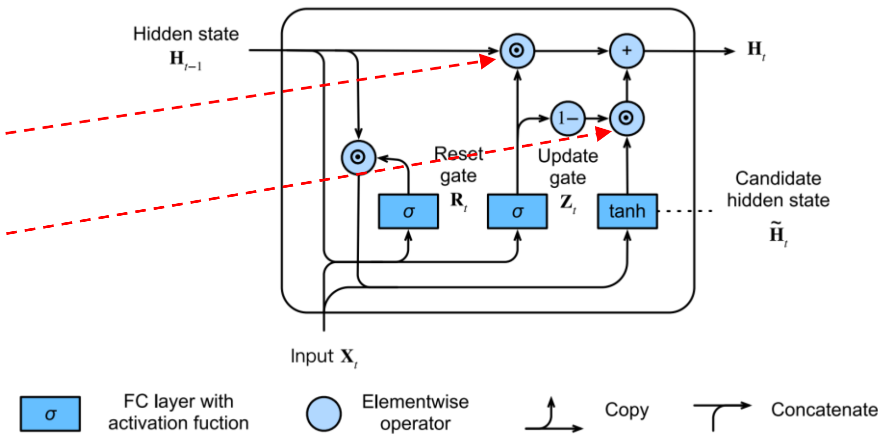
$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \boxed{(\mathbf{R}_t \odot \mathbf{H}_{t-1})} \mathbf{W}_{hh} + \mathbf{b}_h)$$

3. Compute the hidden state.

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







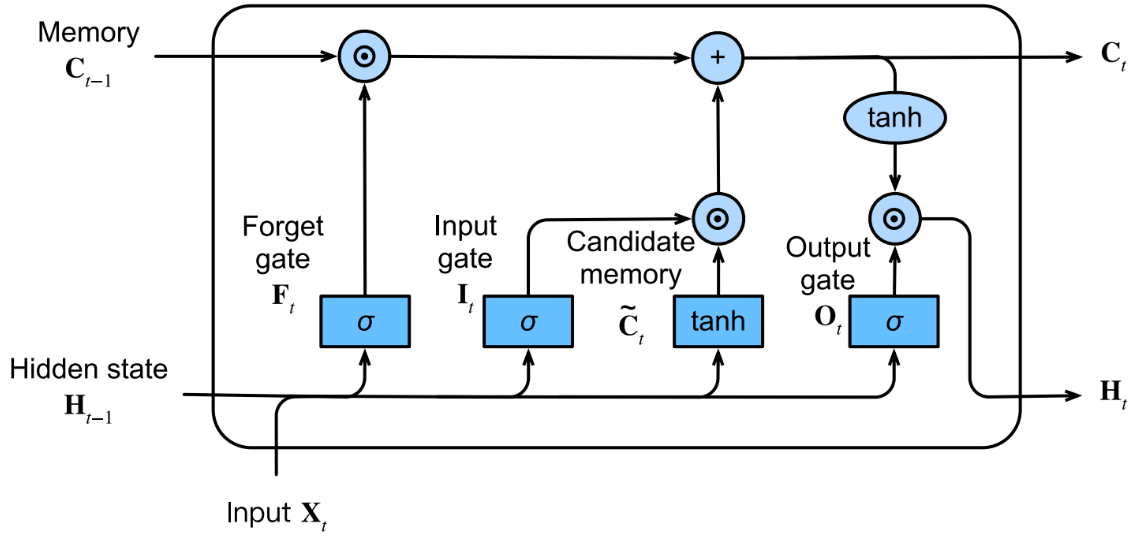
LSTM

Similar to logic gates of a computer.

Introduces a **memory cell** that has the same shape as the hidden state.

Controls the memory cell using a number of gates.

- Output gate to read out the entries from the cell.
- Input gate to decide when to read data into the cell.
- Forget gate to reset the content of the cell.



Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
σ
Input gate $\mathbf{I}_t$
σ
Candidate memory $\tilde{\mathbf{C}}_t$
tanh
Output gate $\mathbf{O}_t$
σ
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$

LSTM

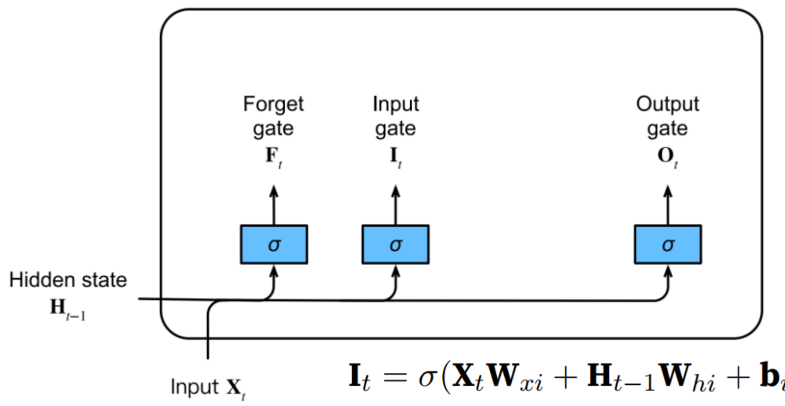
σ FC layer with activation fuction

○ Elementwise operator

Copy

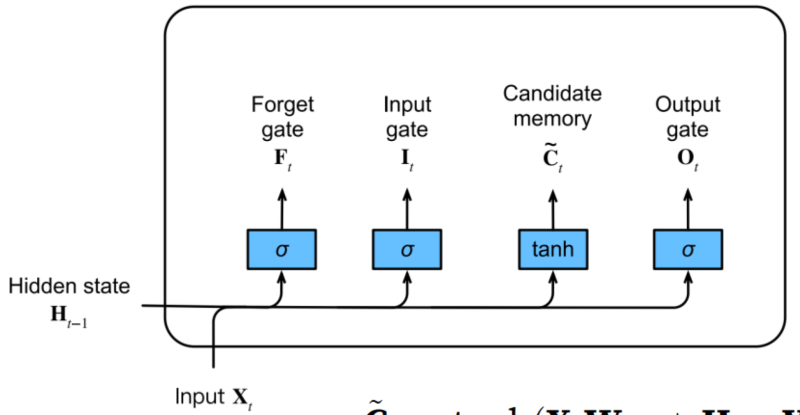
Concatenate

1. Compute the input, forget, output gates.


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Output gate $\mathbf{O}_t$
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$

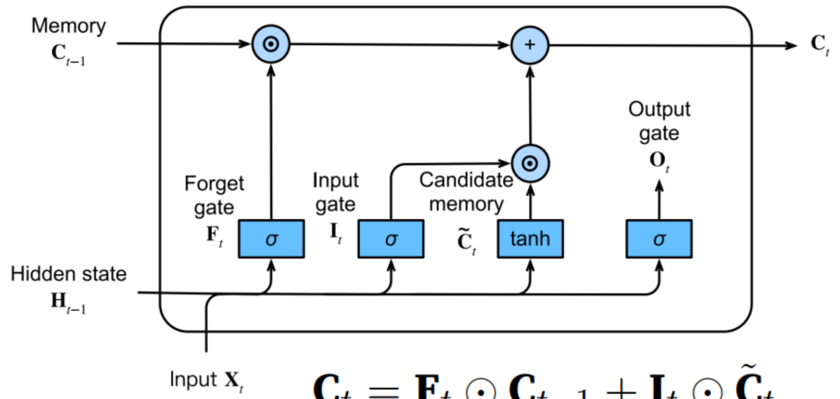
$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)$$

2. Compute the candidate memory cell.


Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$

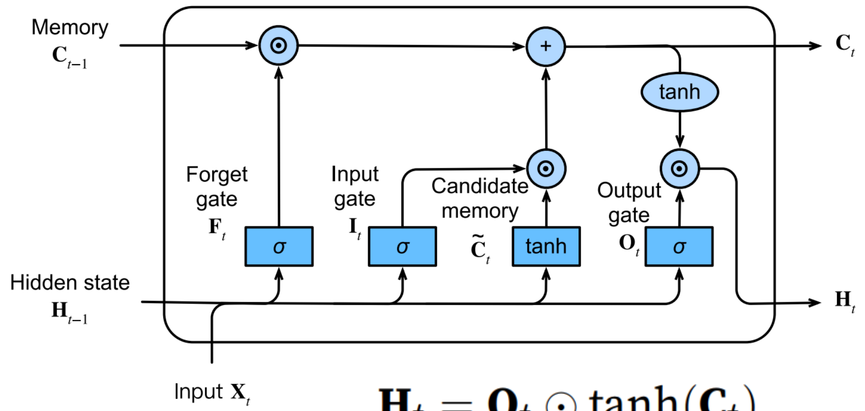
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

3. Compute the memory cell.


Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
tanh
+
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$

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

4. Compute the hidden state.


Memory $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
+
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$

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

Architecture of deep RNNs

Stack multiple layers of RNNs on top of each other, resulting in a flexible mechanism.

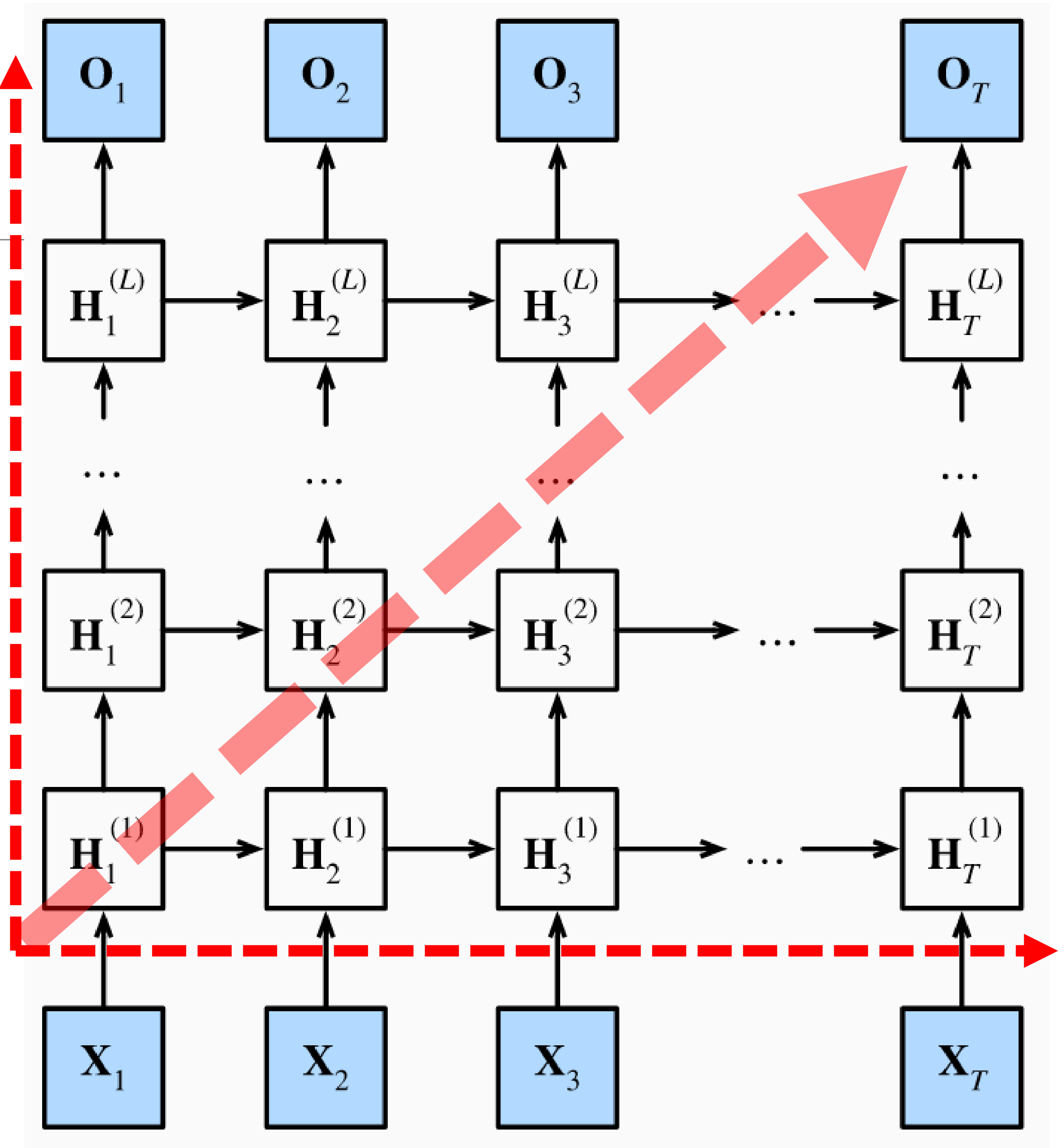
Hierarchical features

For vanilla RNN:

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

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

Extend to GRU and LSTM by replacing the hidden state computation.

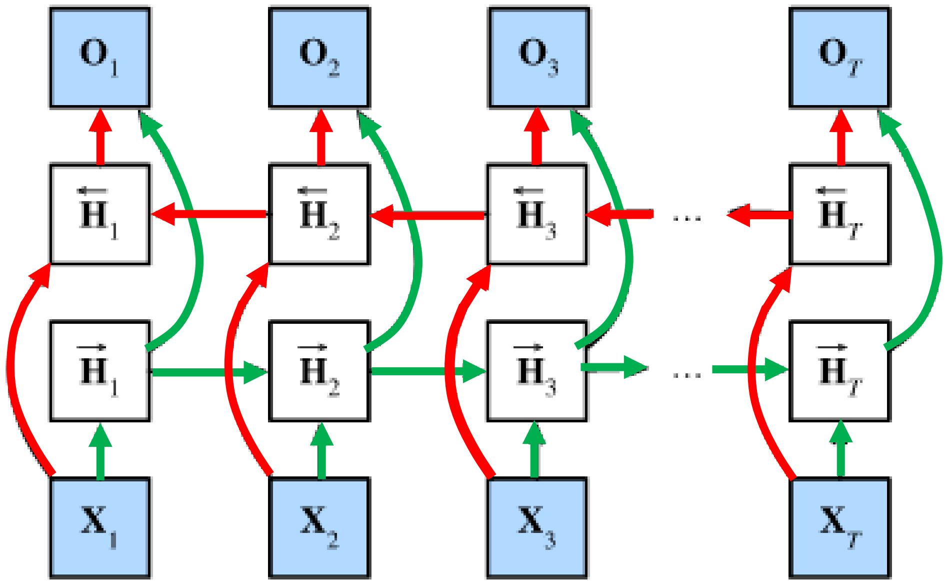


Bidirectional RNN

In bidirectional RNNs, the hidden state for each time step is simultaneously determined by the data prior to and after the current time step.

Bidirectional RNNs are mostly useful for sequence encoding and the estimation of observations given bidirectional context.

Bidirectional RNNs are very costly to train due to long gradient chains.



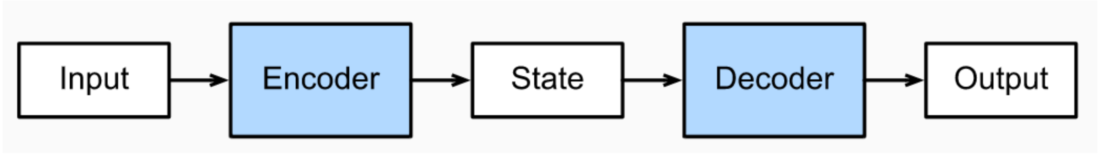
$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{O}_3$
$\mathbf{O}_T$
$\overleftarrow{\mathbf{H}}_1$
$\overleftarrow{\mathbf{H}}_2$
$\overleftarrow{\mathbf{H}}_3$
$\overleftarrow{\mathbf{H}}_T$
$\overrightarrow{\mathbf{H}}_1$
$\overrightarrow{\mathbf{H}}_2$
$\overrightarrow{\mathbf{H}}_3$
$\overrightarrow{\mathbf{H}}_T$
$\mathbf{X}_1$
$\mathbf{X}_2$
$\mathbf{X}_3$
$\mathbf{X}_T$

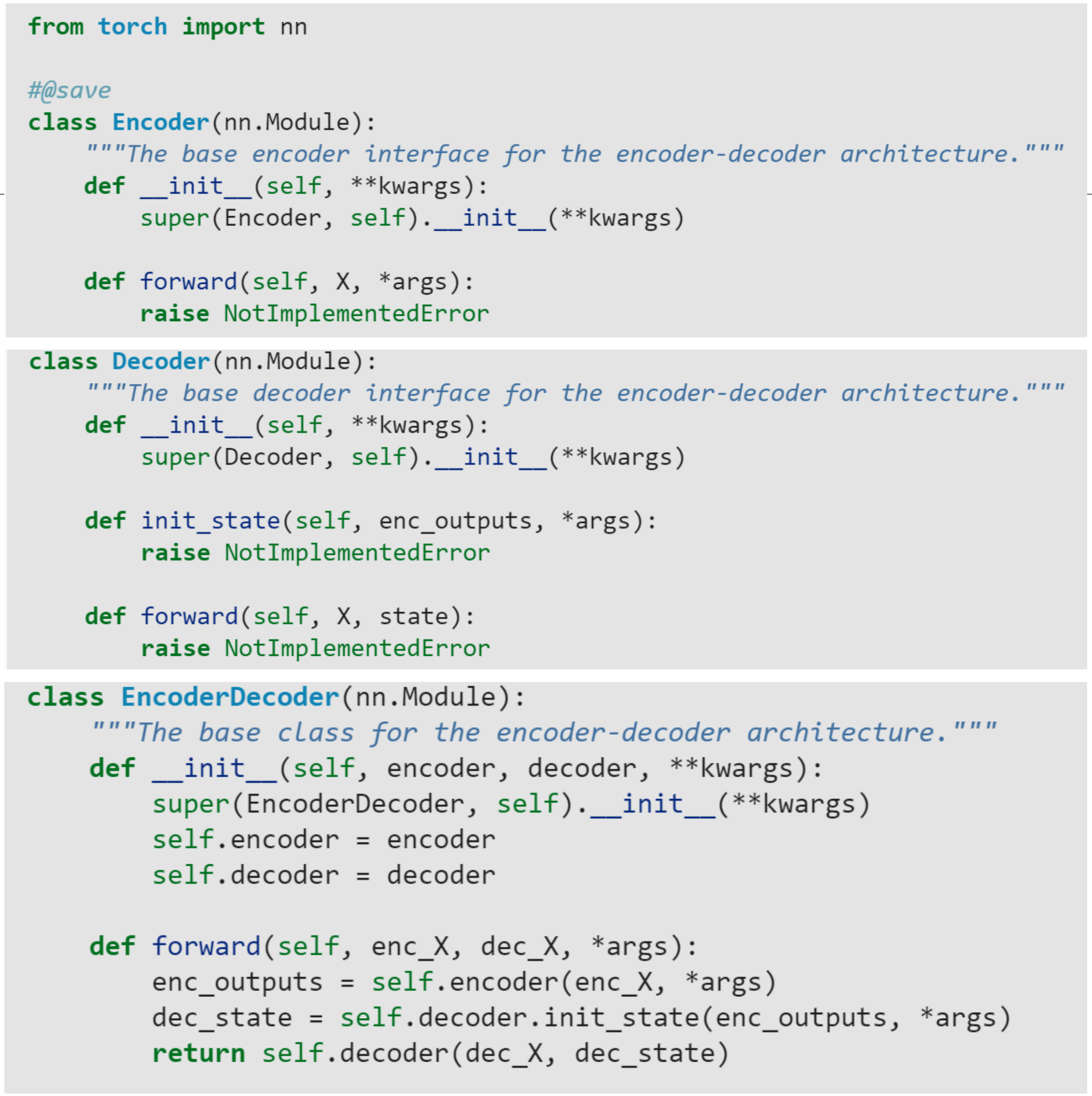
$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1}\mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)})$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1}\mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)})$$

$$\mathbf{H}_t = [\overrightarrow{\mathbf{H}}_t\ \overleftarrow{\mathbf{H}}_t]$$

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

Encoder-Decoder Architecture





```python
from torch import nn

#@save
class Encoder(nn.Module):
    """The base encoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

```python
class Decoder(nn.Module):
    """The base decoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

```python
class EncoderDecoder(nn.Module):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

Terminology
- Hidden Layers
- Hidden Units (Hidden State)
- Time Steps

The encoder-decoder architecture can handle inputs and outputs that are both variable-length sequences, thus is suitable for sequence transduction problems such as machine translation.

The encoder takes a variable-length sequence as the input and transforms it into a state with a fixed shape.

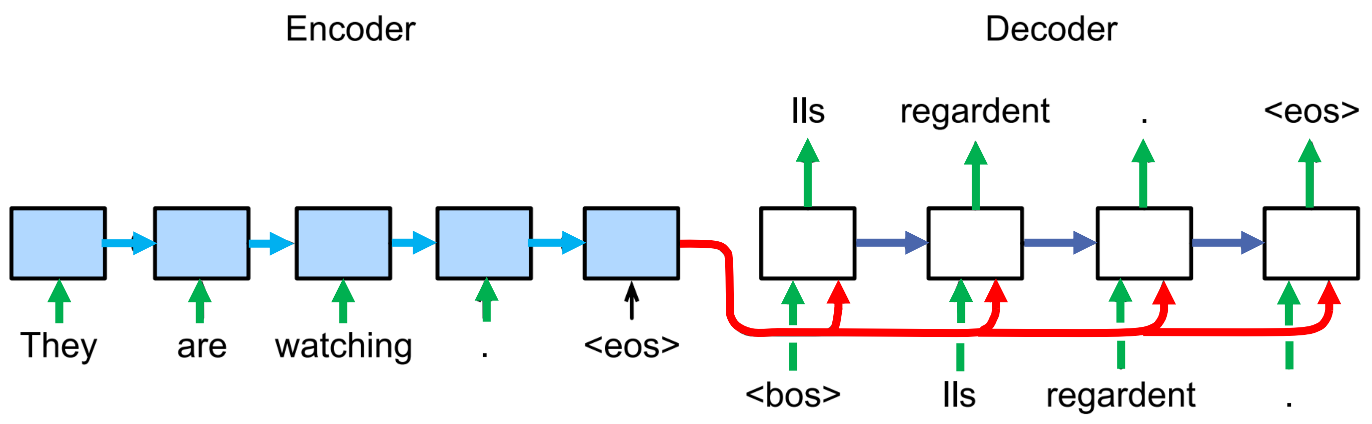
The decoder maps the encoded state of a fixed shape to a variable-length sequence.

Sequence-to-Sequence Learning

RNN encoder-decoder

The encoder transforms (encodes) an input sequence of variable length into a fixed-shape context variable **c**.

Concatenate the context variable with the decoder input at all the time steps.



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

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

Encoder

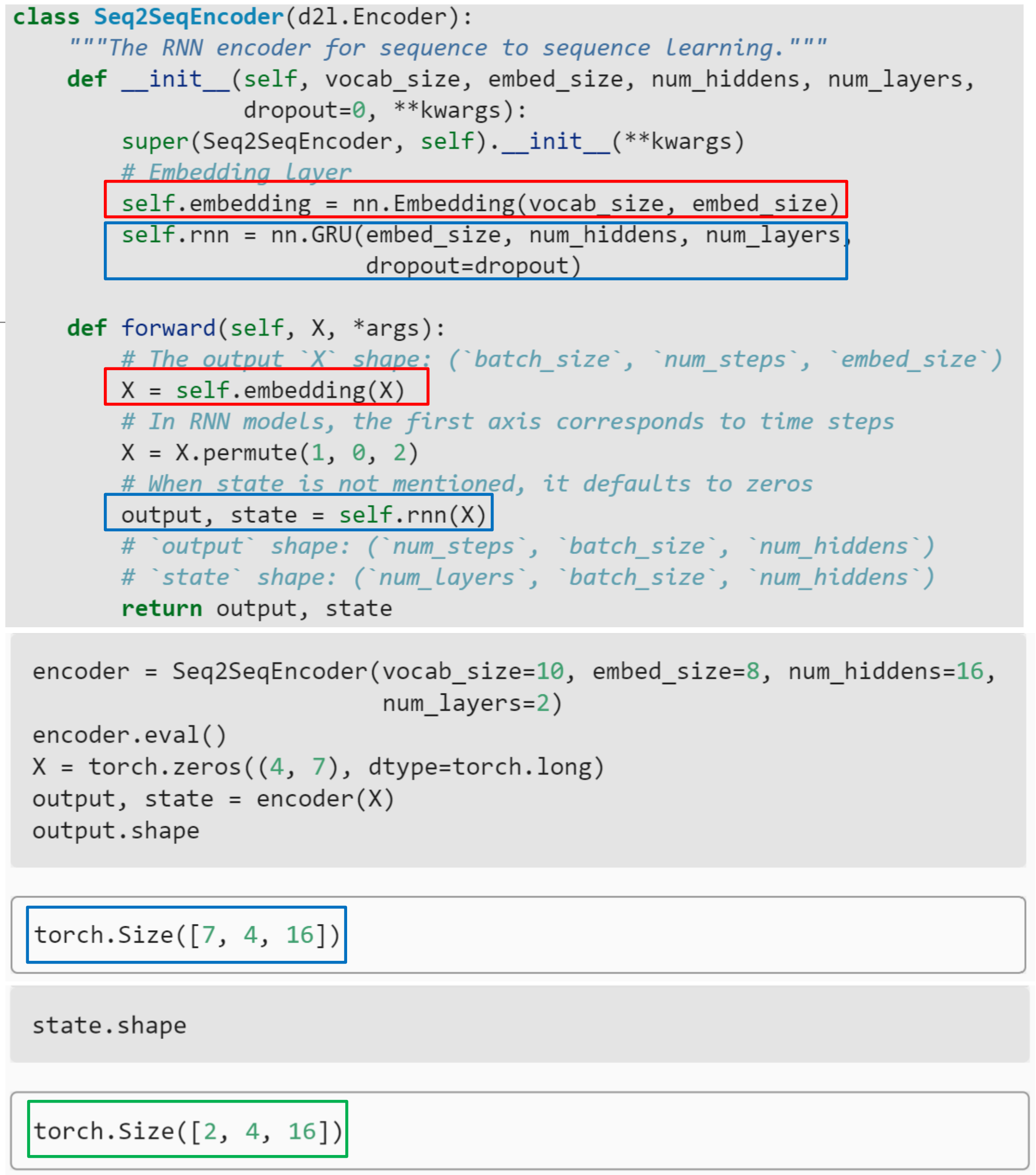
Exemplar implementation of a RNN encoder using a multilayer GRU.

The embedding layer is to obtain the feature vector for each token in the input sequence.

The weight of an embedding layer is a matrix whose number of rows equals to the size of the input vocabulary (vocab_size) and number of columns equals to the feature vector’s dimension (embed_size).

Instantiate a two-layer GRU encoder whose number of hidden units is 16. Given a minibatch of sequence inputs X (batch size=4, number of time steps=7), the output return by the encoder’s recurrent layers are a tensor of shape (number of time steps=7, batch size= 4, number of hidden units= 6).

The shape of the multilayer hidden states at the final time step is (number of hidden layers=2, batch size=4, number of hidden units=16)



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

    def forward(self, X, *args):
        # The output `X` shape: (`batch_size`, `num_steps`, `embed_size`)
        X = self.embedding(X)
        # In RNN models, the first axis corresponds to time steps
        X = X.permute(1, 0, 2)
        # When state is not mentioned, it defaults to zeros
        output, state = self.rnn(X)
        # `output` shape: (`num_steps`, `batch_size`, `num_hiddens`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)
output, state = encoder(X)
output.shape
```

```
torch.Size([7, 4, 16])
```

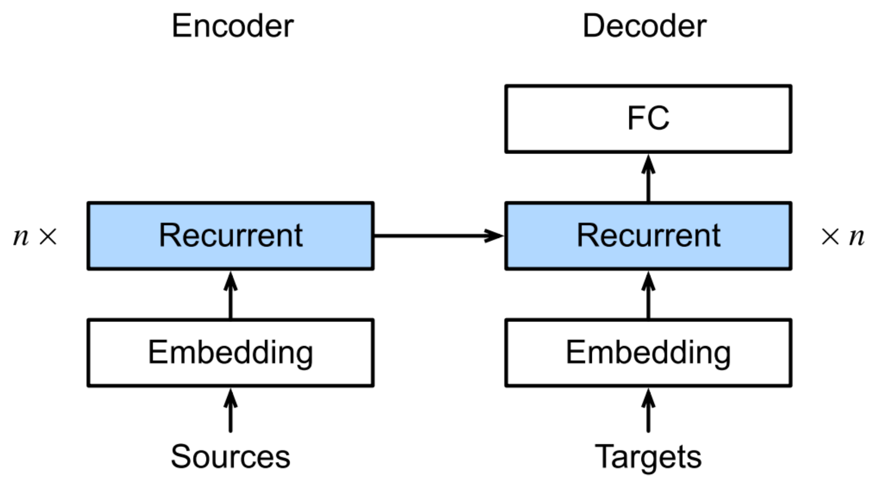
```
state.shape
```

```
torch.Size([2, 4, 16])
```

Decoder

Use the hidden state at the final time step of the encoder to initialize the hidden state of the decoder. (This requires that the RNN encoder and the RNN decoder have the same number of layers and hidden units.)

A fully-connected layer is used to transform the hidden state at the final layer of the RNN decoder to predict the probability distribution of the output token.



```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, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.GRU(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):
        # The output `X` shape: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).swapaxes(0, 1)
        # `context` shape: (`batch_size`, `num_hiddens`)
        context = state[0][-1]
        # Broadcast `context` so it has the same `num_steps` as `X`
        context = np.broadcast_to(
            context, (X.shape[0], context.shape[0], context.shape[1]))
        X_and_context = np.concatenate((X, context), 2)
        output, state = self.rnn(X_and_context, state)
        output = self.dense(output).swapaxes(0, 1)
        # `output` shape: (`batch_size`, `num_steps`, `vocab_size`)
        # `state[0]` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

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

```
((4, 7, 10), 1, (2, 4, 16))
```

Neural ODE models

Standard RNN:
[Hopfield 1982]

$$X(t+1) = f(x(t), I(t), t; \theta)$$

Neural ODE:
[Chen et al. 2018]

$$\frac{dx(t)}{dt} = f(x(t), I(t), t; \theta)$$

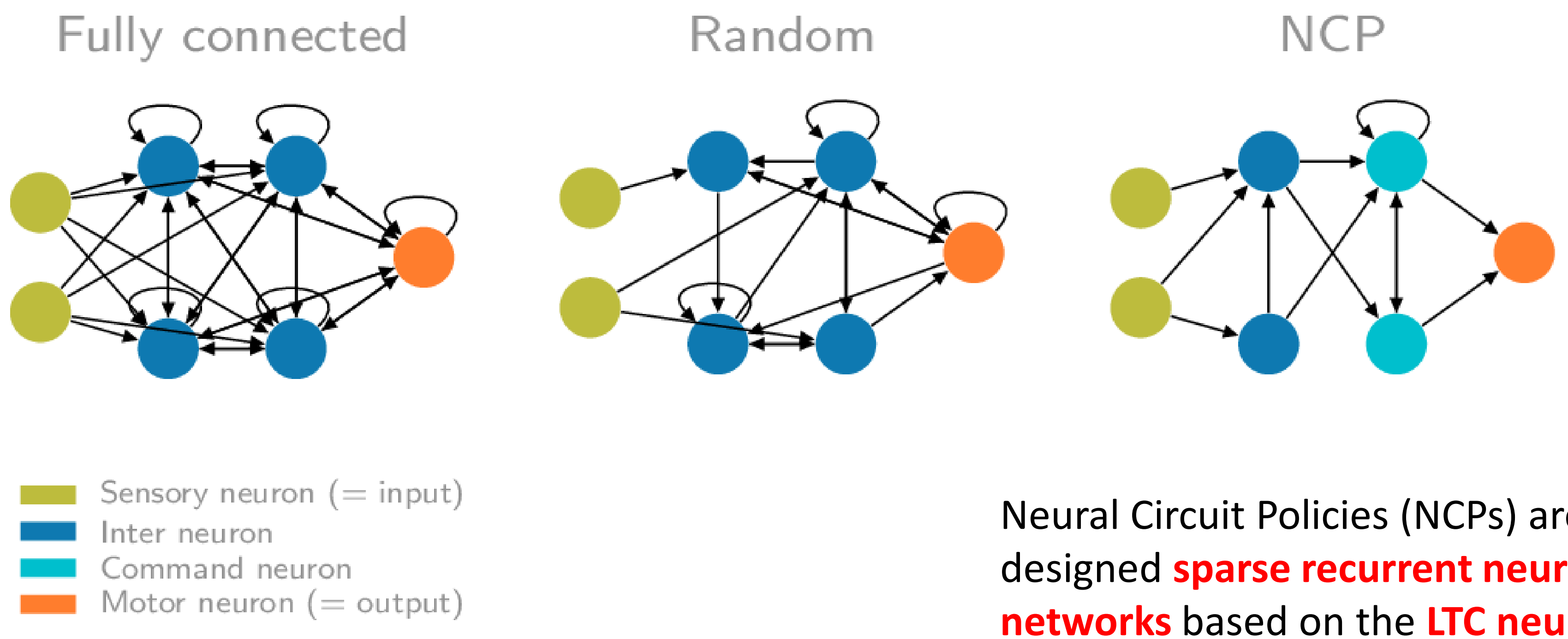
Continuous Time (CT) RNN:
[Funahashi et al. 1993]

$$\frac{dx(t)}{dt} = -\frac{x(t)}{\tau} + f(x(t), I(t), t; \theta)$$

Liquid Time Constant (LTC) Network:
[Hasani et al., 2020]

$$\frac{dx(t)}{dt} = -\frac{x(t)}{\tau_{sys}} + f(x(t), I(t), t, \theta)A \qquad \tau_{sys} = \frac{\tau}{1+\tau f(x(t), I(t), t, \theta)}$$

Liquid Time-Constant (LTC) Networks

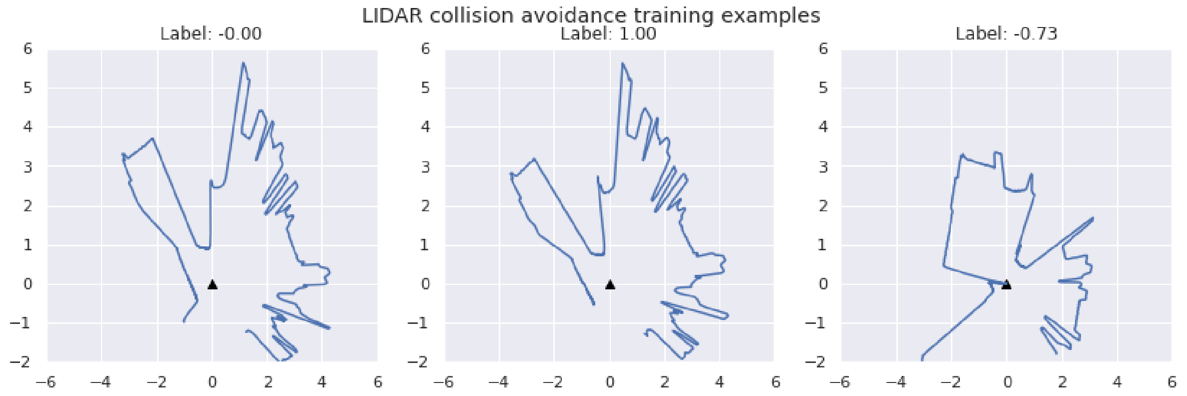


Neural Circuit Policies (NCPs) are designed **sparse recurrent neural networks** based on the **LTC neuron**, which is modeled by an **ODE**.

[Lechner et al., 2018] https://arxiv.org/pdf/1803.08554.pdf https://github.com/mlech26l/keras-ncp

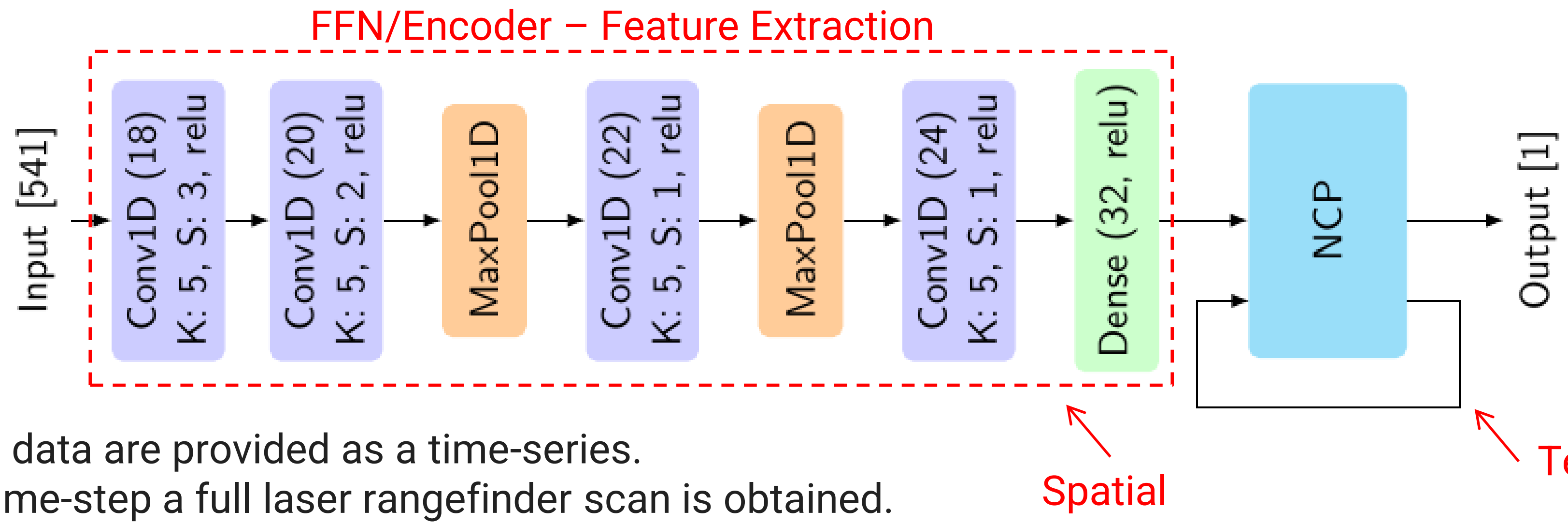
[Hasani et al., 2020] https://arxiv.org/pdf/2006.04439.pdf

LTC



Work as a great recurrent "head" network on the top of features extracted.

Example: Control a mobile robot using LiDAR scan as input to avoid obstacles in its path.


FFN/Encoder – Feature Extraction
Input [541]
Conv1D (18) K: 5, S: 3, relu
Conv1D (20) K: 5, S: 2, relu
MaxPool1D
Conv1D (22) K: 5, S: 1, relu
MaxPool1D
Conv1D (24) K: 5, S: 1, relu
Dense (32, relu)
NCP
Output [1]
Spatial

https://github.com/mlech26l/keras-ncp

1. The input data are provided as a time-series.
2. At each time-step a full laser rangefinder scan is obtained.
3. The LIDAR scan is fed through the FFN/Encoder to obtain a low-dimensional (32-dimensional) latent representation. (Note: the **same set of weights in Encoder** is used at each time step.)
4. The LTC (NCP) takes this latent feature as input and updates its internal state and output prediction.