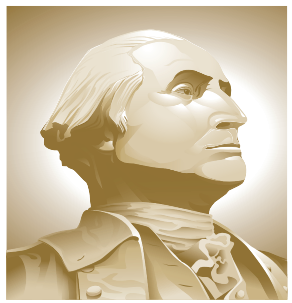
Recurrent Neural Network (RNN)
LSTM & GRU

Natural Language Processing
Lecture 8



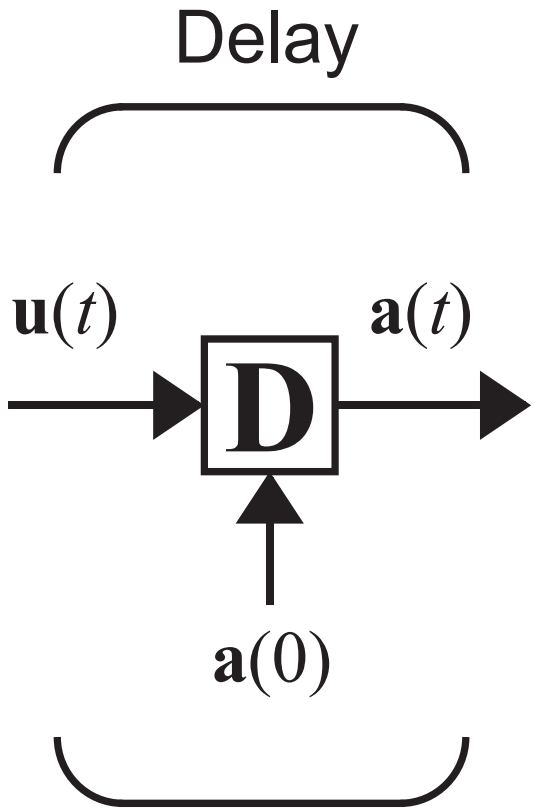
THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Dynamic Networks

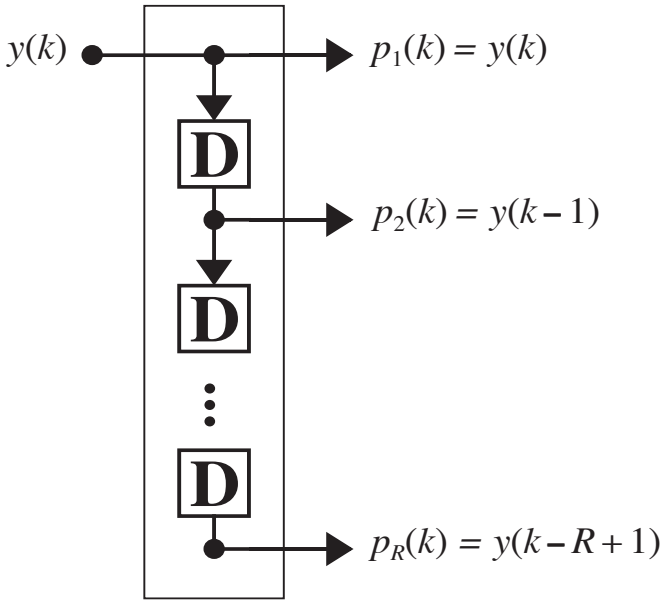
- Dynamic networks are networks that contain delays and that operate on a sequence of inputs.
- The ordering of the inputs is important to the operation of the network.
- In dynamic networks, the output depends not only on the current input to the network, but also on the current or previous inputs, outputs or states of the network.

- It store pervious data in.

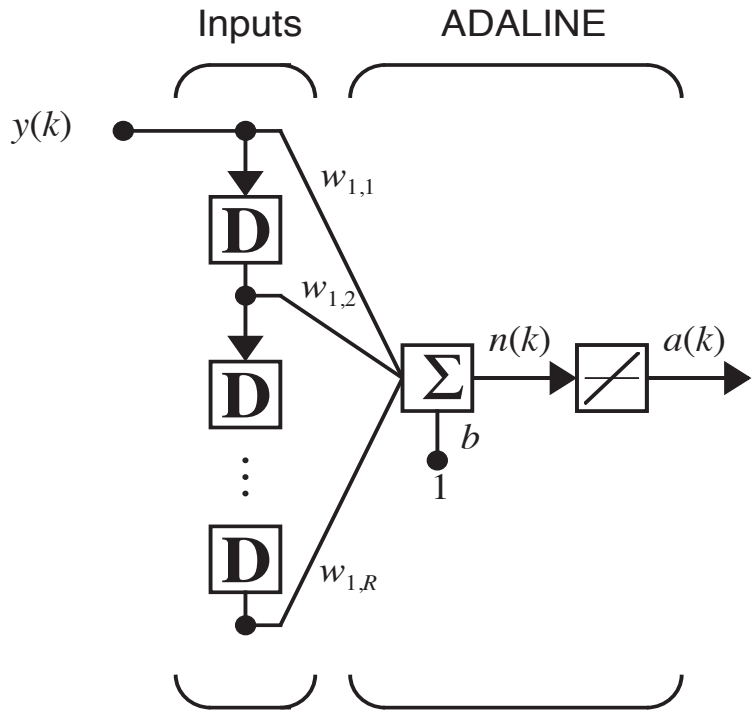


$$\mathbf{a}(t) = \mathbf{u}(t - 1)$$

Tapped Delay Line

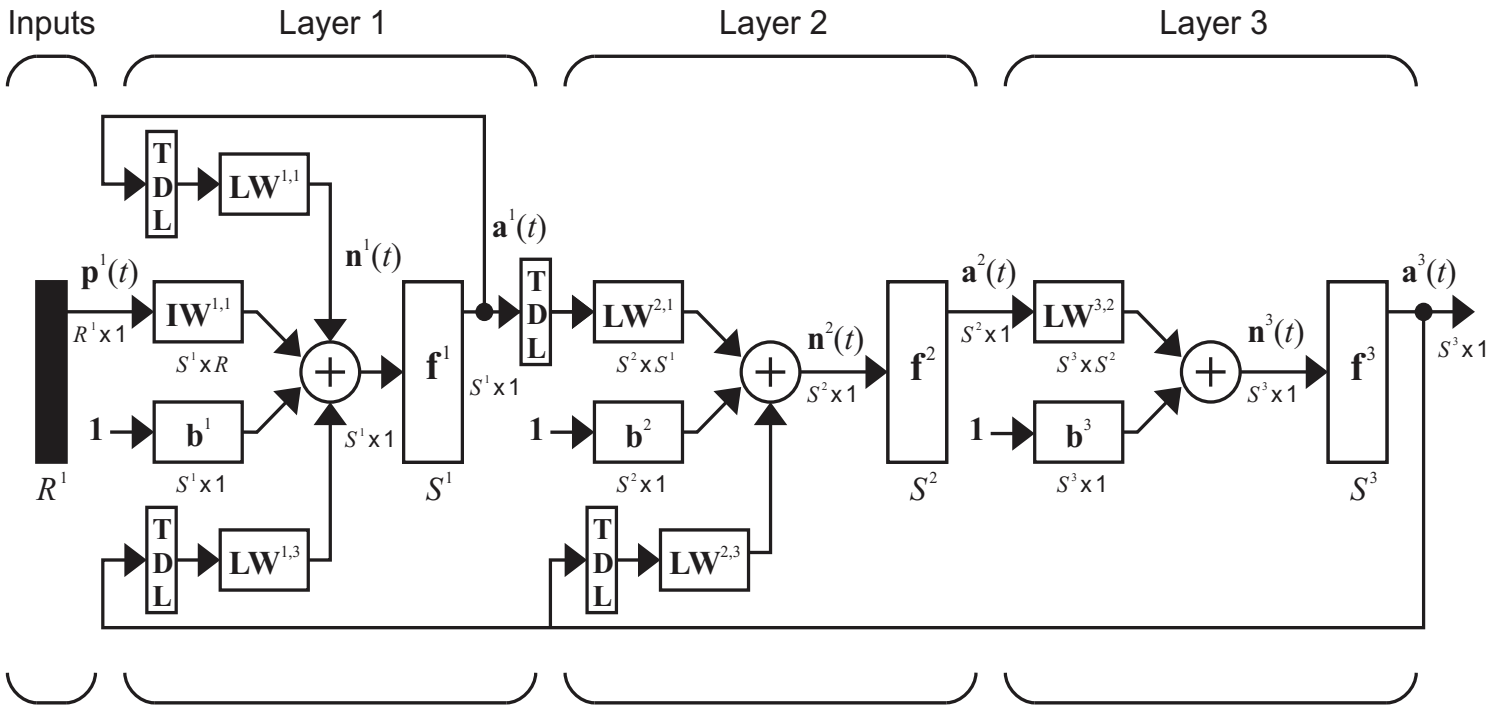


Adaptive Filter



$$a(k) = purelin(\mathbf{W}\mathbf{p} + b) = \sum_{i=1}^{R} w_{1,i}\, y(k - i + 1) + b$$

Layered Digital Dynamic Networks

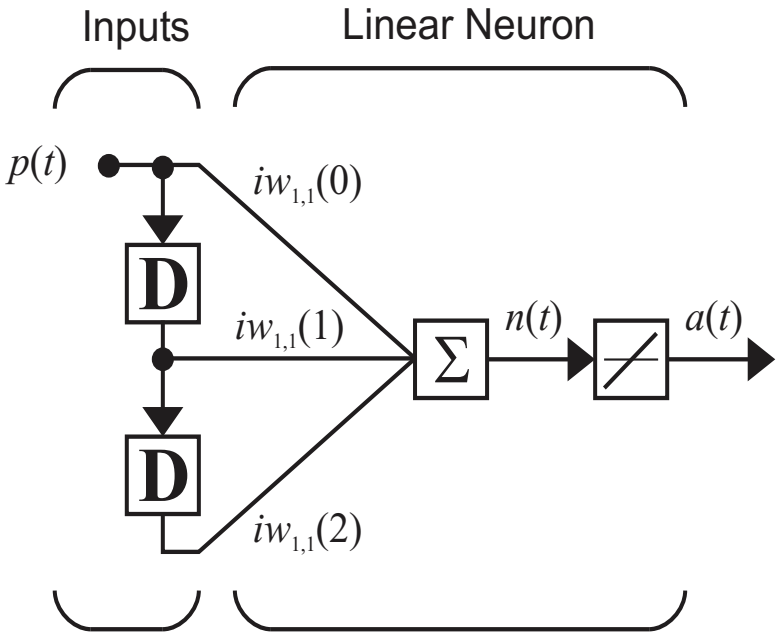


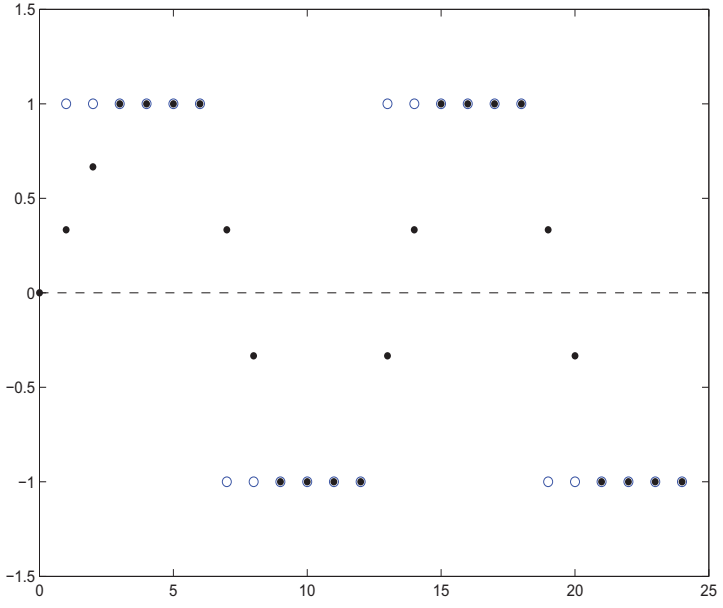
$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\,\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\,\mathbf{p}^1(t-d) + \mathbf{b}^m$$

$$\mathbf{a}^m(t) = \mathbf{f}^m(\mathbf{n}^m(t))$$

- A set of weight matrices that come into that layer (which may connect from other layers or from external inputs),
- Any tapped delay lines that appear at the input of a set of weight matrices,
- A bias vector,
- A summing junction,
- A transfer function.

Example Feedforward Dynamic Network

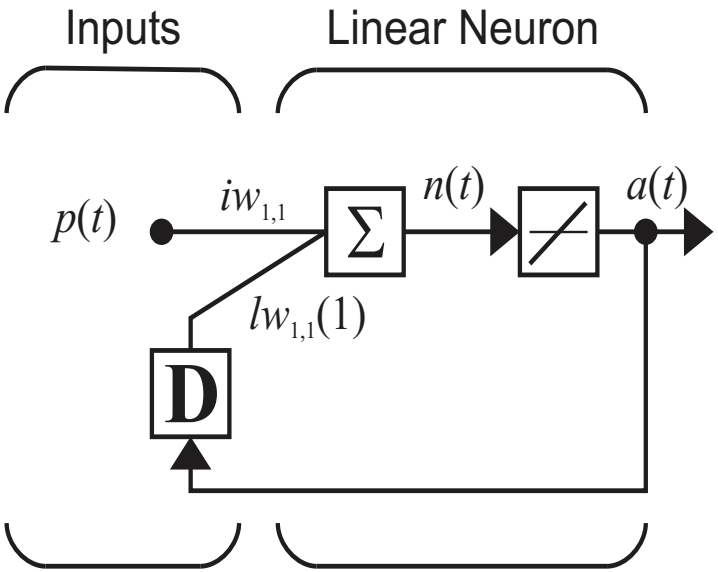




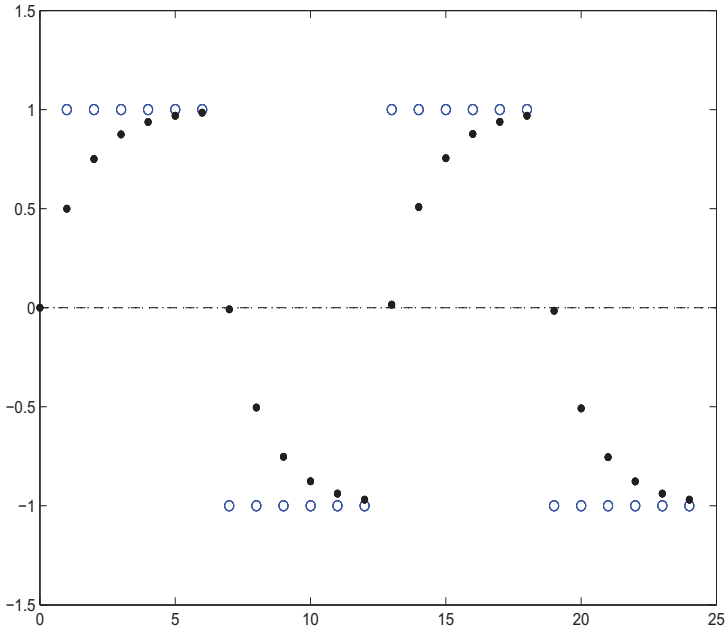
$$a(t) = iw_{1,1}(0)p(t)+iw_{1,1}(1)p(t-1)+iw_{1,1}(2)p(t-2)$$

$$iw_{1,1}(0) = \frac{1}{3} \qquad iw_{1,1}(1) = \frac{1}{3} \qquad iw_{1,1}(2) = \frac{1}{3}$$

Example Recurrent Dynamic Network



$$a(t) = iw_{1,1}p(t) + lw_{1,1}(1)a(t\text{-}1)$$

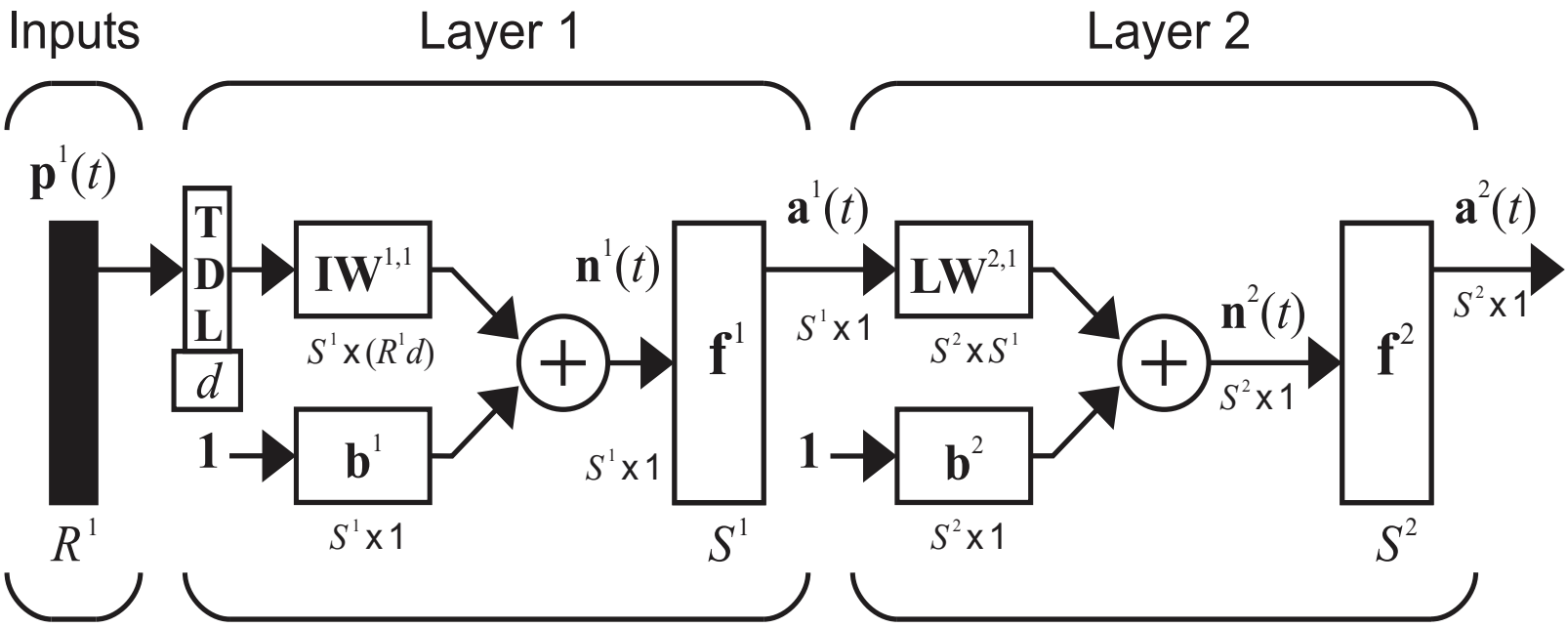


$$lw_{1,1}(1) = \frac{1}{2} \qquad iw_{1,1} = \frac{1}{2}$$

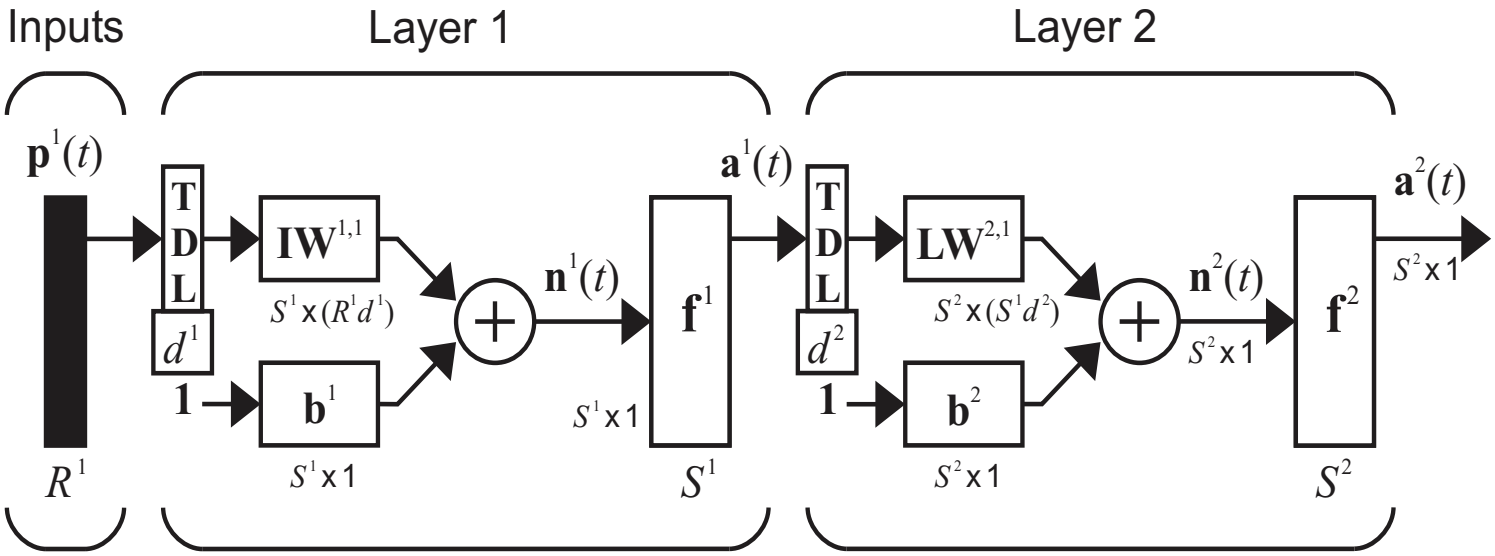
Applications of Dynamic Networks in NLP

- Natural Machine Translation
- Sequence to Sequence model
- Text Classification (Entity Recognition, Sentiment Analsis)
- Text Generation
- Question Answering
- Speech to Text

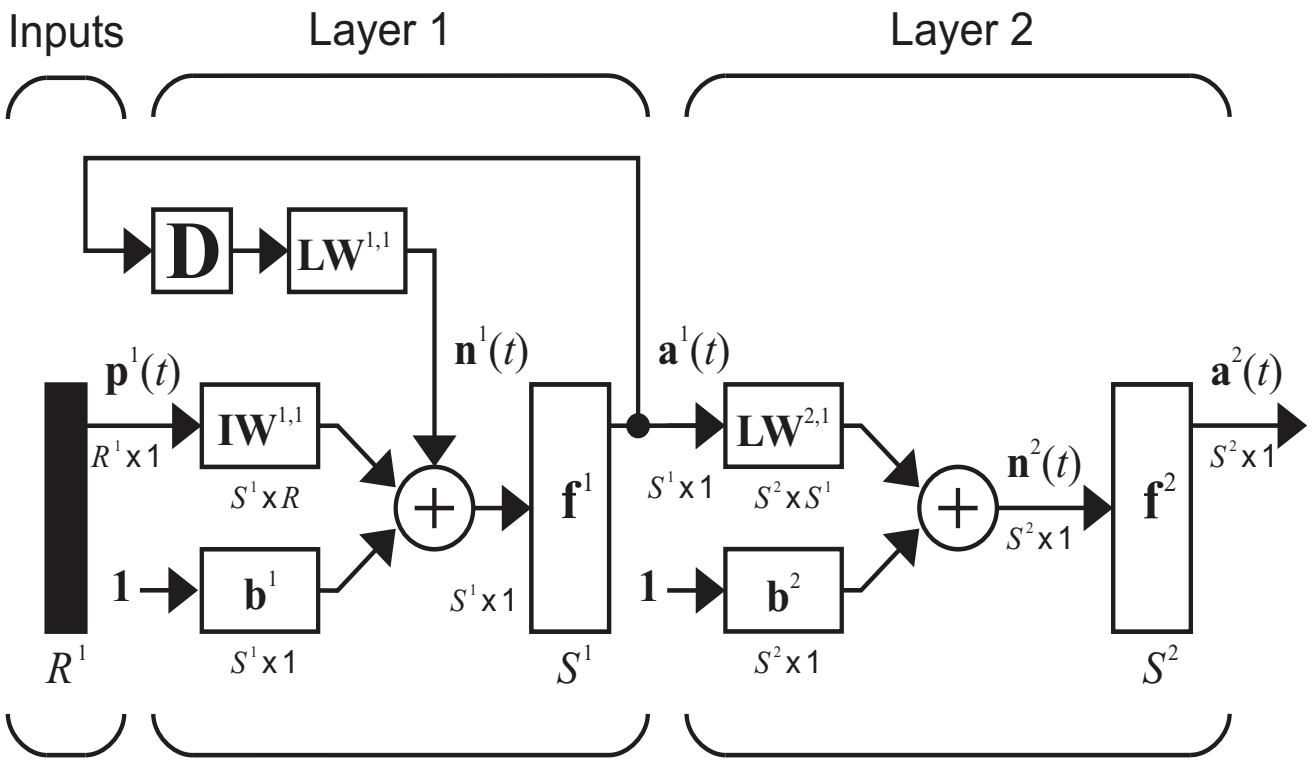
Focused Time Delay Network



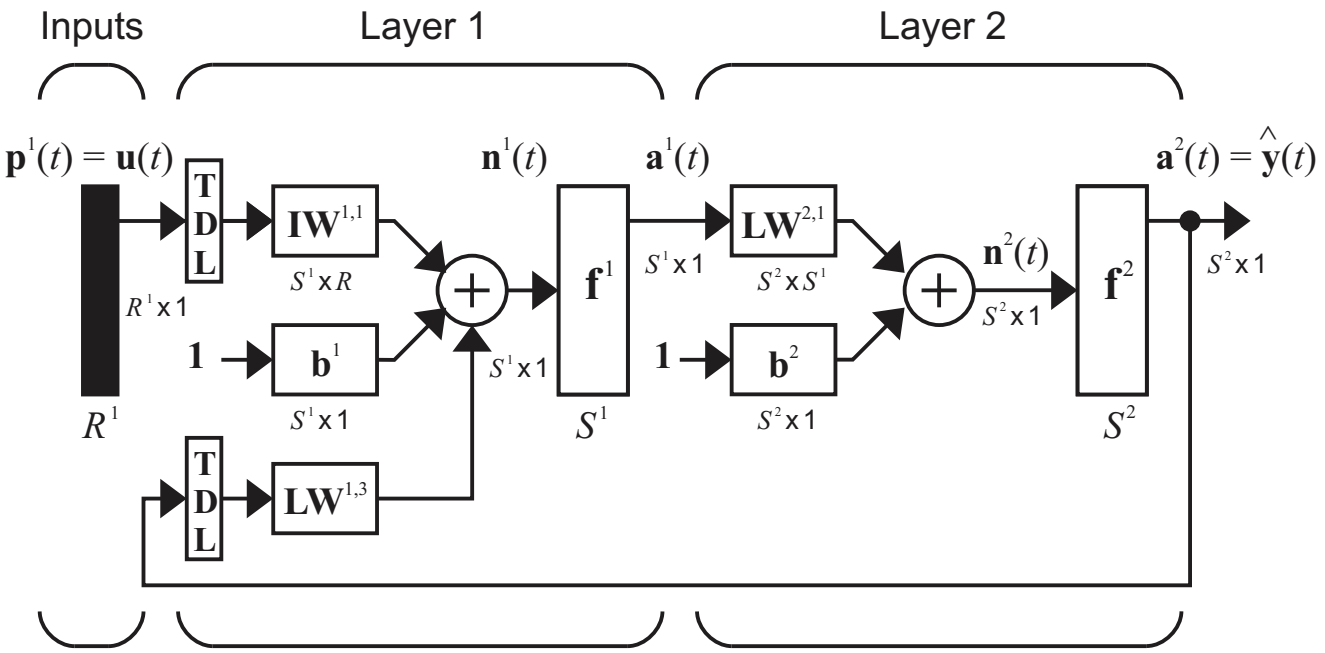
Distributed Time Delay Network



Layered Recurrent Network - Elman



$$y(t) = f(y(t-1), y(t-2), \ldots, y(t-n_y), u(t-1), u(t-2), \ldots, u(t-n_u))$$

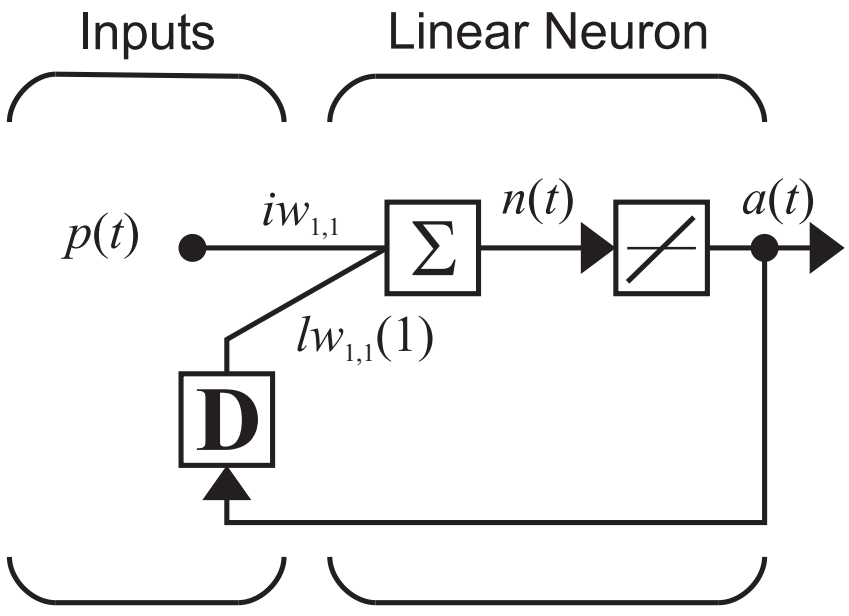


Long Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)

Learning long term dependencies

- For some recurrent network problems, we would like to predict responses that may be significantly delayed from the corresponding stimulus.
  - A chess game can be lost 12 moves after a mistake.
  - Words in a previous paragraph can provide context for a translation.
- A network structure must enable this possibility (have long term memory).
- Even within a structure that allows long term memory, the actual memory depends on the weights.
- Long memory systems can flirt with instability.

Simple recurrent network

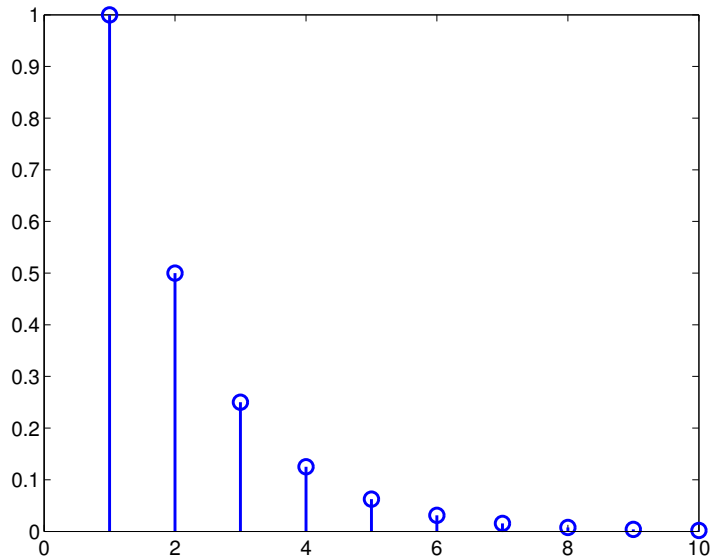


$$a(t) = iw_{1,1}p(t) + lw_{1,1}(1)a(t\text{-}1)$$

Network response

Let $lw_{1,1} = 0.5$ and $iw_{1,1} = 1$. Apply the input sequence of a one followed by 9 zeros, with zero initial condition.

$$a(t) = iw_{1,1}p(t) + lw_{1,1}a(t-1) = p(t) + 0.5a(t-1)$$
$$a(1) = p(1) = 1$$
$$a(2) = 0.5a(1) = 0.5$$
$$a(t) = 0.5^{(t-1)}$$



Gradient also decays quickly

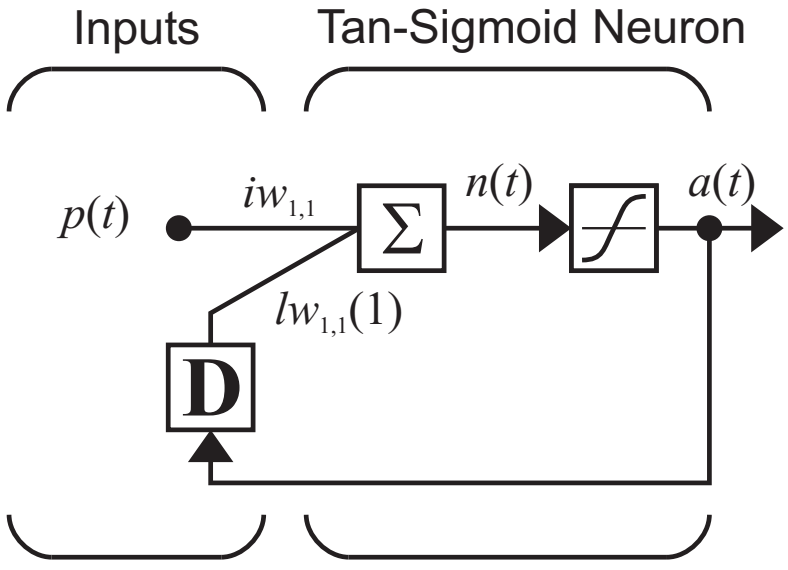
$$\frac{\partial F}{\partial a(t)} = \frac{\partial^e F}{\partial a(t)} + lw_{1,1} \frac{\partial F}{\partial a(t+1)}$$

If the error only occurs at the 10th time point, and $lw_{1,1} = 0.5$, we have

$$\frac{\partial F}{\partial a(t)} = 0.5^{(10-t)} \frac{\partial^e F}{\partial a(10)}$$

- If the network is stable, derivatives decay as you go back in time.
- Because the derivatives are small, it will take a long time to learn long term dependencies.

Nonlinear transfer function increases decay



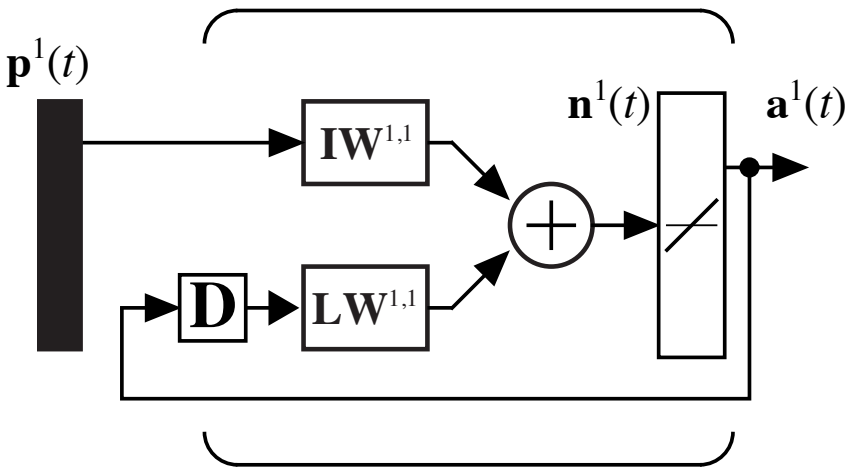
$$\frac{\partial F}{\partial a(t)} = \frac{\partial^e F}{\partial a(t)} + \dot{f}(n(t)) lw_{1,1} \frac{\partial F}{\partial a(t+1)}, \left|\dot{f}(n(t))\right| \leq 1$$

Difficulties in learning long term dependencies

- Long term memories are network weights – short term memories are layer outputs.
- We need a network with long short term memory
- In recurrent networks, as the weights change during training, the length of the short term memory will change.
- If the initial weights produce a network without long short term memory, it will be difficult to increase it.
- This is because the gradient will be small for short short term memory networks.
- If the initial weights produce long short term memory, instabilities can easily occur.

Constant error carousel (CEC)

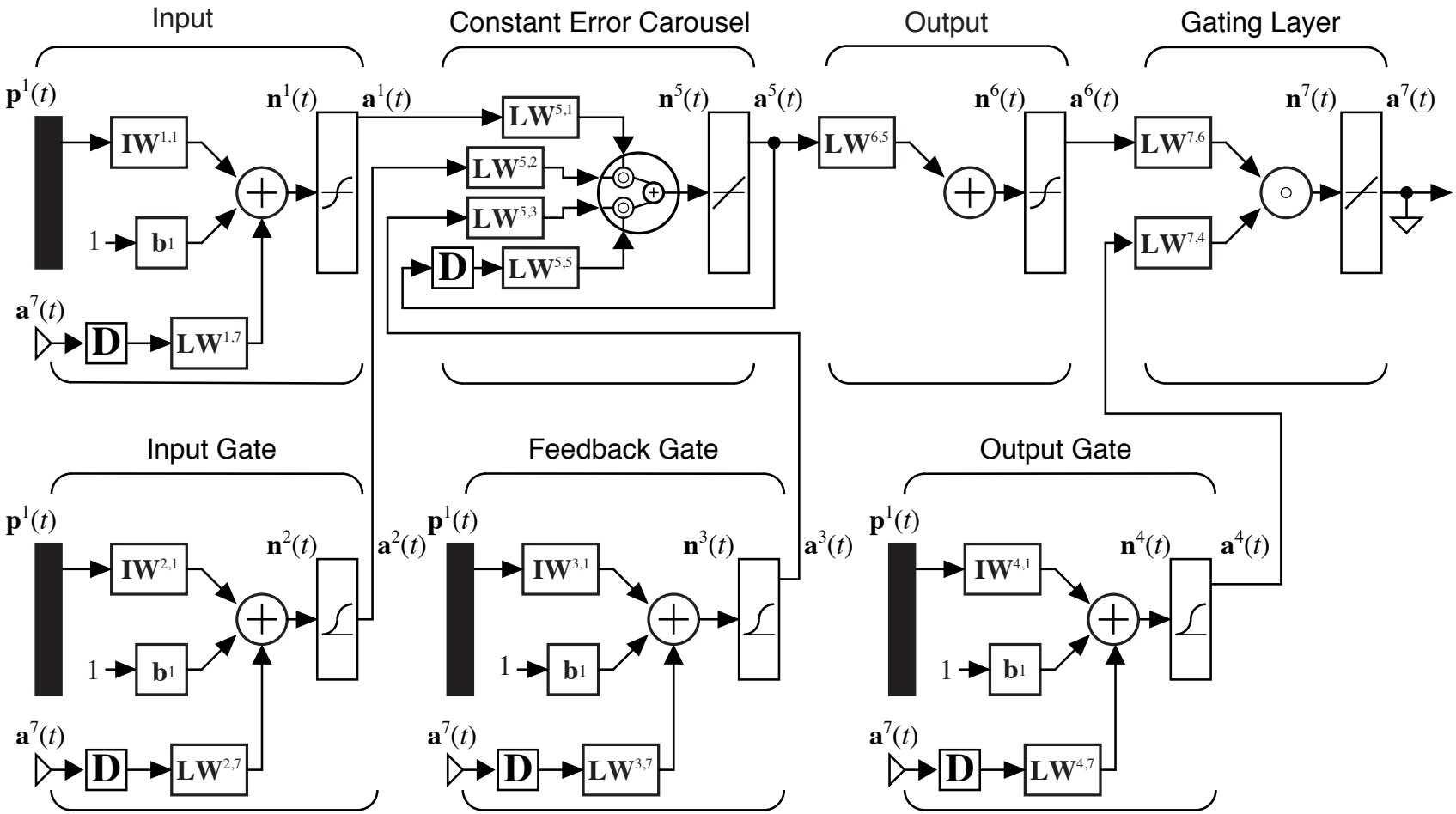
- To maintain a long memory, we would like the feedback matrix $\mathbf{LW}^{1,1}$ to have some eigenvalues very close to one.
- This has to be maintained during training, or the gradients will vanish.
- In addition, to ensure long memories, the derivative of the transfer function should be constant.
- Eigenvalues greater than one $\rightarrow$ unstable.
- Solution: Set $\mathbf{LW}^{1,1} = \mathbf{I}$ and use linear transfer function.



Gating

- We don't want to indiscriminately remember everything.
- To remember selectively, we introduce several gates (switches).
  - An input gate will allow selective inputs into the CEC.
  - A feedback (or forget) gate will clear the CEC.
  - An output gate will allow selective outputs from the CEC.
- Each gate will be a layer with inputs from the gated ouputs and the network inputs.
- The result is called Long Short Term Memory (LSTM).
- With the CEC, short term memories last longer.

Long Short Term Memory Network



LSTM Details

- The $\circ$ operator is the Hadamard product, which is an element by element matrix multiplication.
- The weights in the CEC are all fixed to the identity matrix. They are not trained.
- The output and gating layer weights are also fixed to the identity matrix.
- It has been shown that the best results are obtained when initializing the feedback (forget) gate bias, $\mathbf{b}^3$, to all ones or larger values. This turns the gate on initially.
- Other weights and biases are set to small random numbers.
- The output of the gating layer generally connects to another layer or a multilayer network with softmax transfer function.
- Multiple LSTMs can be cascaded together.

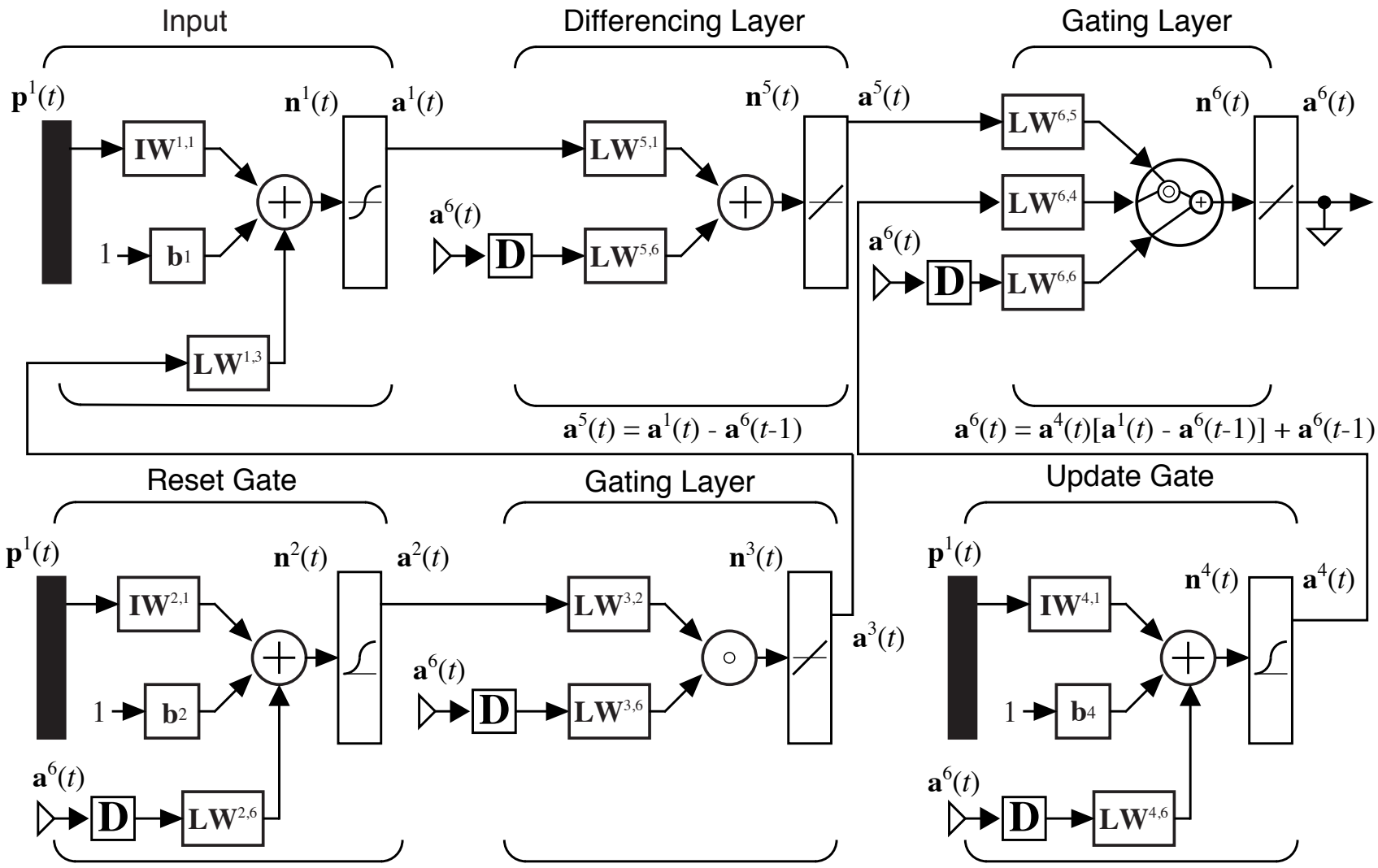
LSTM Training

- LSTM is trained with the standard gradient-based algorithms as with other deep networks.
- The original LSTM paper used RTRL for computing the gradient.
- An approximate gradient is used, in which the derivatives are only propagated in time through the CEC delays. Only static derivatives are computed for the remaining terms.

Variations of the LSTM

- There have been many variations of the LSTM
- In the original LSTM, there was no feedback (forget) gate.
- Some variations feeback the CEC output $\mathbf{a}^5(t)$ to the gate layers. These are called "peephole" connections.
- See http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf for an experimental comparison of various alternatives.
- One of the popular alternatives is the Gated Recurrent Unit (GRU), shown on the following page.

GRU Network


Input
Differencing Layer
Gating Layer
Reset Gate
Gating Layer
Update Gate
$\mathbf{p}^1(t)$
$\mathbf{n}^1(t)$
$\mathbf{a}^1(t)$
$\mathbf{n}^5(t)$
$\mathbf{a}^5(t)$
$\mathbf{n}^6(t)$
$\mathbf{a}^6(t)$
$\mathbf{n}^2(t)$
$\mathbf{a}^2(t)$
$\mathbf{n}^3(t)$
$\mathbf{a}^3(t)$
$\mathbf{n}^4(t)$
$\mathbf{a}^4(t)$
IW1,1
b1
LW1,3
LW5,1
LW5,6
LW6,5
LW6,4
LW6,6
IW2,1
b2
LW2,6
LW3,2
LW3,6
IW4,1
b4
LW4,6
D
1
$\mathbf{a}^5(t) = \mathbf{a}^1(t) - \mathbf{a}^6(t\text{-}1)$
$\mathbf{a}^6(t) = \mathbf{a}^4(t)[\mathbf{a}^1(t) - \mathbf{a}^6(t\text{-}1)] + \mathbf{a}^6(t\text{-}1)$

- The differencing layer subtracts the delayed network output $\mathbf{a}^6(t-1)$ from the processed input $\mathbf{a}^1(t)$.
- $\mathbf{LW}^{5,1} = \mathbf{I}$ and $\mathbf{LW}^{5,6} = -\mathbf{I}$. They are not trained.
- The gating layer weights are fixed to the identity matrix.
- The delayed network output is gated by the reset gate, before it is fed into the input layer. This resets the memory.
- The overall output can be considered a weighted average of a candidate potential output $\mathbf{a}^1(t)$ and the previous output $\mathbf{a}^6(t-1)$.

$$\begin{aligned}\mathbf{a}^6(t) &= \mathbf{a}^4(t) \times [\mathbf{a}^1(t) - \mathbf{a}^6(t-1)] + \mathbf{a}^6(t-1) \\ &= \mathbf{a}^4(t) \times \mathbf{a}^1(t) + [1 - \mathbf{a}^4(t)] \times \mathbf{a}^6(t-1)\end{aligned}$$

Click here to open colah blog.

Pads and Pack Variable Length sequences in Pytorch

Textual Data Length

- There are many kinds of sequence data but here we will look at text data.
- It is most commonly used in natural language processing.
- The length of the text data is different. The sequence data has a variable length.
- Now, let's look at an example of the sequence data.
- Here, you can see 3 text sequence data composed of word units. There are different sizes in one batch.

first	second	third
first	second	
first		

- If you want to pass these sequences to some recurrent neural network architecture then you have to $< pad >$all of the sequences (typically with 0s) in our batch to the maximum sequence length is 3.

first	second	third
first	second	<pad>
first	<pad>	<pad>

Problem With Padding

- For the sake of understanding, let's also assume that we will matrix multiply the above-padded batch of sequences of shape $(3, 3)$ with a weight matrix W.

- Thus, we will have to perform $3 \times 3 = 9$ multiplication and $3 \times 2 = 6$ addition (nrows x $(n-1)$ cols) operations, only to throw away most of the computed results since they would be 0s (where we have pads).

Packed Sequences

- Read the data and encode them

```
1  import torch
2  from torch import nn
3  from torch.nn.utils.rnn import pad_sequence,pad_packed_sequence,pack_padded_sequence
4
5  docs = ['first second third',
6          'first second',
7          'first']
8
9  word_set=set()
10 for seq in docs:
11   for word in seq.split(" "):
12     word_set.add(word)
13
14 word_list=['<pad>']+list(word_set)
15 print(word_list)
16
17 word2idx={word: idx for idx,word in enumerate(word_list)}
18 vocab_size=len(word_list)
19 embedding_dim=10
```

Pad Sequences and Creating a Pack Sequence

- Feed packed sequence in RNN

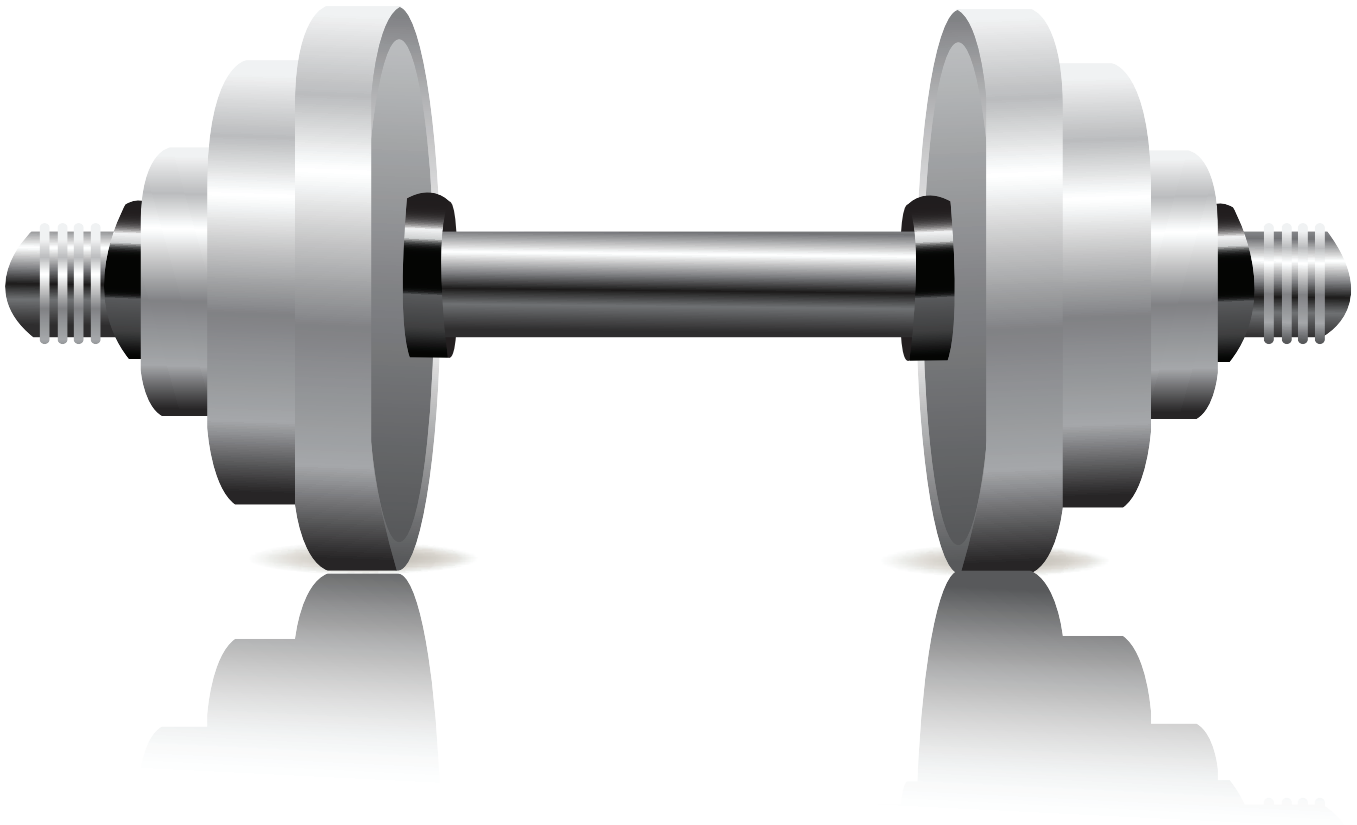
```
x = [torch.LongTensor([word2idx[word] for word in seq.split(" ")]) for seq in docs]
x_padded = pad_sequence(x, batch_first=True, padding_value=0)

print(x_padded)
seq_len=torch.LongTensor(list(map(len,x)))
print(seq_len)

embed=nn.Embedding(vocab_size,embedding_dim)
lstm=nn.LSTM(embedding_dim,hidden_size=5,batch_first=True)

embedding_seq_tensor=embed(x_padded)
print(embedding_seq_tensor)

packed_input = pack_padded_sequence(embedding_seq_tensor, seq_len.cpu().numpy(),
                                    batch_first=True,enforce_sorted=False)
print(packed_input.data.shape)
packed_output,(ht,ct)=lstm(packed_input)

print(packed_output.data.shape)
output, input_sizes = pad_packed_sequence(packed_output, batch_first=True)
print(ht[-1])
```

More on Pad Pack Sequence

- Lets run the following code ***2-More_on_Pad_Pack.py***

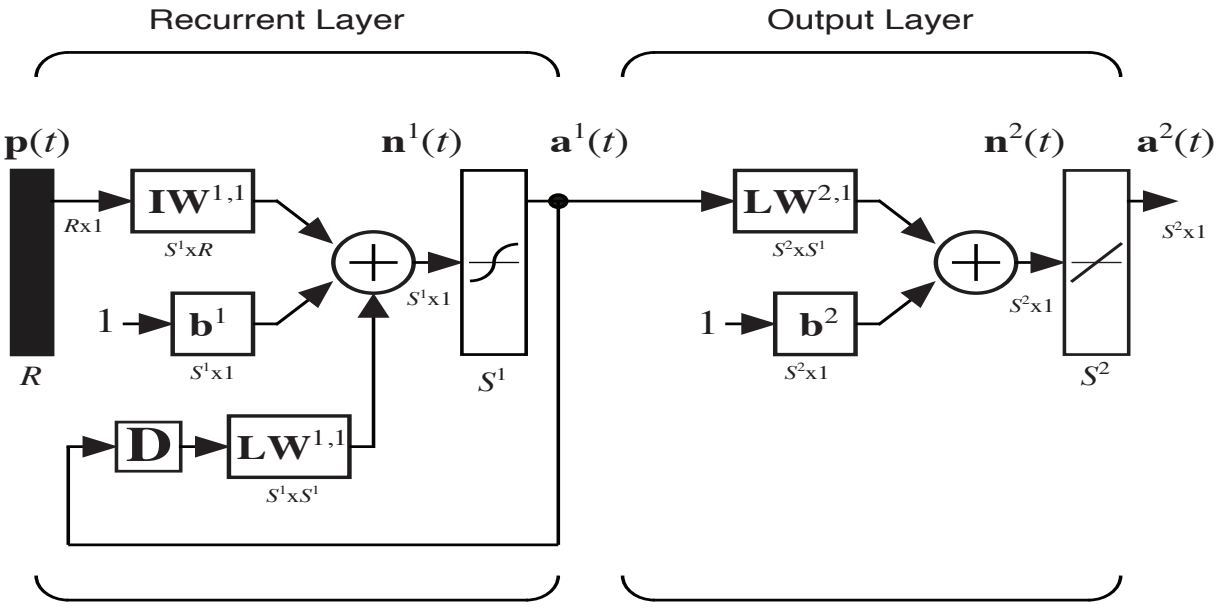
Exercise - Lecture 8

- Exercise 1 to 2
- Class-Ex-Lecture8.py



Sequence 2 Sequence Model (seq2seq)

- A recurrent network is a network that has feedback connections.



$$\mathbf{a}^1(t) = \mathbf{tansig}\left(\mathbf{IW}^{1,1}\mathbf{p}(t) + \mathbf{LW}^{1,1}\mathbf{a}^1(t-1) + \mathbf{b}^1\right)$$

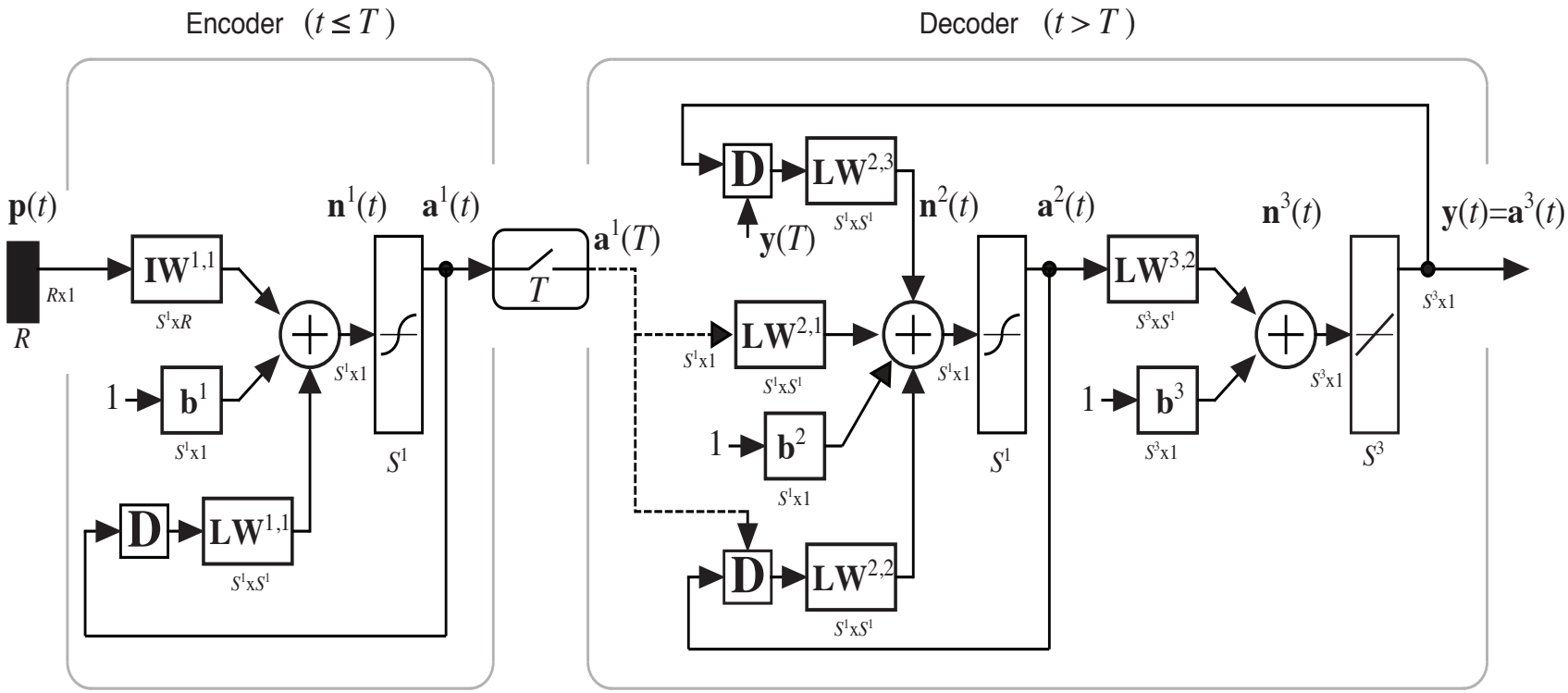
$$\mathbf{a}^2(t) = \mathbf{LW}^{2,1}\mathbf{a}^1(t) + \mathbf{b}^2$$

Seq2seq Model

- An interesting network that uses two RNNs as components is the sequence-to-sequence network (seq2seq).
- The first RNN is called the Encoder, and the second RNN is called the decoder.
- This network has been used for such tasks as machine translation.
- For example, the $u(t)$ input sequence might represent English words and the output sequence 4y(t)4 might represent French words.

Seq2seq Model

- seq2seq model with simple recurrent network layer.



Encoder

- The encoder takes the input sequence, which has T time points (words, or word fragments), and produces a layer output $a^1(t)$.
- This output is sampled at the final time point $t = T$.
- The dotted line at the output of the sampler indicates that this value is fixed, and does not change with time.
- For example, the $u(t)$ input sequence might represent English words and the output sequence 4y(t)4 might represent French words.

$$a^1(t) = tansig(IW^{1,1}p(t) + LW^{1,1}a^1(t-1) + b^1)$$

Decoder

- The decoder initializes its state with the final state of the encoder.
- The final state of the encoder also becomes a constant input to the first layer of the decoder.
- Hint: (There are many different formulations of the seq2seq model. In some cases the final encoder state is only used to initialize the state of the decoder.)
- This final encoder state is sometimes referred to as the **context**.

$$a^2(t) = tansig(LW^{2,3}a^3(t-1) + LW^{2,1}a^1(T) + LW^{2,2}a^2(t-1) + b^1)$$
$$a^3(t) = LW^{3,2}a^1 + b^3$$

Attention

- The recurrent layer of the encoder is able to store information from the beginning of the input sequence, but it may not be able to store it in the most efficient way.

- To provide more flexibility, we can add a tapped delay line of previous values of the encoder state, rather than using just the final value. This process is called attention.

- The first step is to use a TDL_hstack to form a horizontal concatenation of previous encoder states, as in

$$A^1(t) = \begin{bmatrix} a11(t) & a^1(t-1) & ..... & a1(t-D+1) \end{bmatrix}$$

- We want to combine these vectors together in some way to form the context vector to send to the decoder (in place of just the final encoder state).
- The question is how much should each previous encoder state contribute to the context.
- This could be done by finding the inner product between the previous decoder state and all previous encoder states, as in the following equation

$$n^4(t) = \left[A^1(T)\right]^T a^2(t-1)$$

- We should then normalize these correlations by using the softmax activation function, as in

$$a^4(t) = softmax(n^4(t))$$

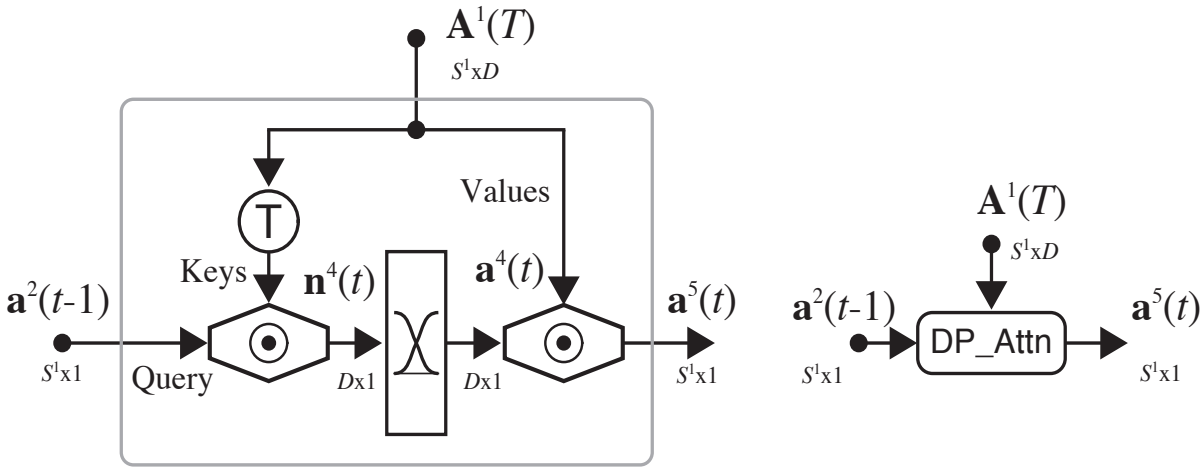
- Finally, we combine the previous encoder state vectors according to the relative amount of correlation with the current decoder state.

$$a^5(t) = A^1(T)a^4(t)$$

- The resulting $a^5(t)$ is the context vector that will be passed to the decoder.

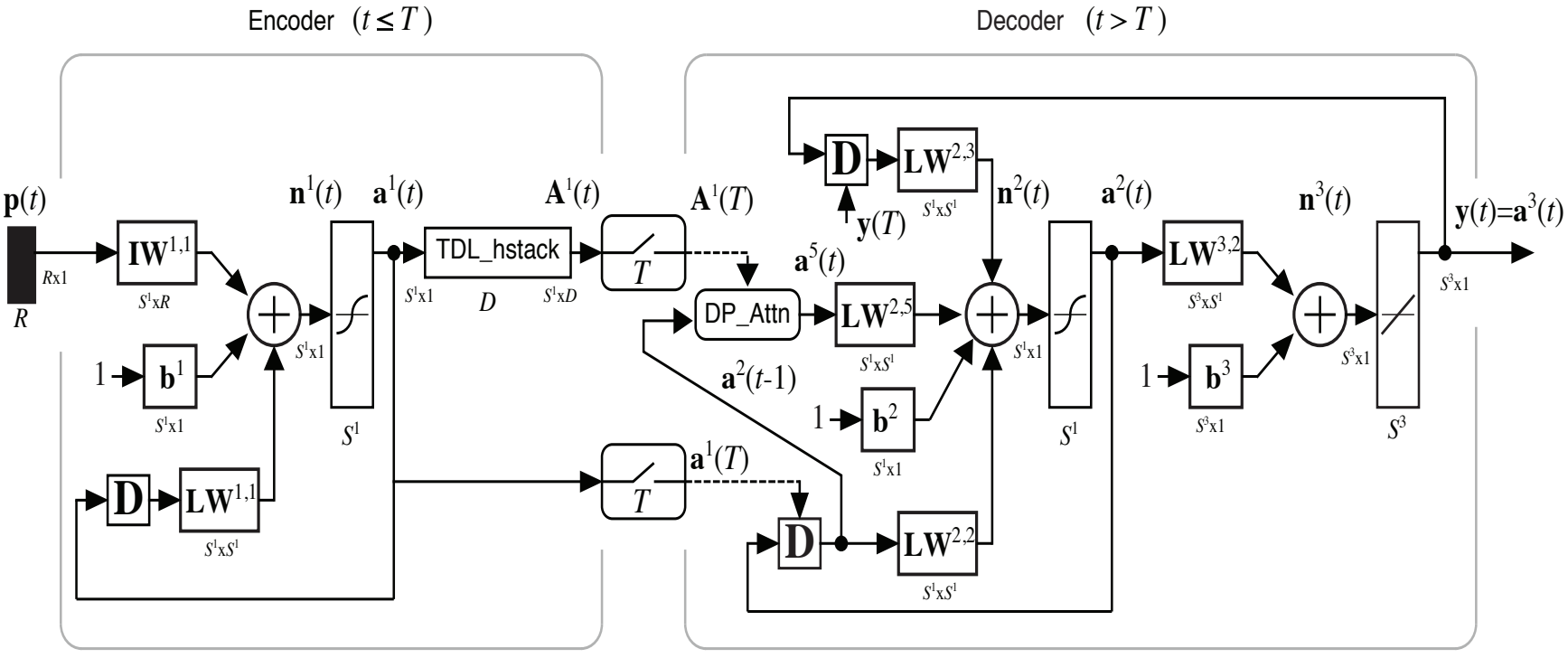
Attention Model

- The idea of attention is sometimes generalized as operations between a query vector and sets of key and value pairs.
- The query is compared to each of the keys to determine how much of each corresponding value is included in the result.



- In our case, the query is the previous decoder state $a^2(t-1)$, the keys are all previous encoder states, and the values are also the previous encoder states.

seq2seq with Attention Model



Click here to open seq2seq model with attention