LAB 4: RECURRENT NEURAL NETWORKS (RNNs)

University of Washington, Seattle

Spring 2025

OUTLINE

Part 1: Introduction to RNNs

- Why do we need RNNs?
- RNN Architecture
- RNN in PyTorch
- Embedding and Decoder

Part 2: Training RNNs

- Backpropagation in RNNs
- Vanishing/Exploding Gradient Problem
- Training with Teacher Forcing

Part 3: Gated RNNs

- LSTM and GRU

Part 4: RNN Problem Types

- RNN Configurations

Part 5: RNN Implementation in PyTorch

- Character Level Generation Shakespeare Dataset

Lab Assignment

- Create Arthur Conan Doyle AI

INTRODUCTION TO RNNs

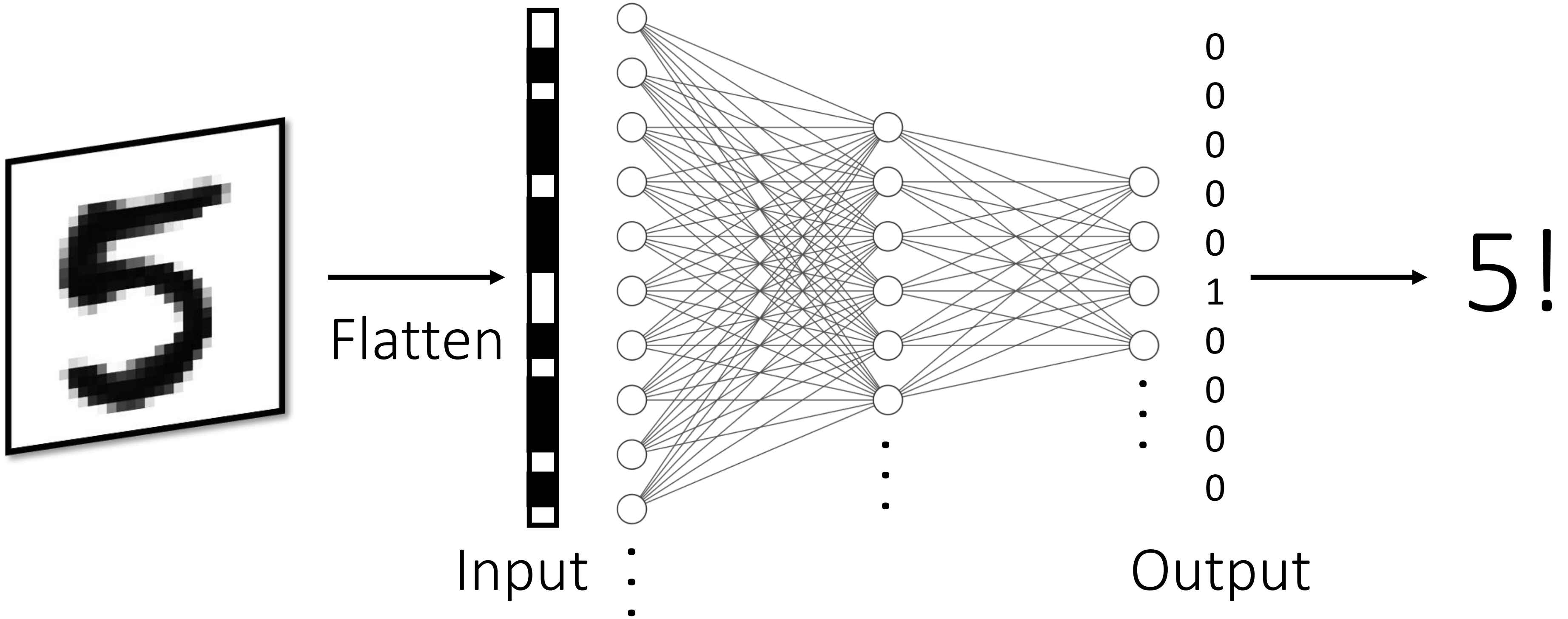
Why do we need RNNs?

RNN Architecture

RNN in PyTorch

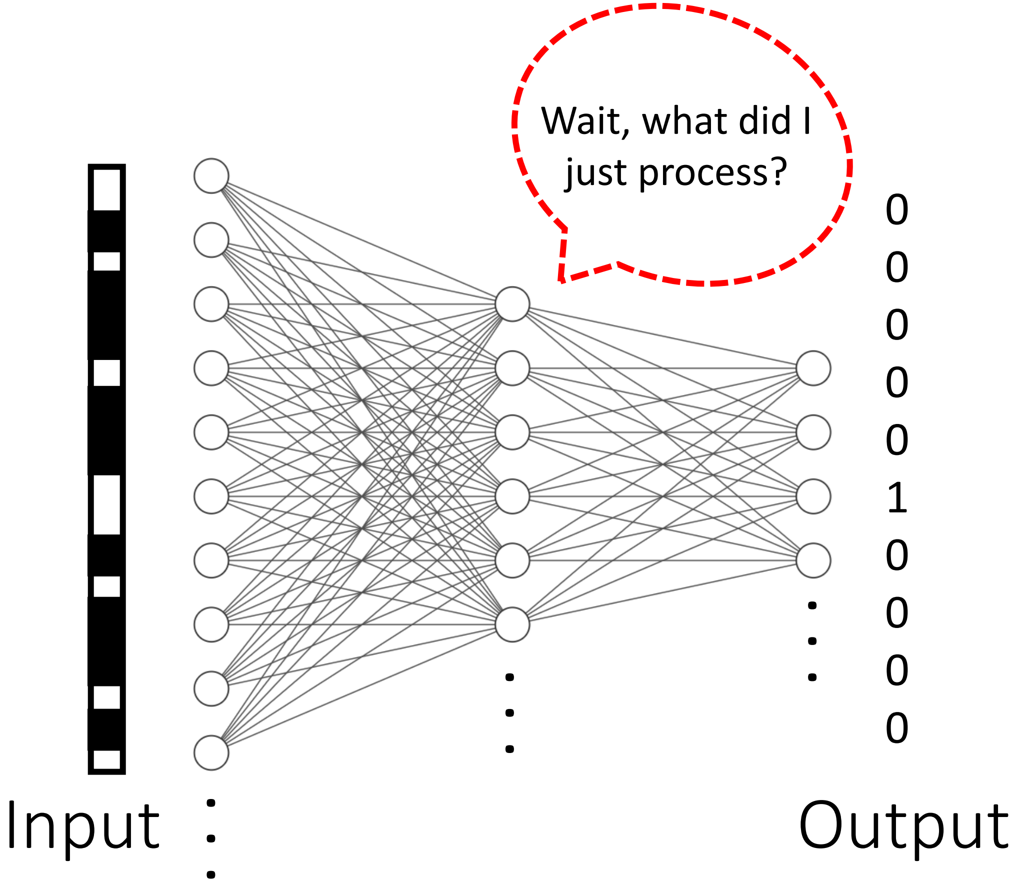
Embedding and Decoder

Why Do We Need RNNs?



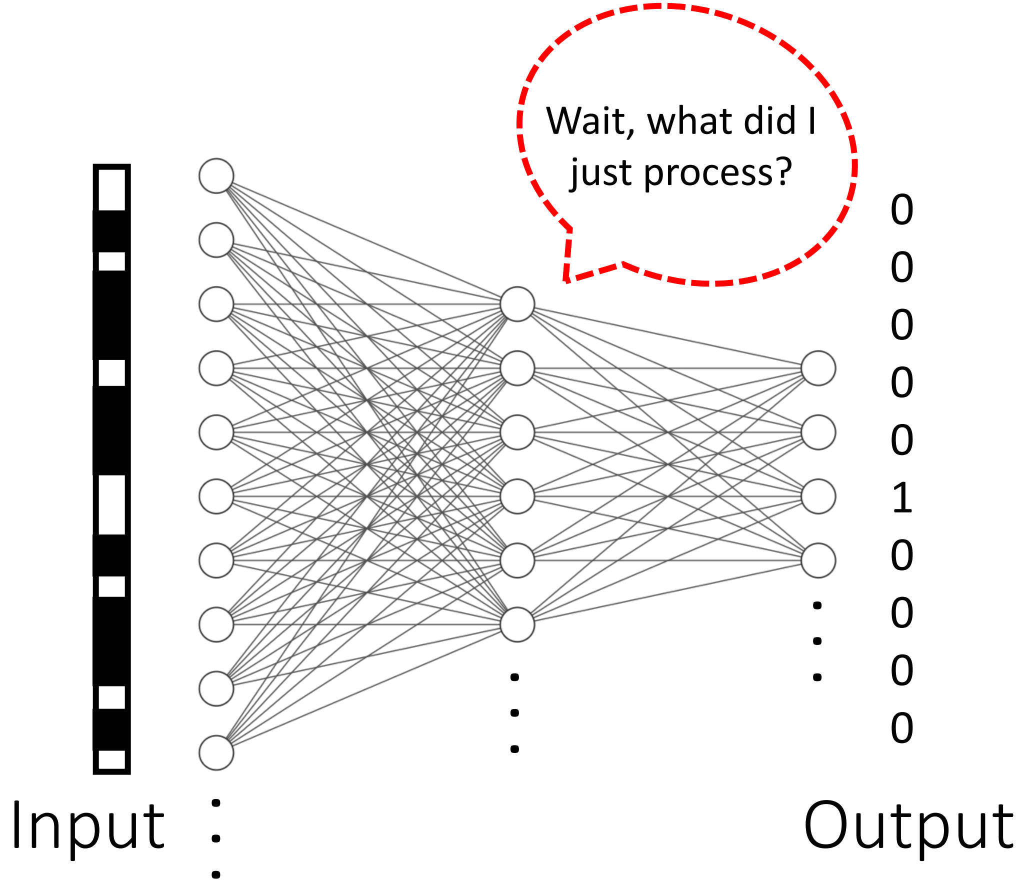
Feed-Forward Network

Why Do We Need RNNs?



Feed-Forward Network

Why Do We Need RNNs?



Feed-Forward Network has no memory of past inputs

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름은 지민이에요

English Hello, my name is Jimin

Why Do We Need RNNs?

Korean 안녕하세요,

English Hello,

Why Do We Need RNNs?

Korean 안녕하세요, 제

English Hello, my

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름

English Hello, my name

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름은

English Hello, my name is

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름은 지민이에요

English Hello, my name is Jimin

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름은 지민이에요

English Hello, my name is Jimin

Each word in a sentence is dependent to the past words →
Need memory

Why Do We Need RNNs?

Korean 안녕하세요, 제 이름은 지민이에요, 그리고 저는 비디오게임을 좋아해요

English Hello, my name is Jimin, and I like videogames

A sentence (input) could have **different sizes**

Why Do We Need RNNs?

We need a neural network architecture that can handle:

Why Do We Need RNNs?

We need a neural network architecture that can handle:

- Data order

Why Do We Need RNNs?

We need a neural network architecture that can handle:

- Data order
- Temporal dependencies

Why Do We Need RNNs?

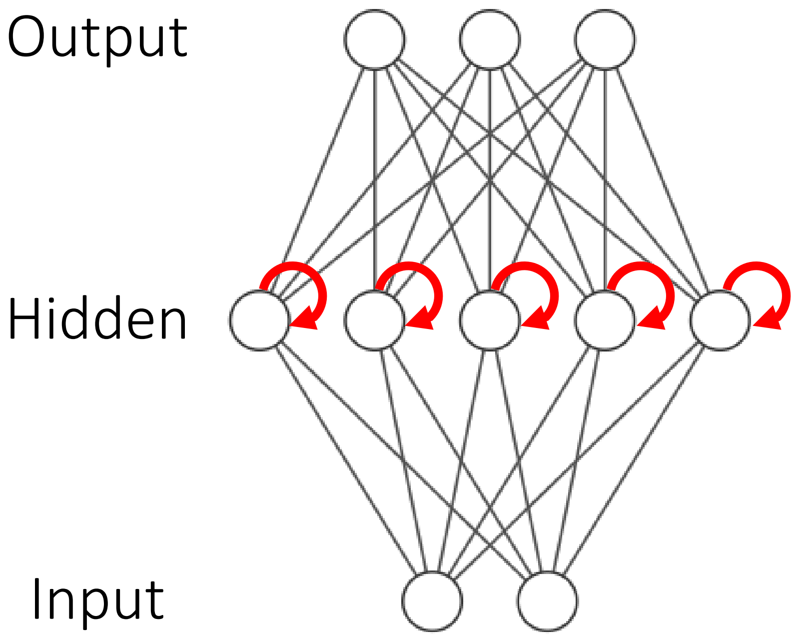
We need a neural network architecture that can handle:

- Data order
- Temporal dependencies
- Variable input sizes

Why Do We Need RNNs?

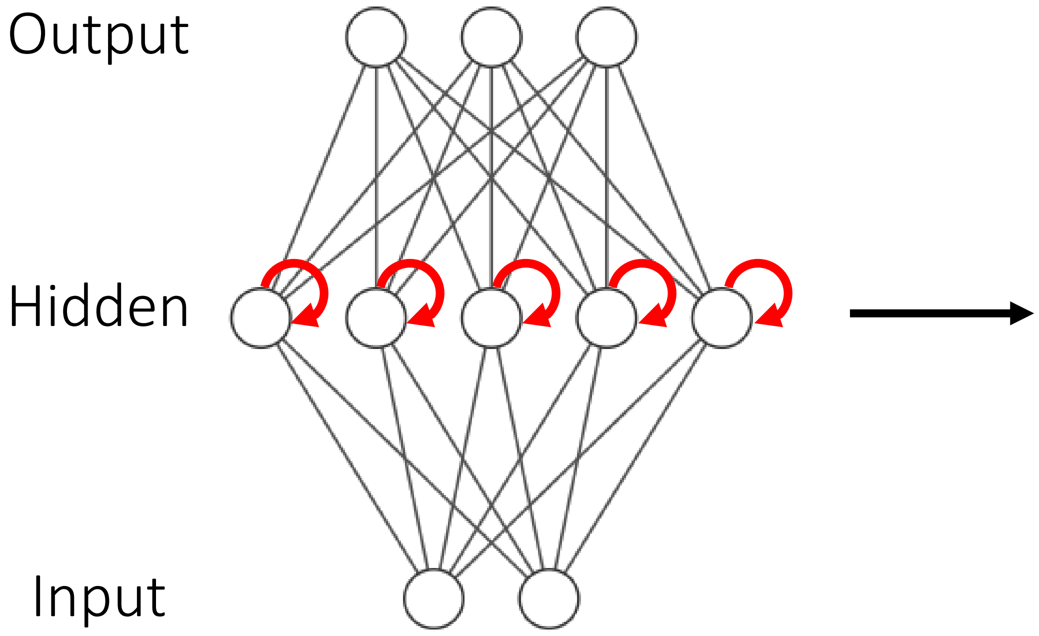
We need a neural network architecture that can handle:

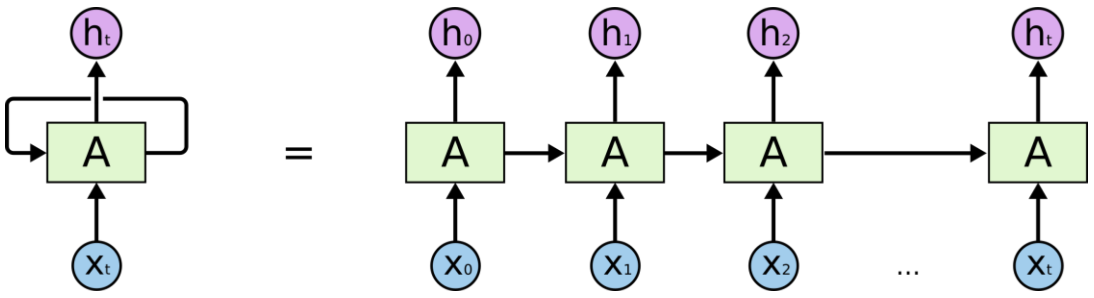
- Data order
- Temporal dependencies
- Variable input sizes



Recurrent Neural Network

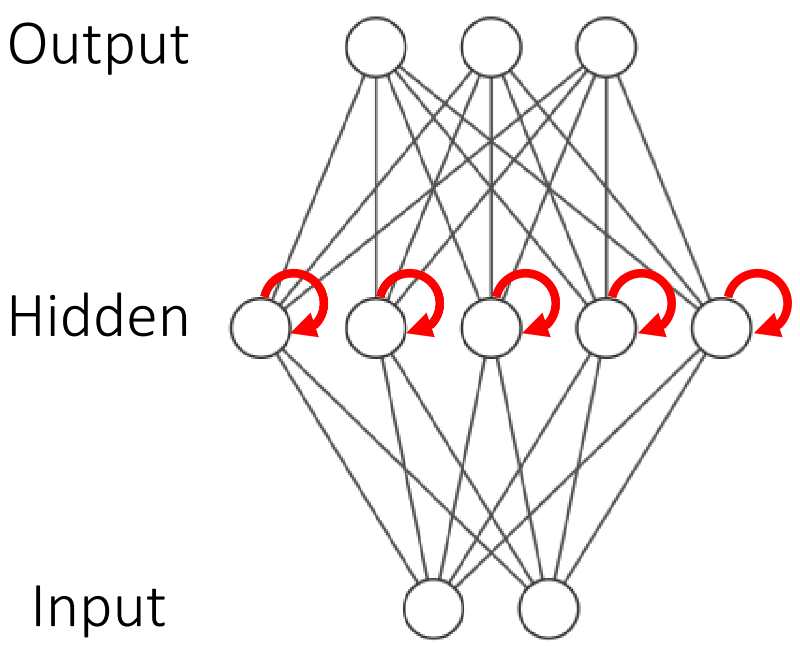
RNN Architecture

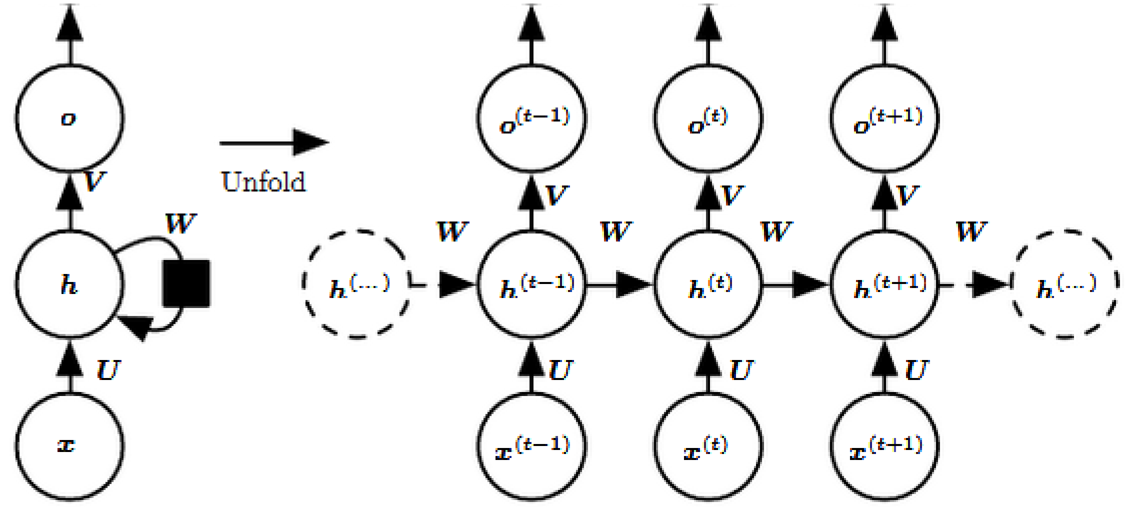




Unfold in Time

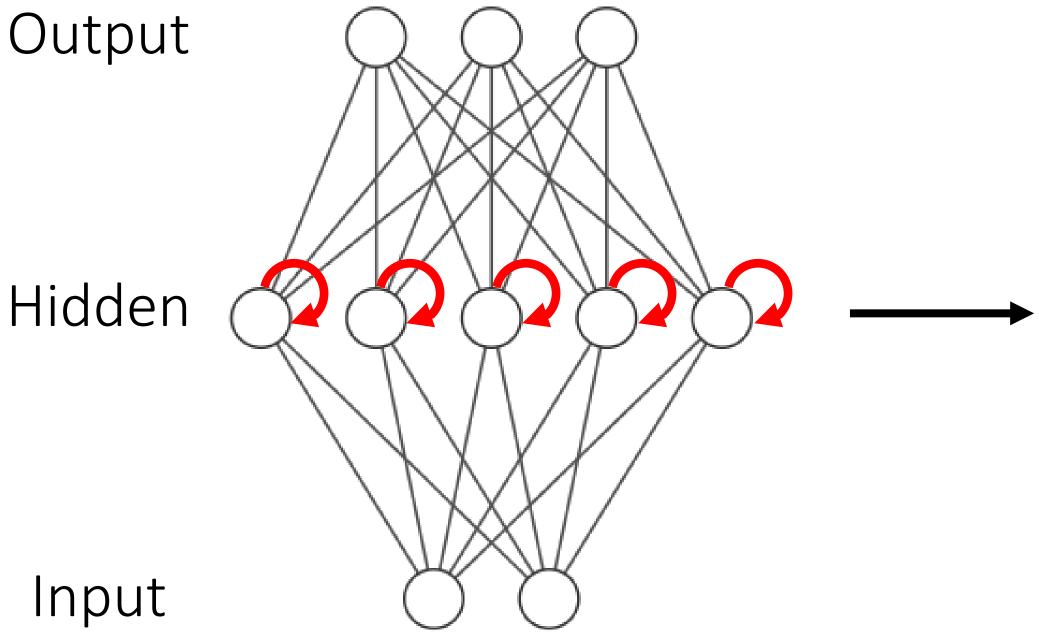
RNN Architecture

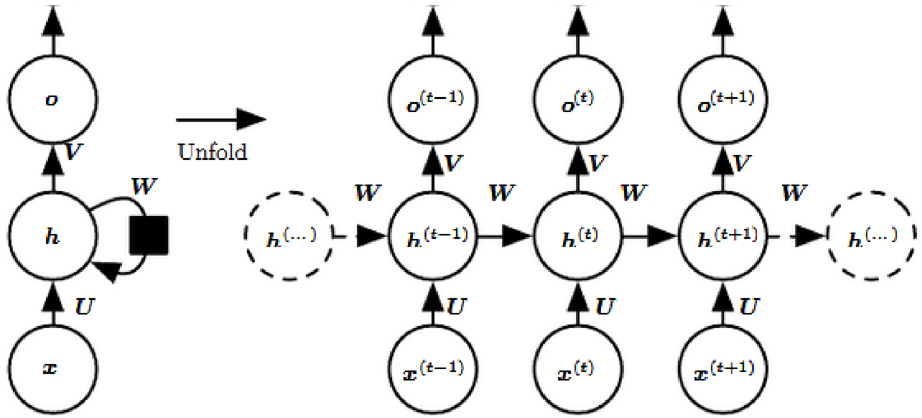




Unfold in Time

RNN Architecture

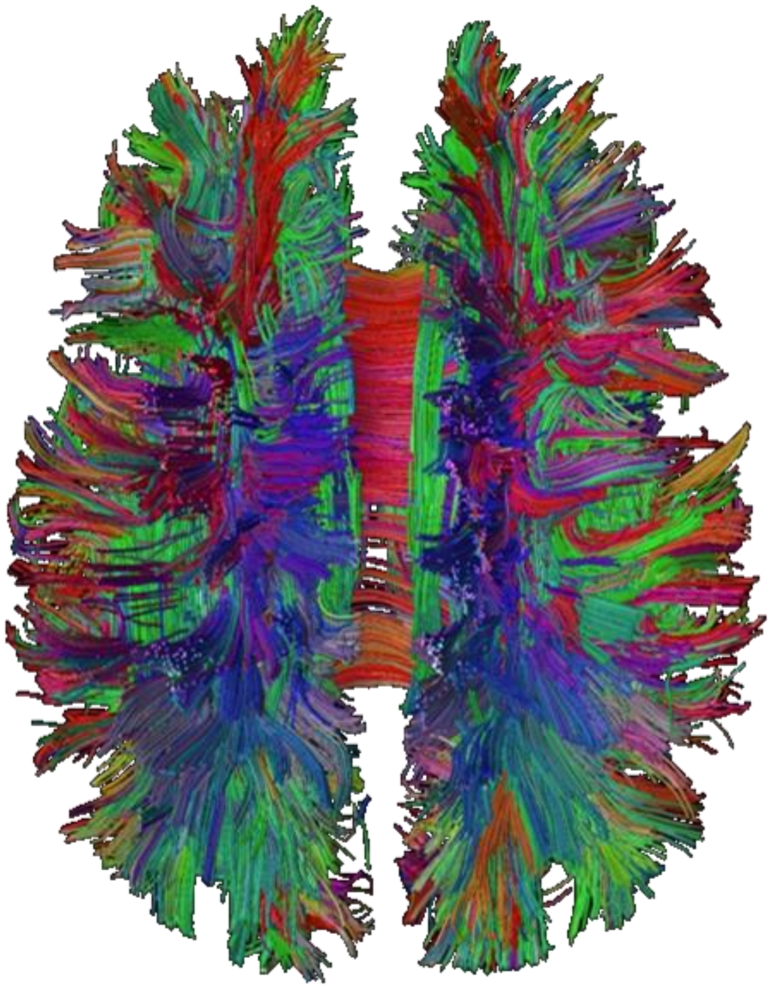




Unfold in Time

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Brain is Highly Recurrent

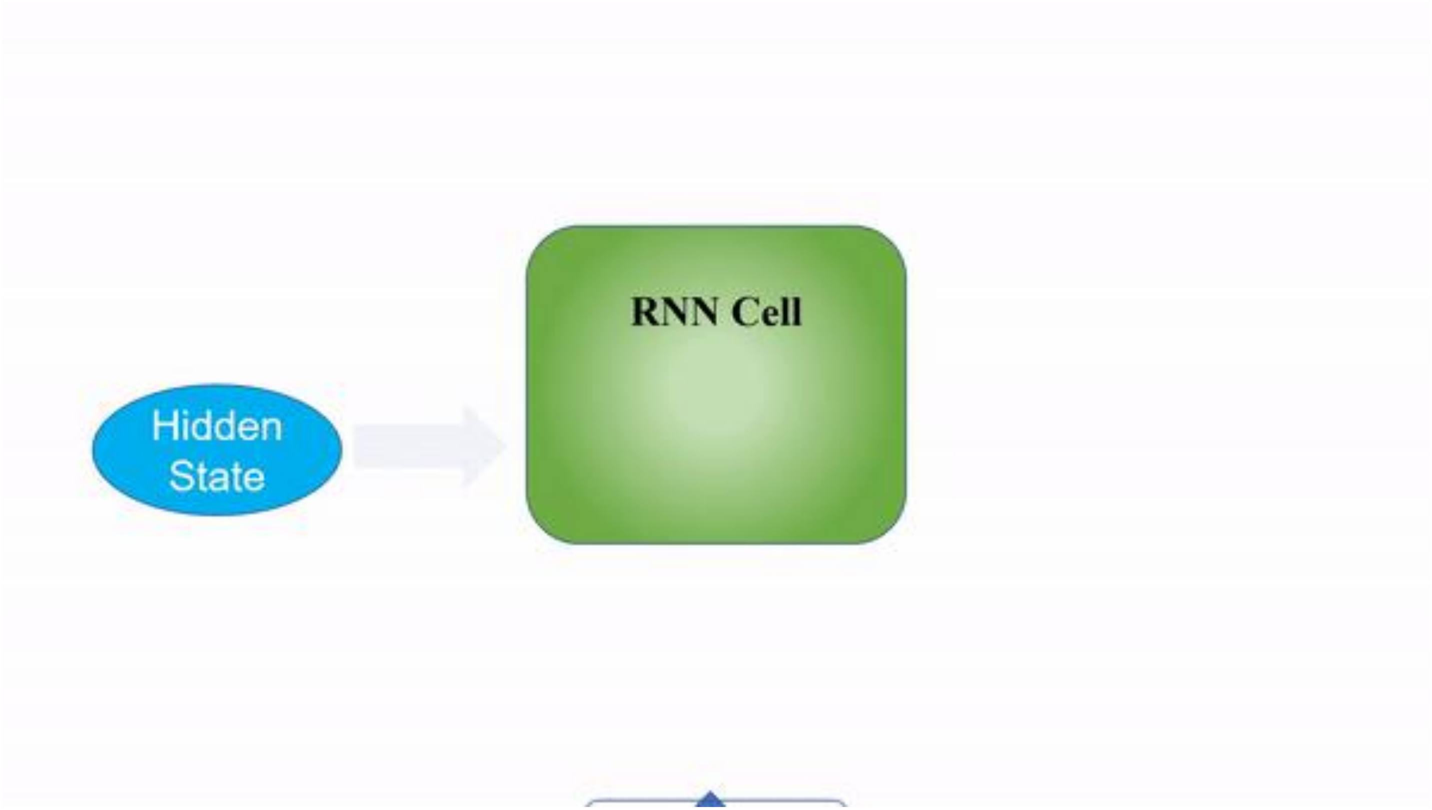


Neurons themselves have continuous voltage dynamics

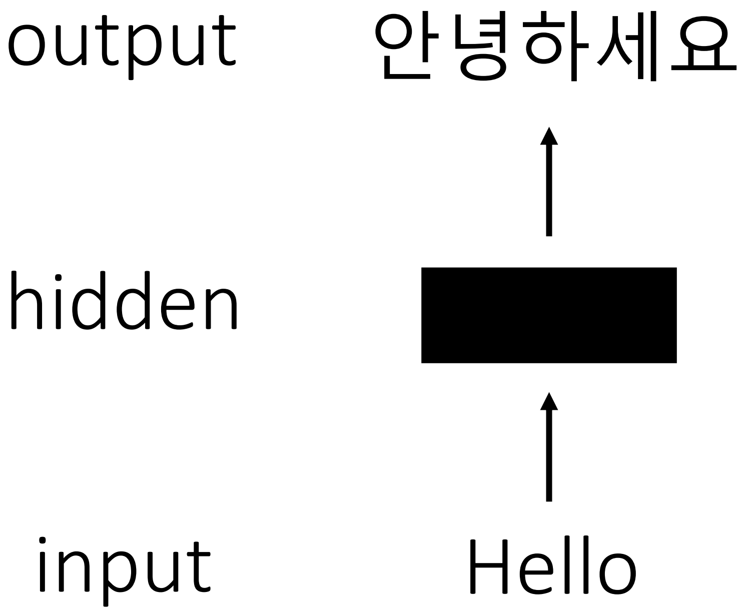
Different parts of brain exchange information both forward and backward

Brain is Highly Recurrent

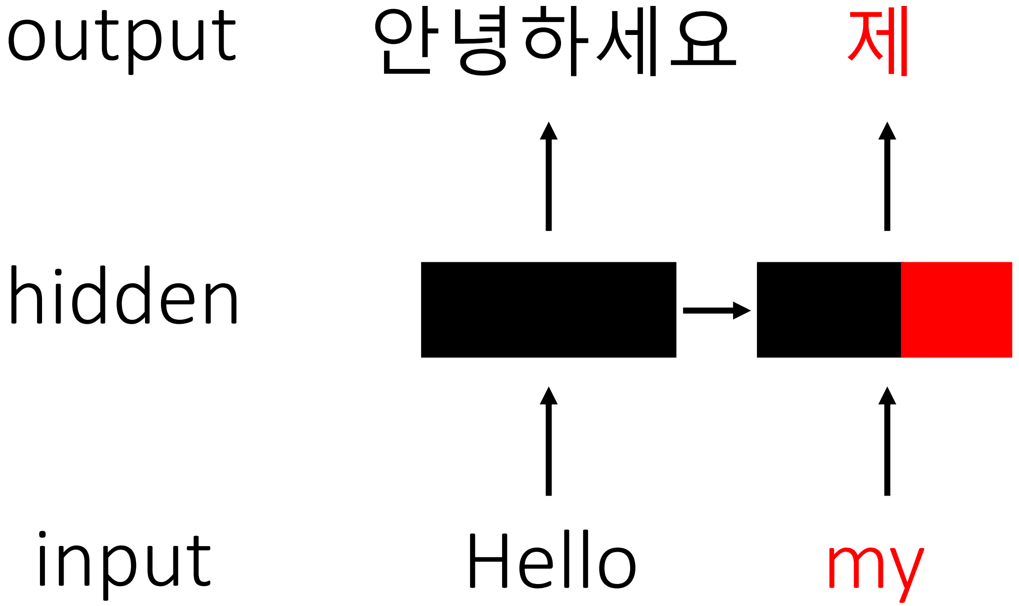
RNN Architecture



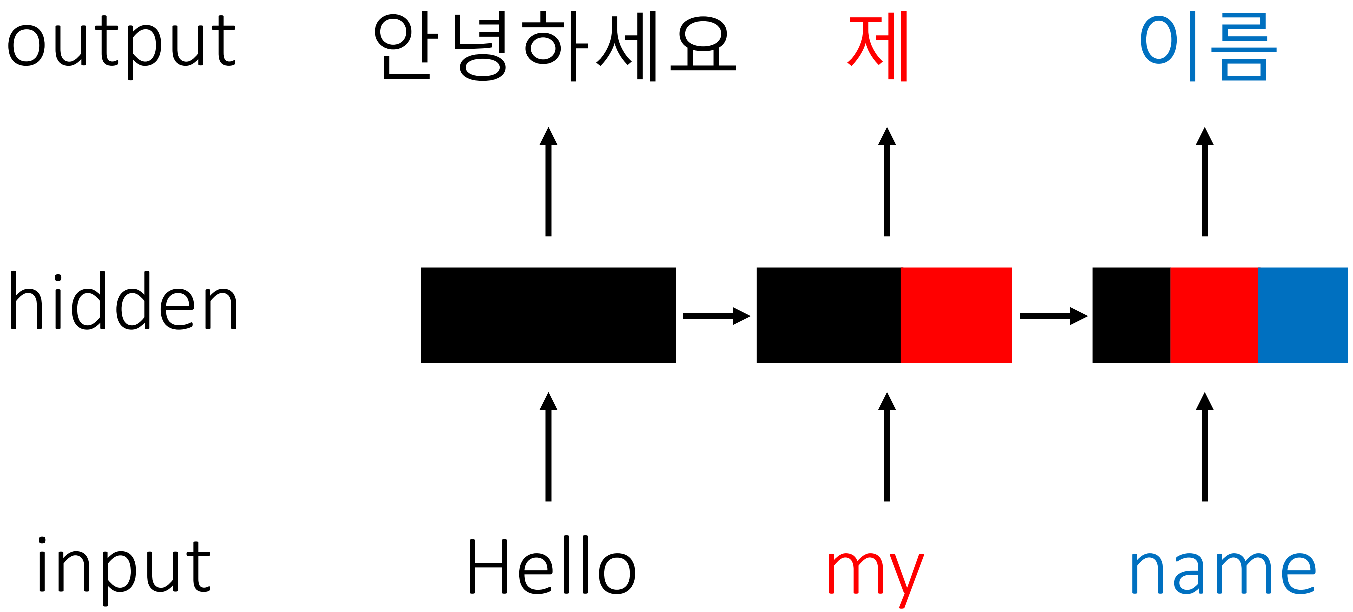
RNN Architecture



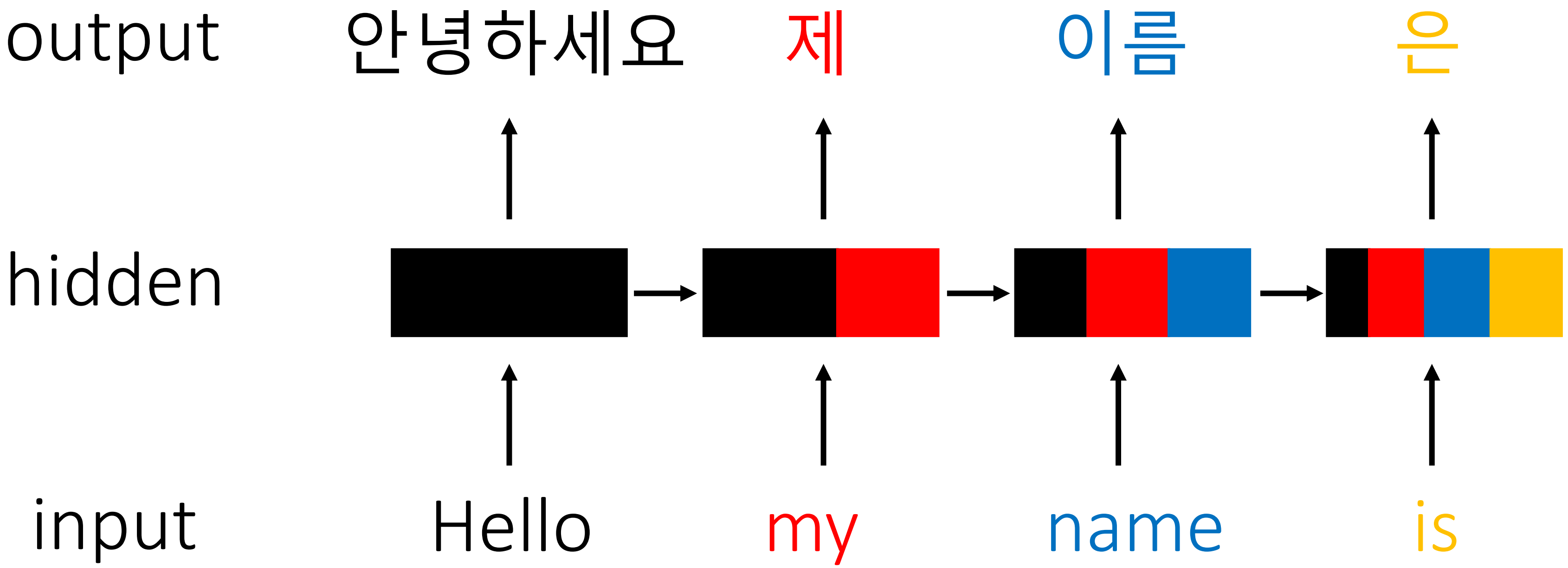
RNN Architecture



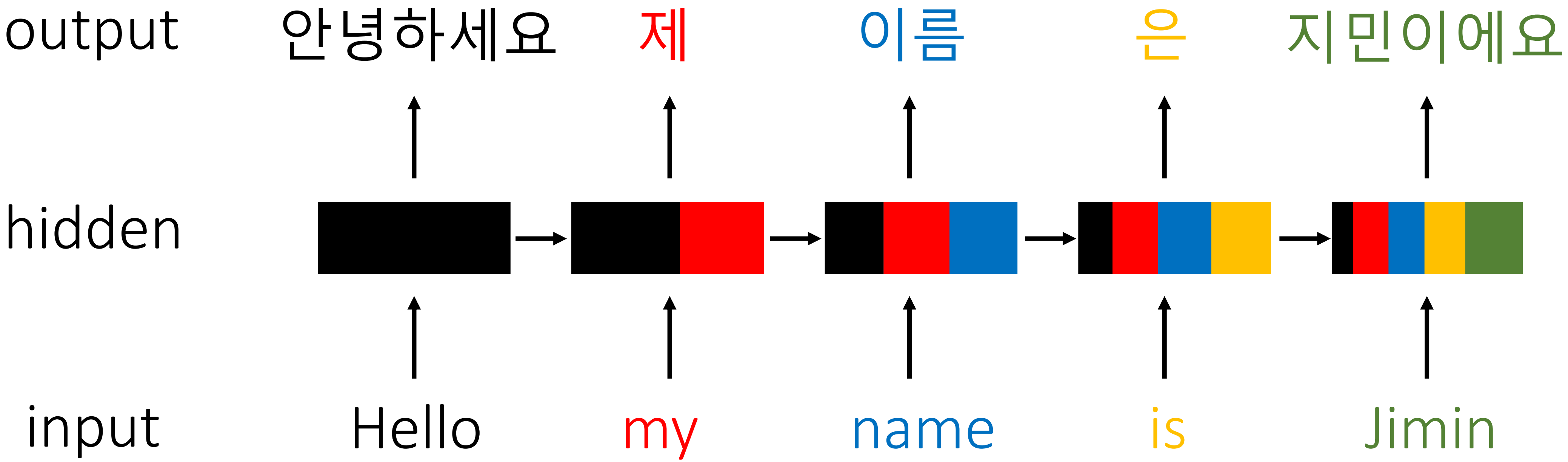
RNN Architecture



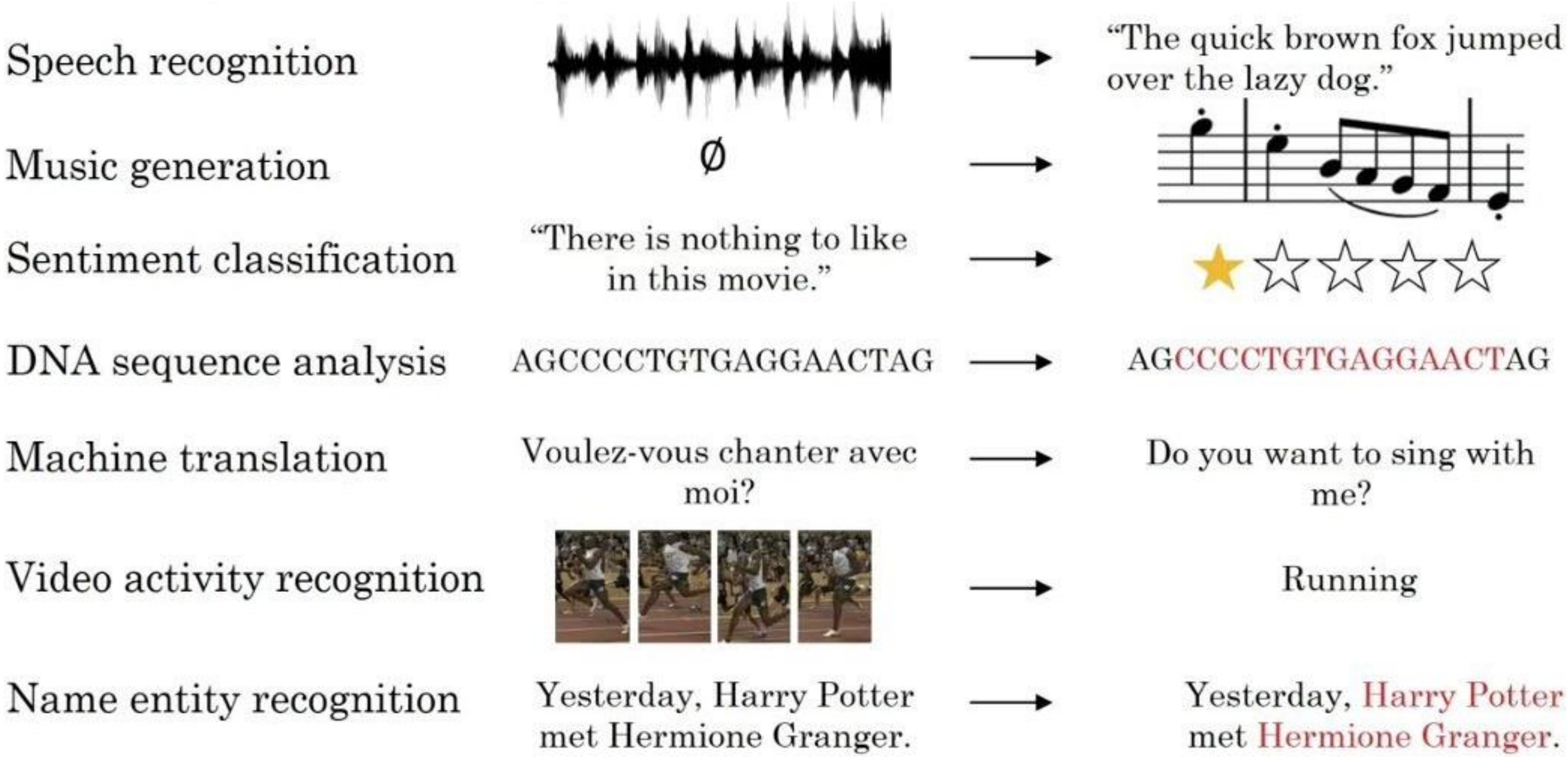
RNN Architecture



RNN Architecture



Sequential Data



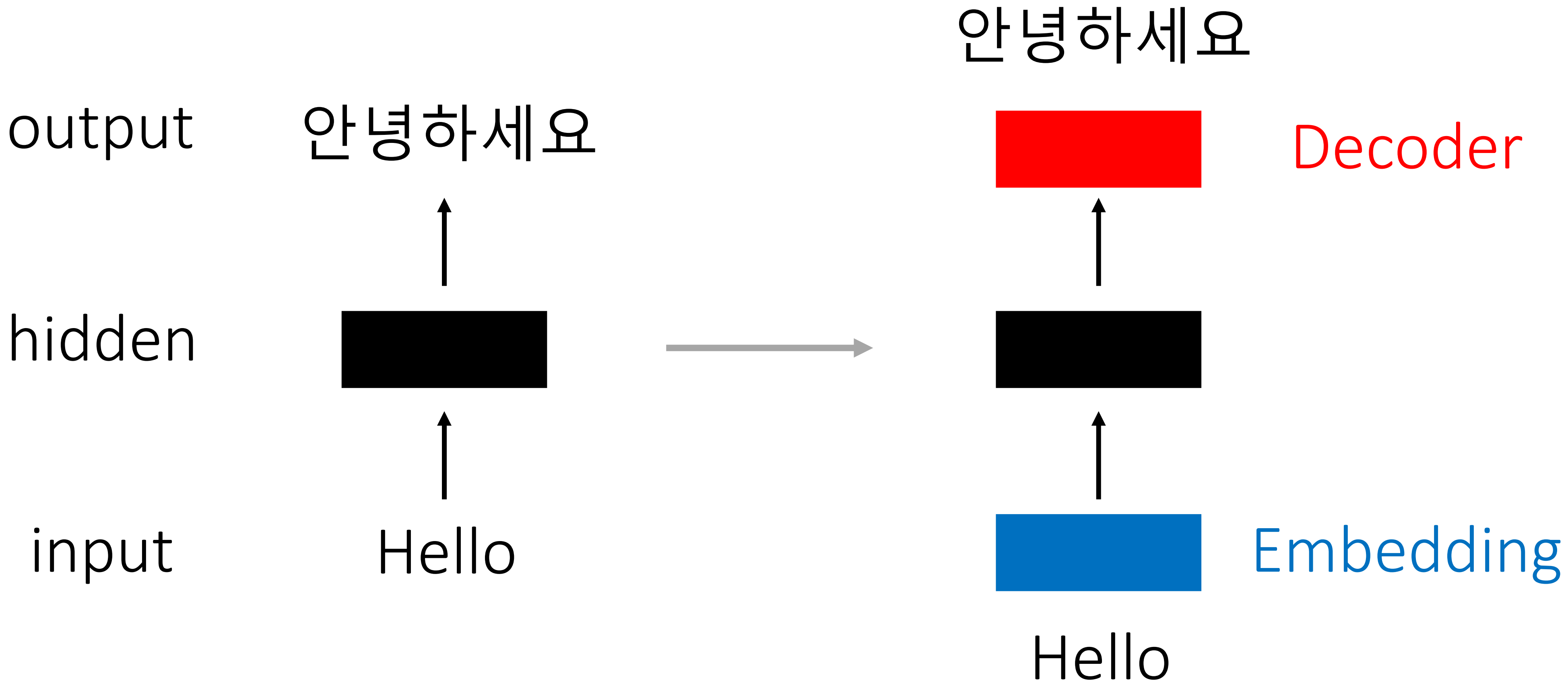
RNN in PyTorch

torch.nn.RNN(	Parameter description	Data type
- input_size	# of expected features in the input	int
- hidden_size	# of features in the hidden state	int
- num_layers	# of recurrent layers	Default = 1
- Nonlinearity	Non-linearity to use	int or tuple (default = ‘tanh’)

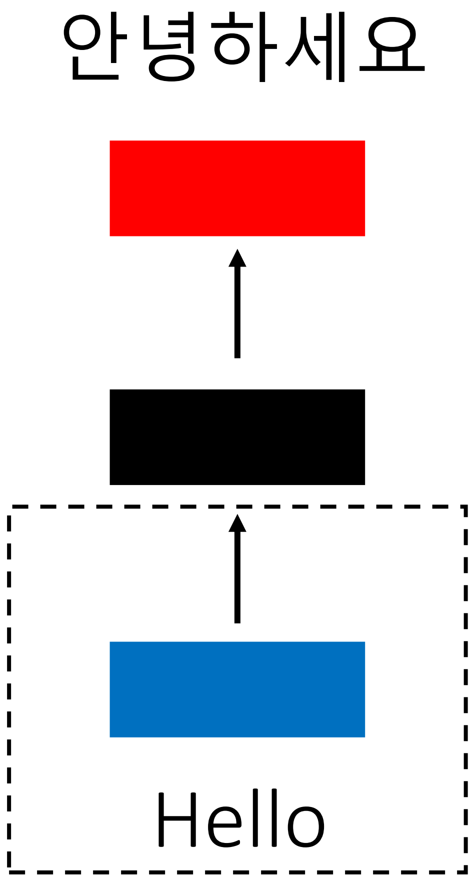
)

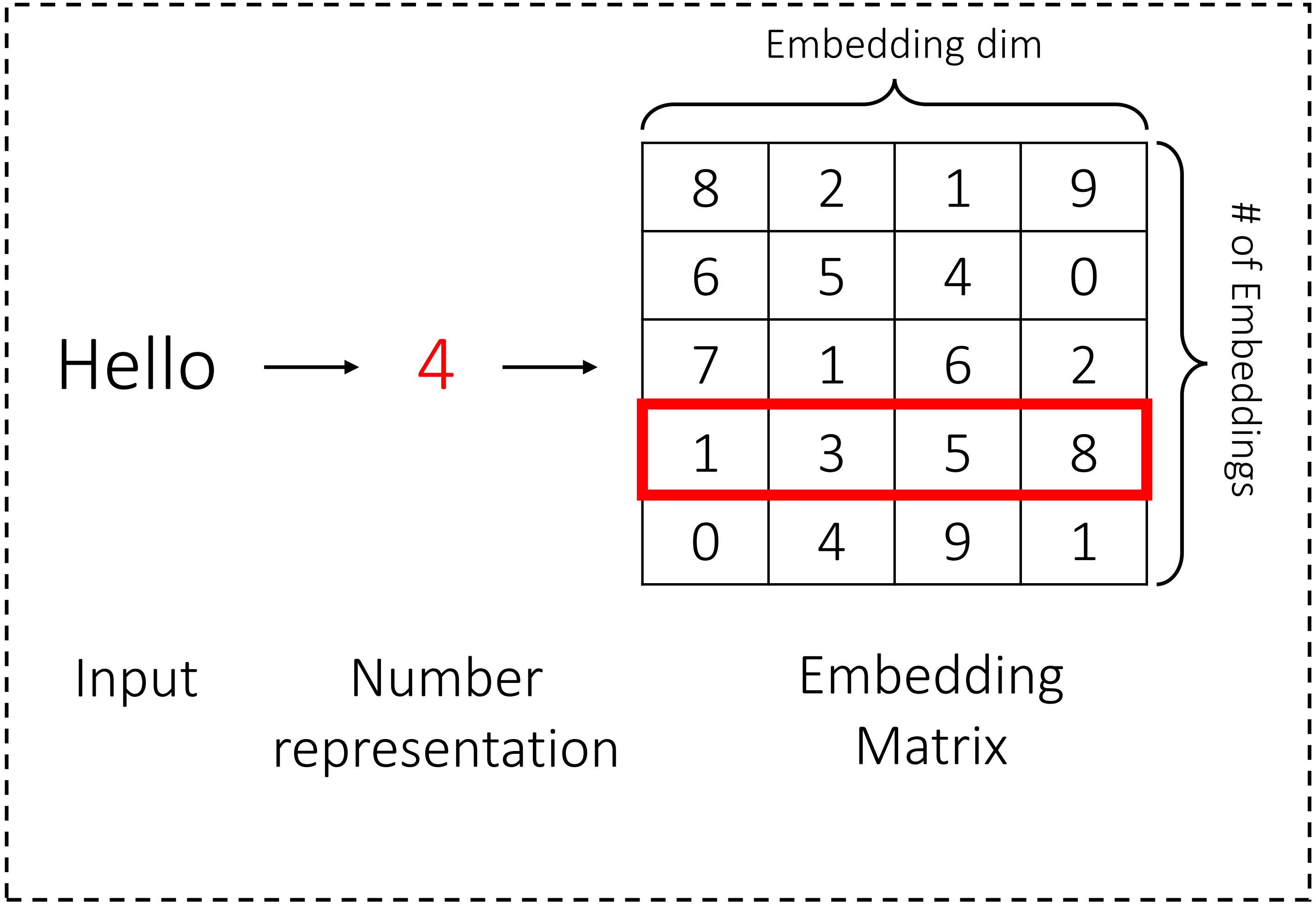
Official documentation: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

Embedding and Decoder

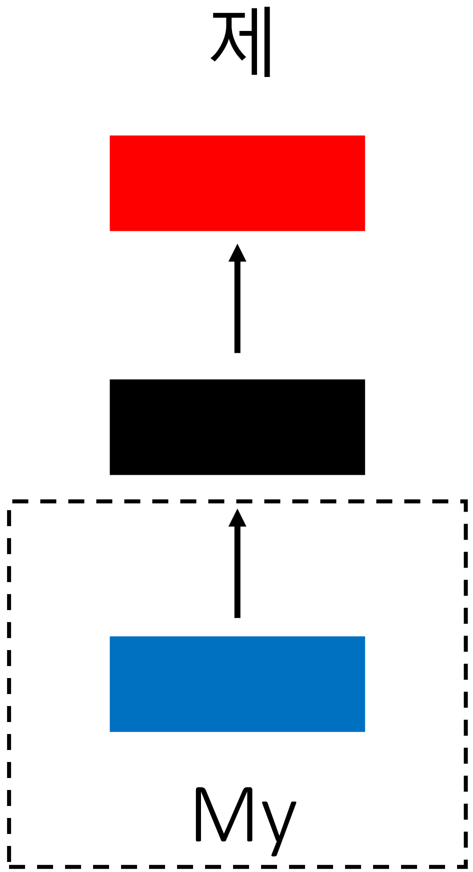


Embedding

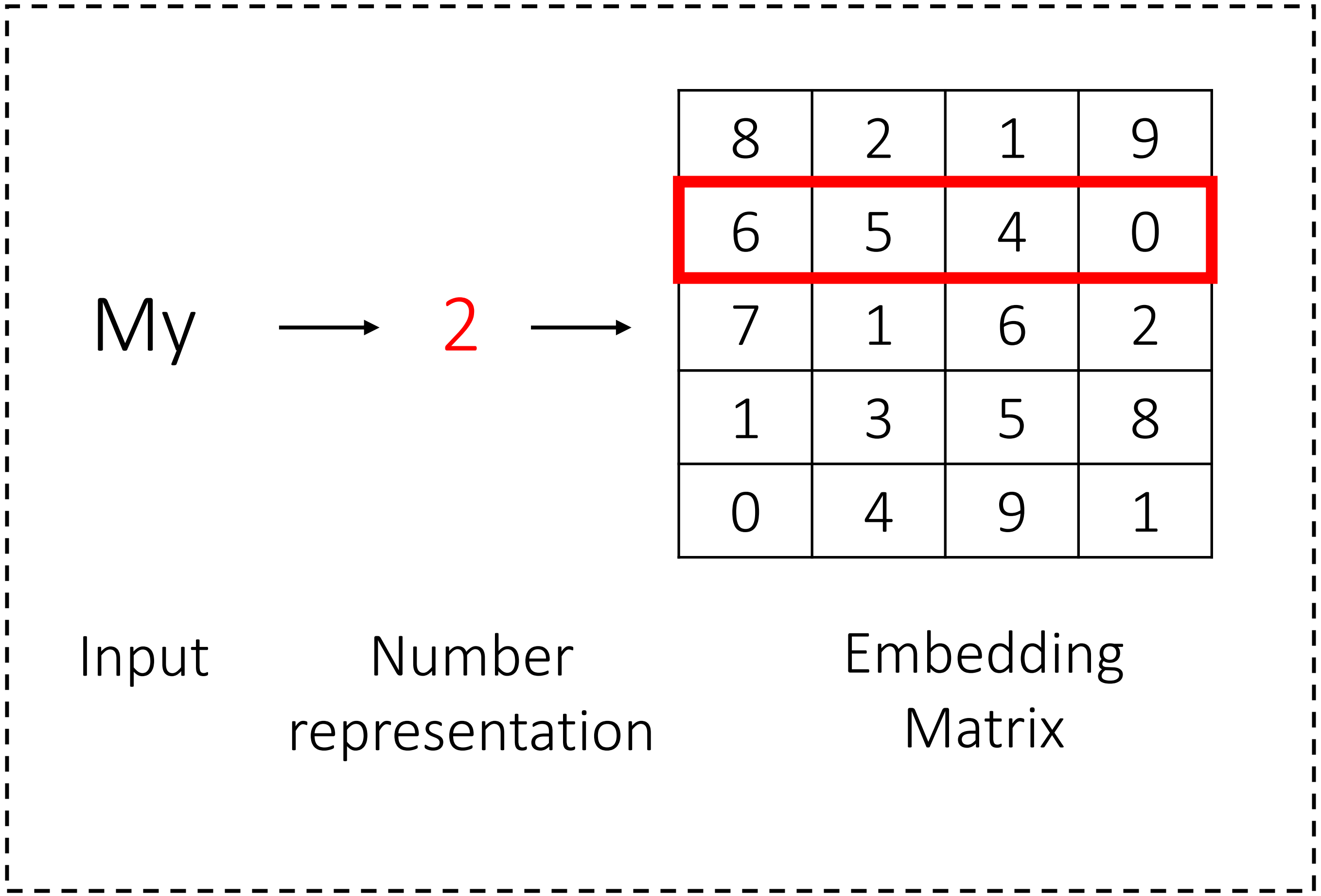

안녕하세요
Hello


Embedding dim
# of Embeddings
Hello
4
8 2 1 9
6 5 4 0
7 1 6 2
1 3 5 8
0 4 9 1
Input
Number representation
Embedding Matrix

Embedding

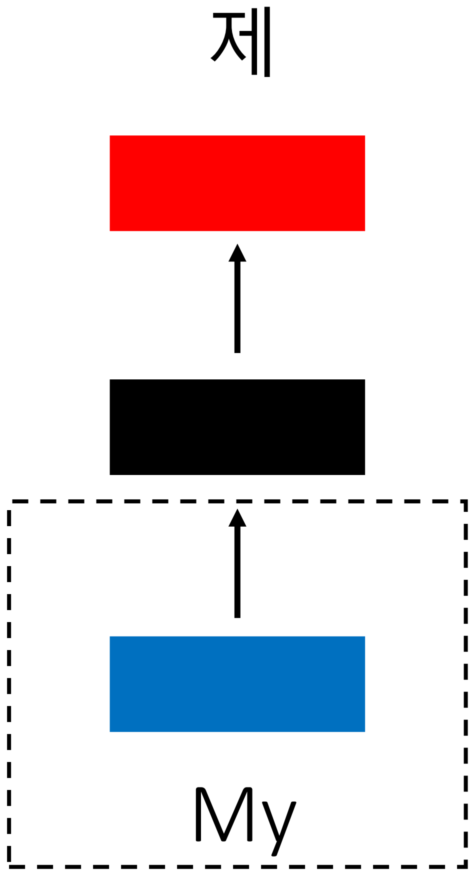


제
My



My
2
8 2 1 9
6 5 4 0
7 1 6 2
1 3 5 8
0 4 9 1
Input
Number representation
Embedding Matrix

Embedding



My → 2 →

8	2	1	9
6	5	4	0
7	1	6	2
1	3	5	8
0	4	9	1

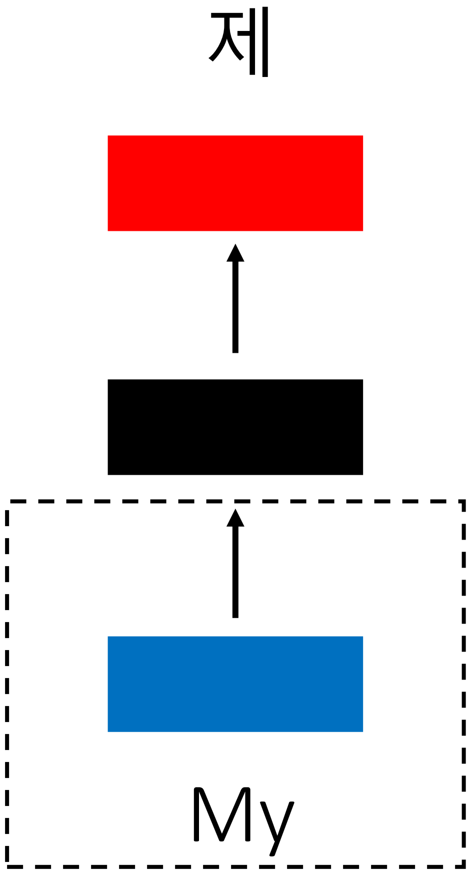
Input

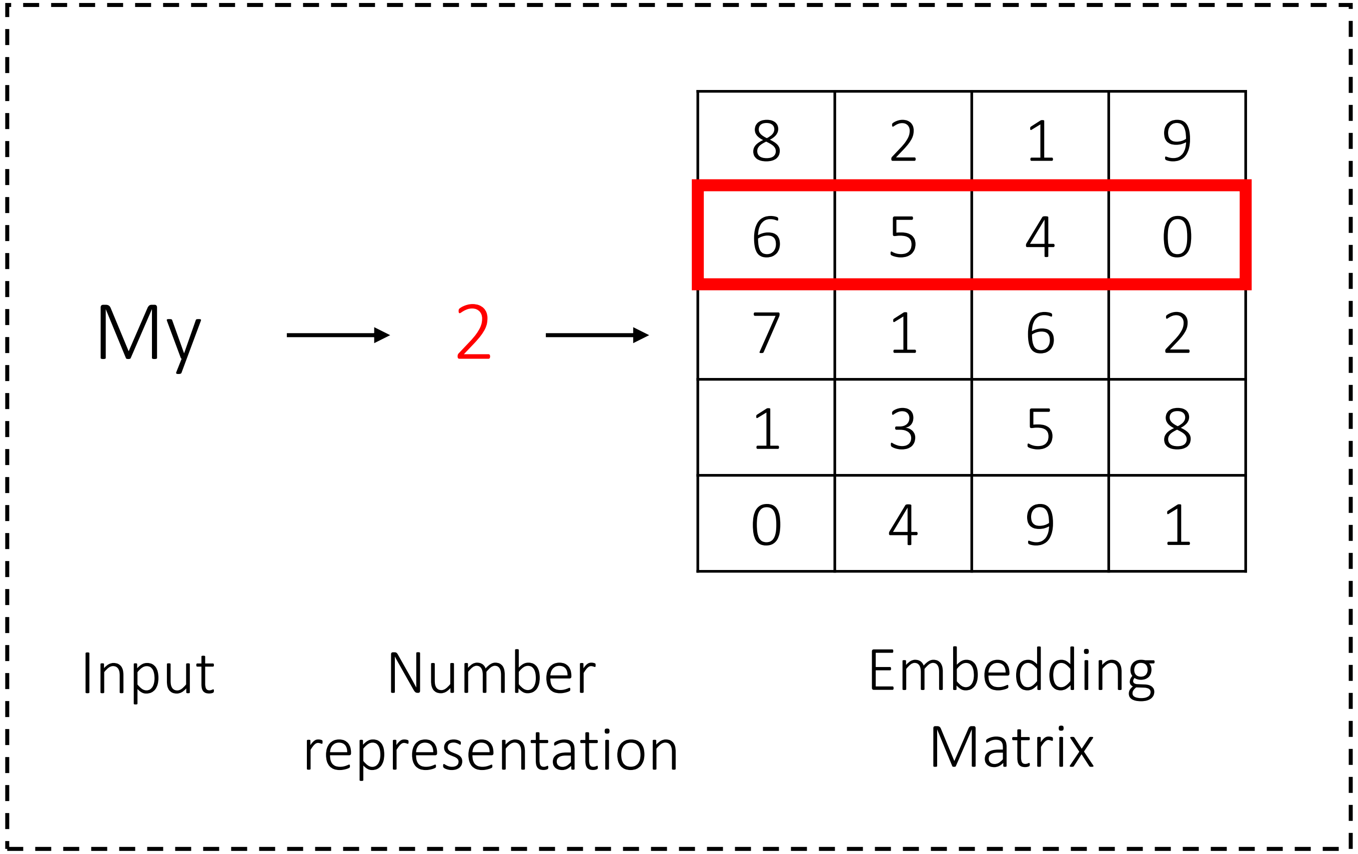
Number representation

Embedding Matrix

Embedding matrix is trainable

Embedding

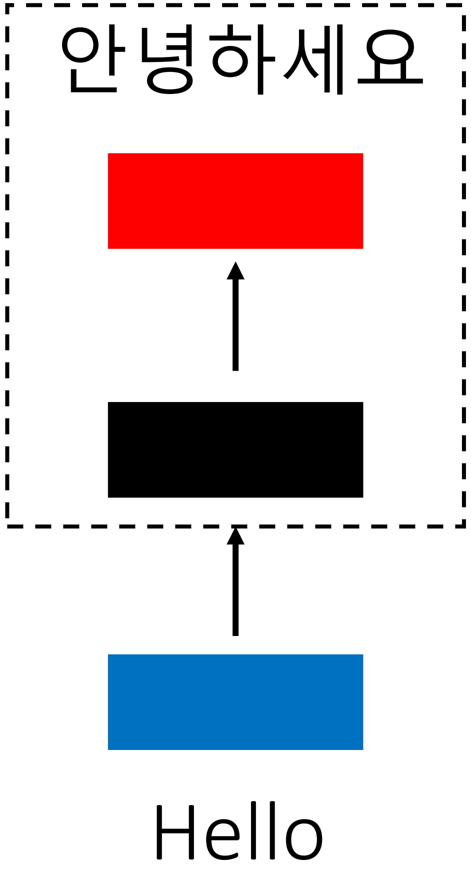




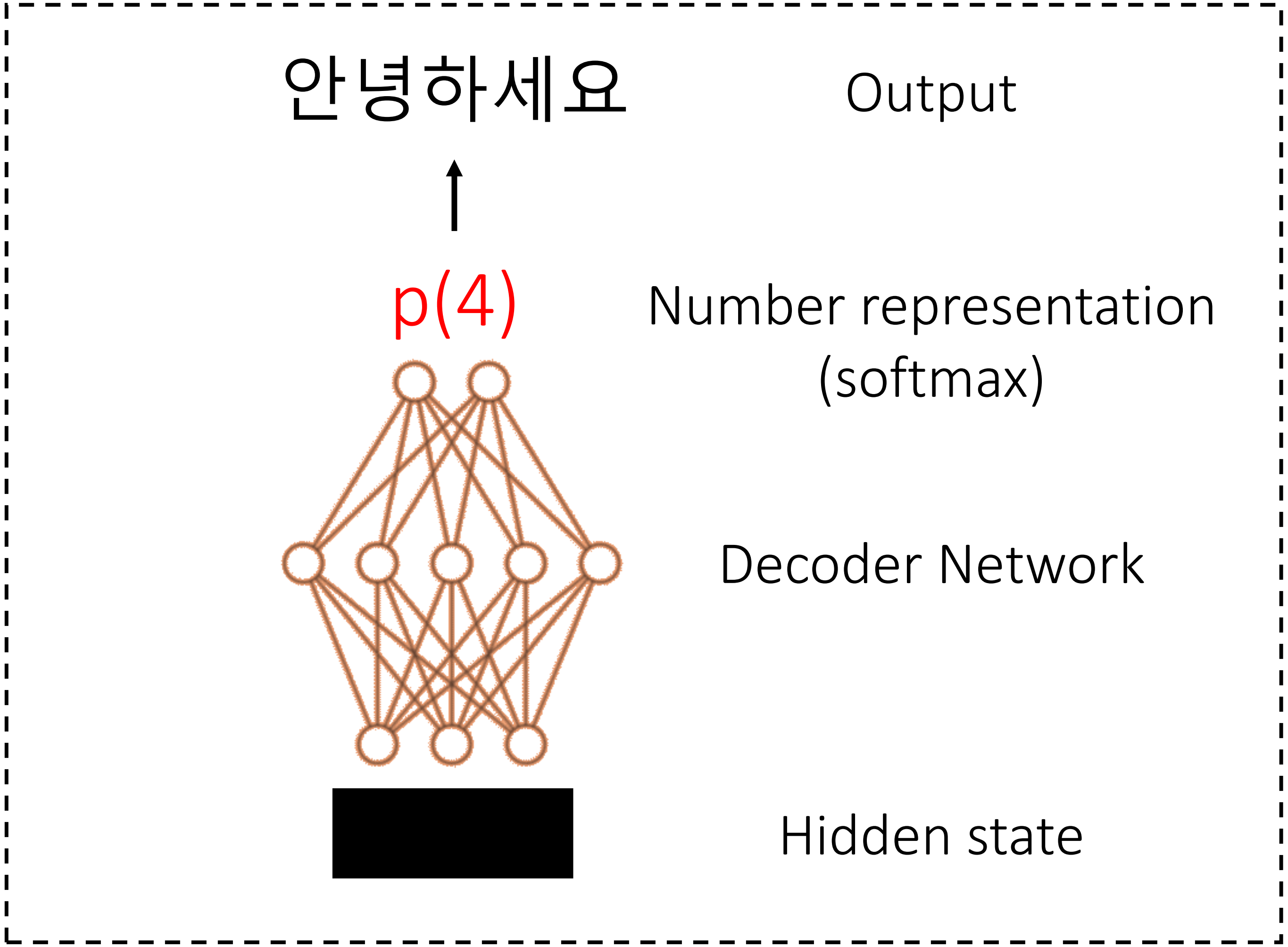
torch.nn.embedding(num_embeddings, embedding dim)

https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html

Decoder

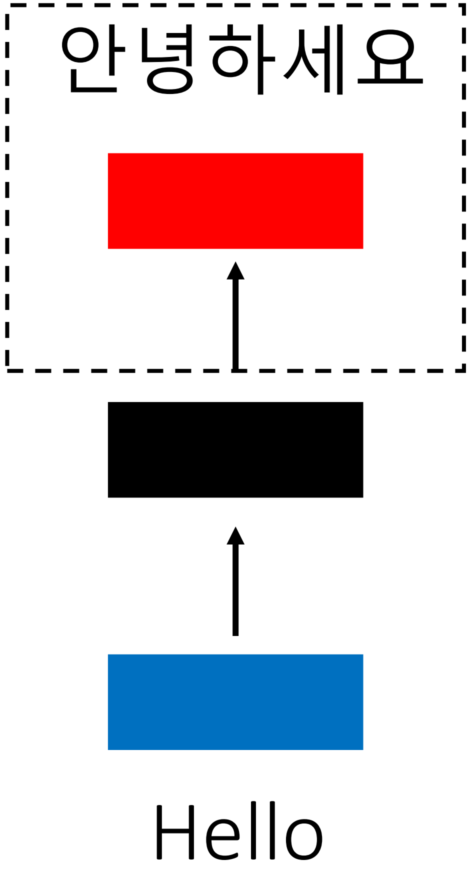


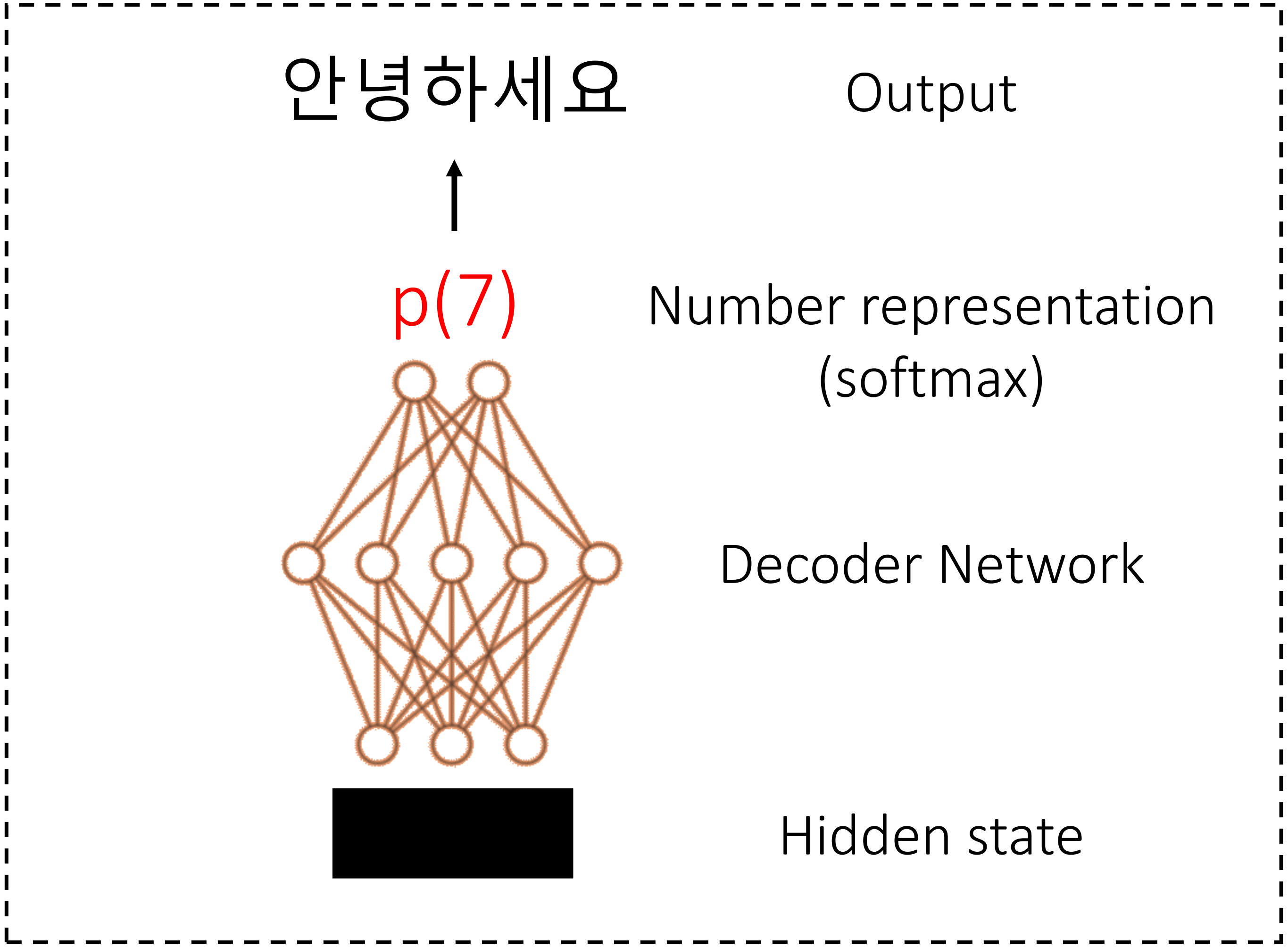
안녕하세요
Hello



안녕하세요
Output
p(4)
Number representation
(softmax)
Decoder Network
Hidden state

Decoder





torch.nn.Linear(hidden_size, output_size)

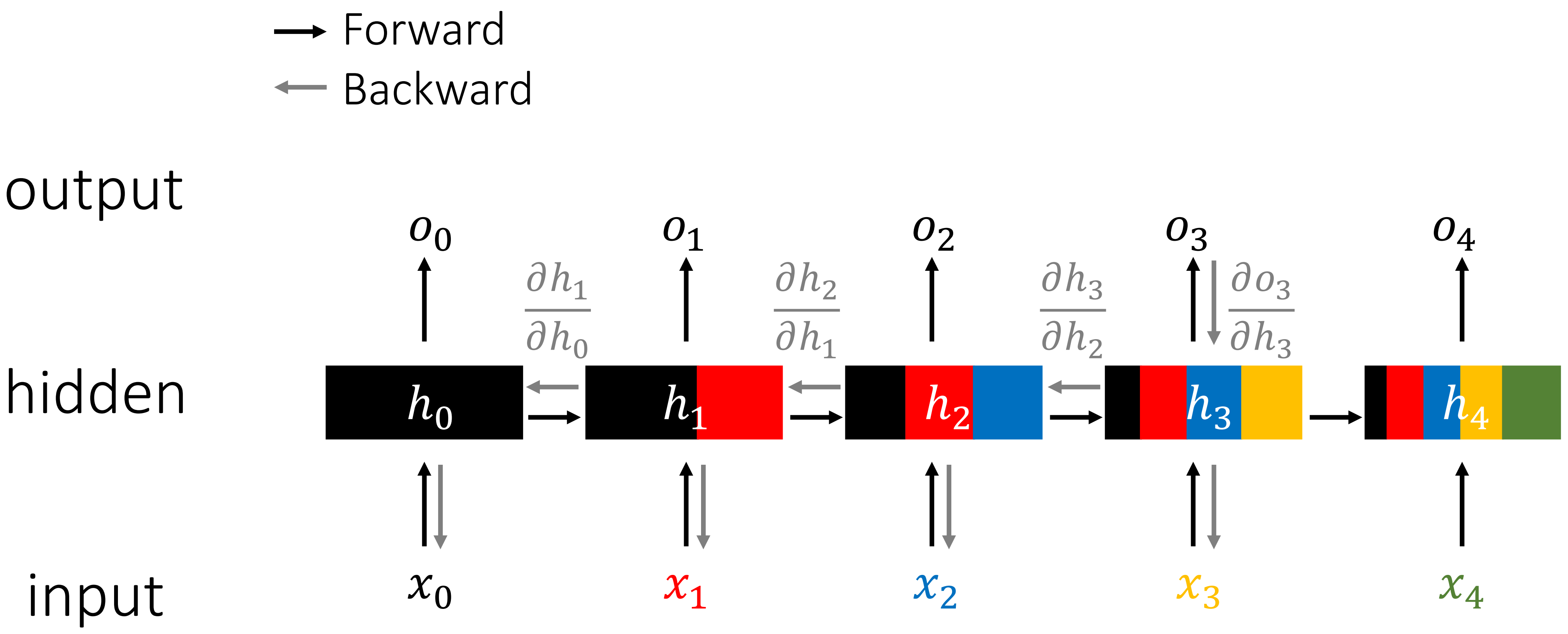
TRAINING RNNs

Backpropagation in RNNs

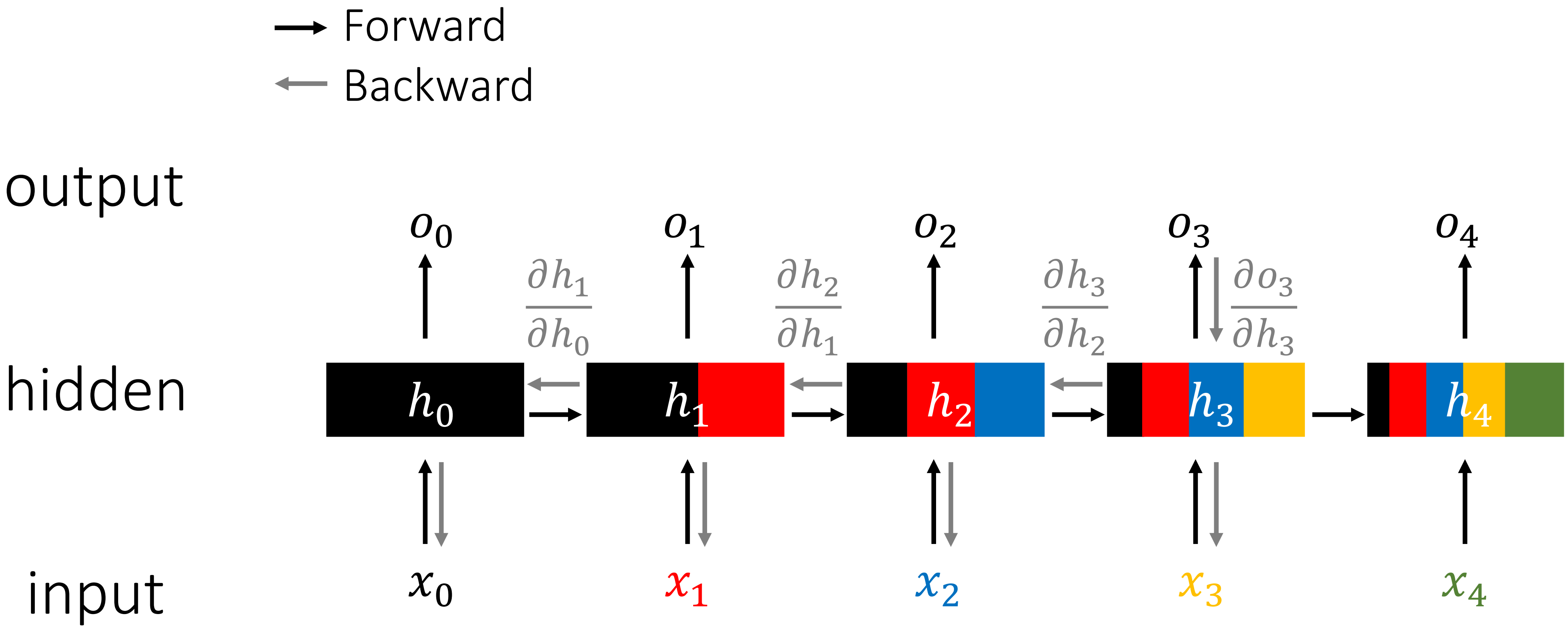
Vanishing/Exploding Gradient Problems

Training RNNs with Teacher Forcing

Backpropagation in RNNs

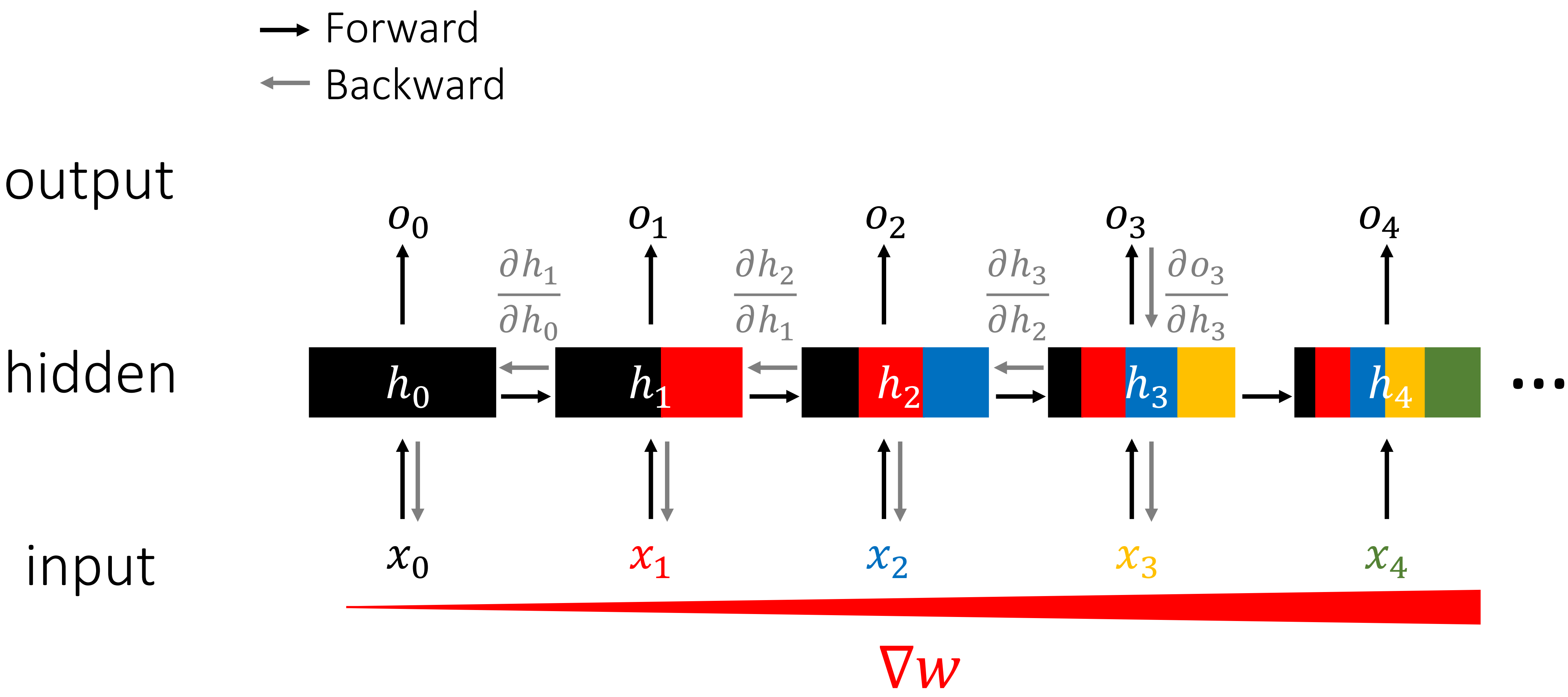


Backpropagation in RNNs

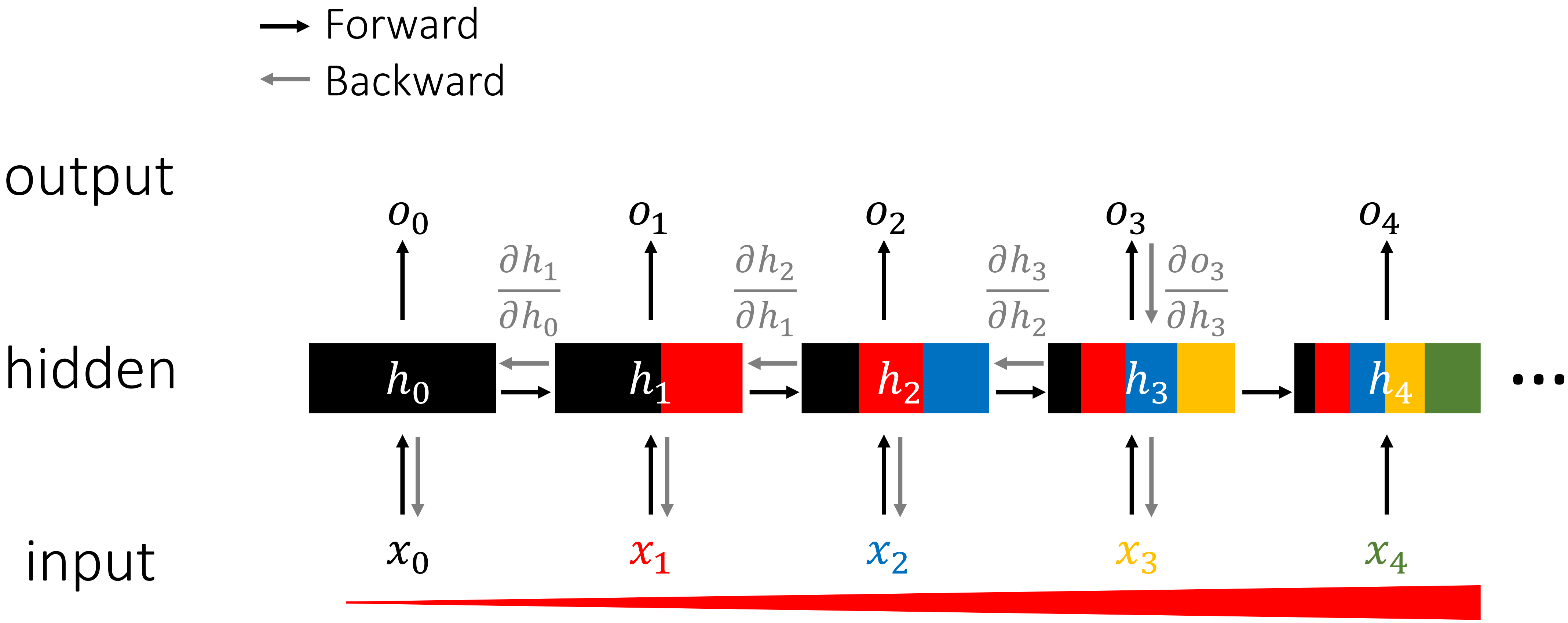


Backpropagation is performed backward in time

Vanishing and Exploding Gradients



Vanishing and Exploding Gradients



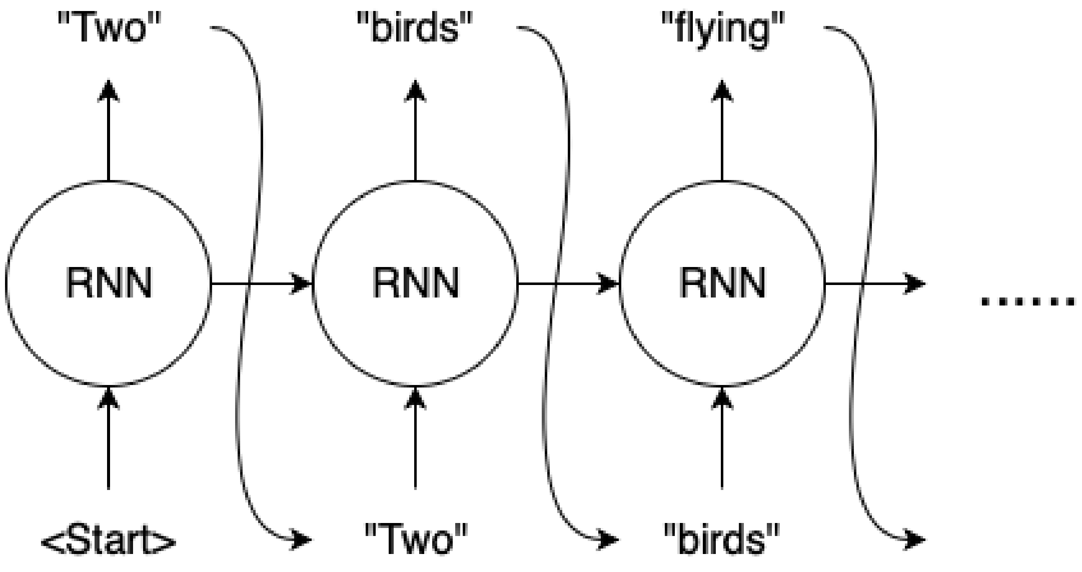
Longer input sequence →

higher risk of Vanishing/Exploding Gradients!

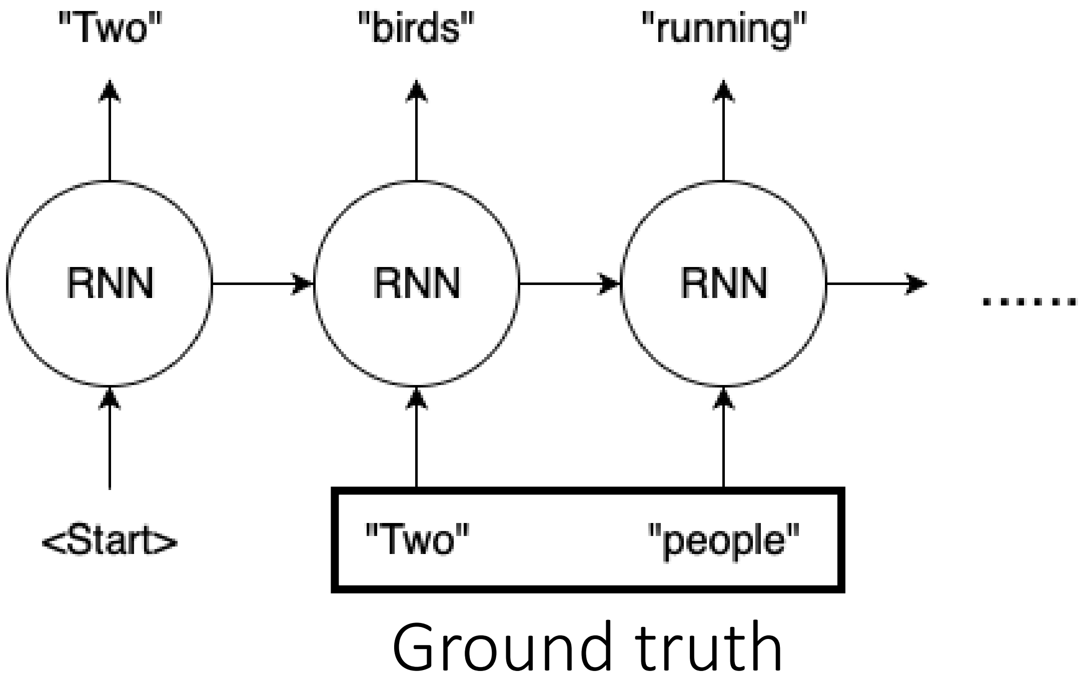
Vanishing and Exploding Gradients

- Use gated RNN architecture e.g., LSTM, GRU
- ReLU activation as nonlinearity
- Smaller number of sequence
- Smaller learning rate

Training RNN with Teacher Forcing



Without Teacher Forcing



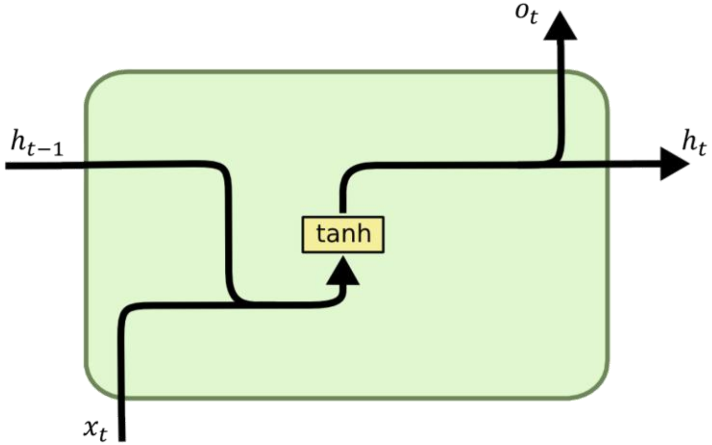
With Teacher Forcing

GATED RNNs

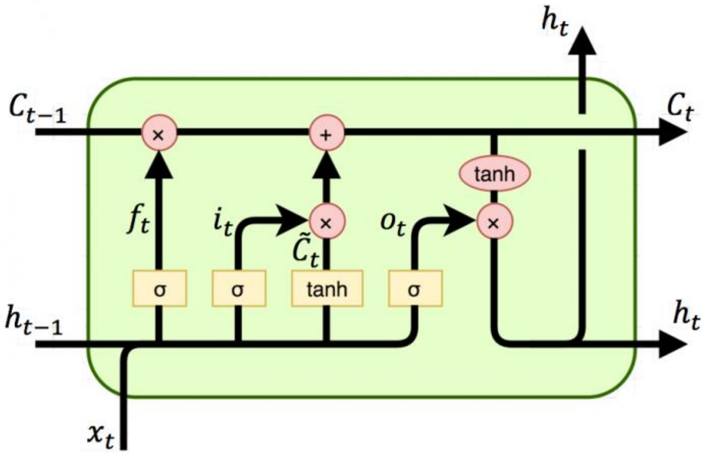
Long Short-Term Memory (LSTM)

Gated Recurrent Unit (GRU)

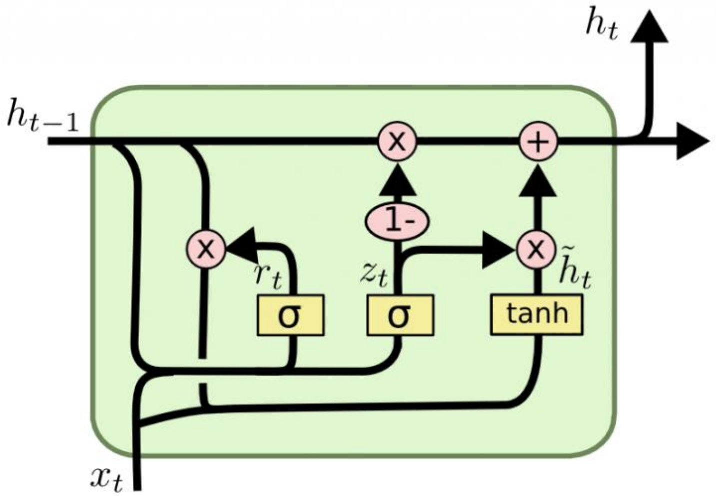
Gated RNNs



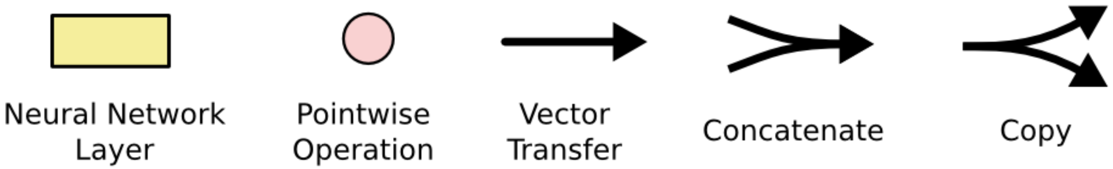
Vanilla RNN



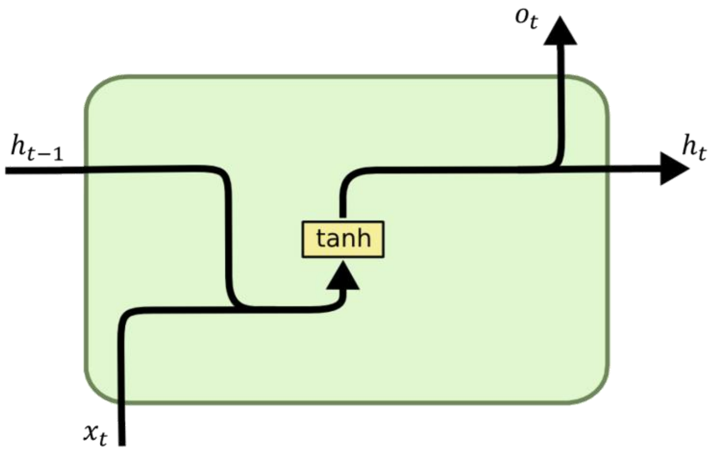
LSTM



GRU

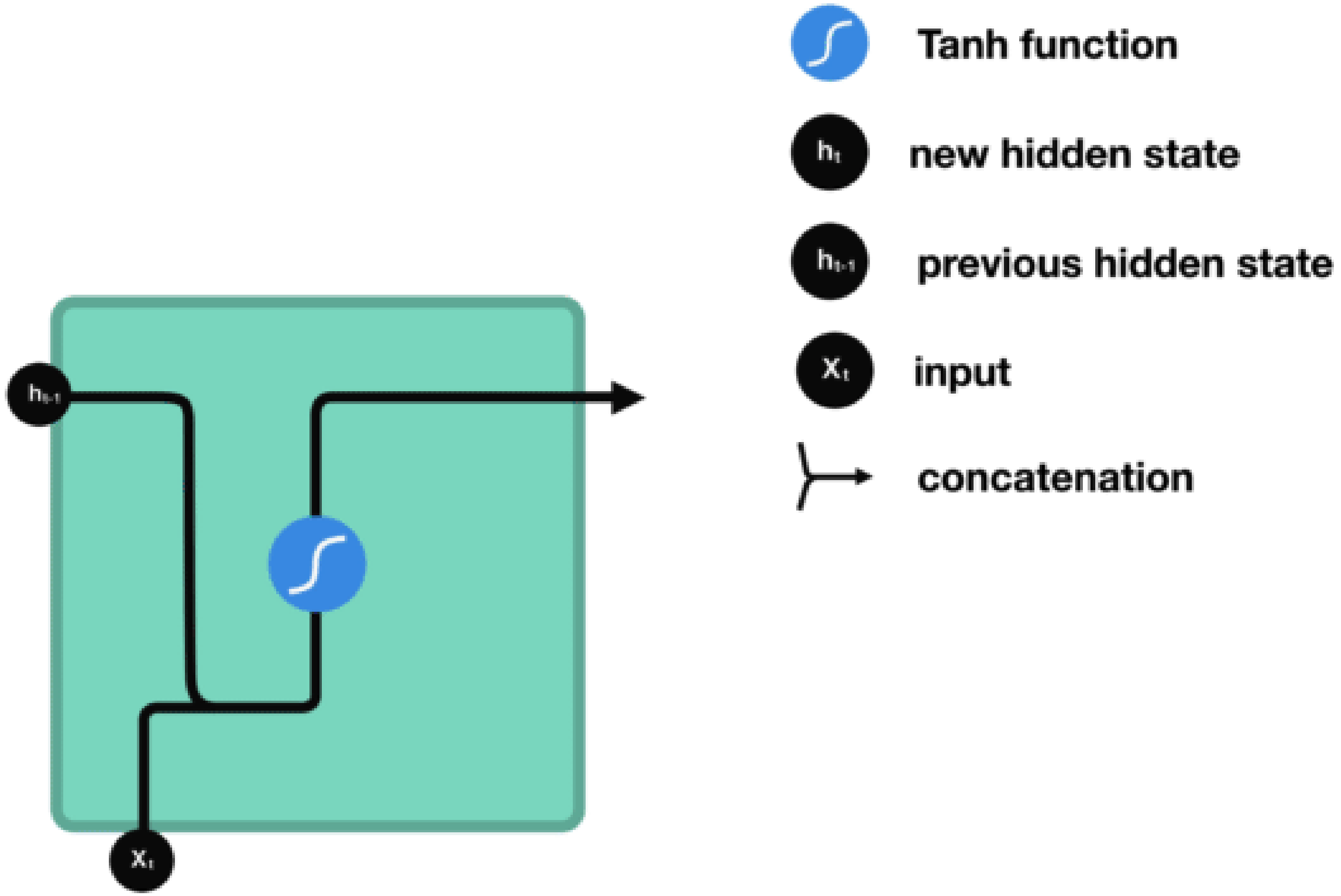


Vanilla RNN



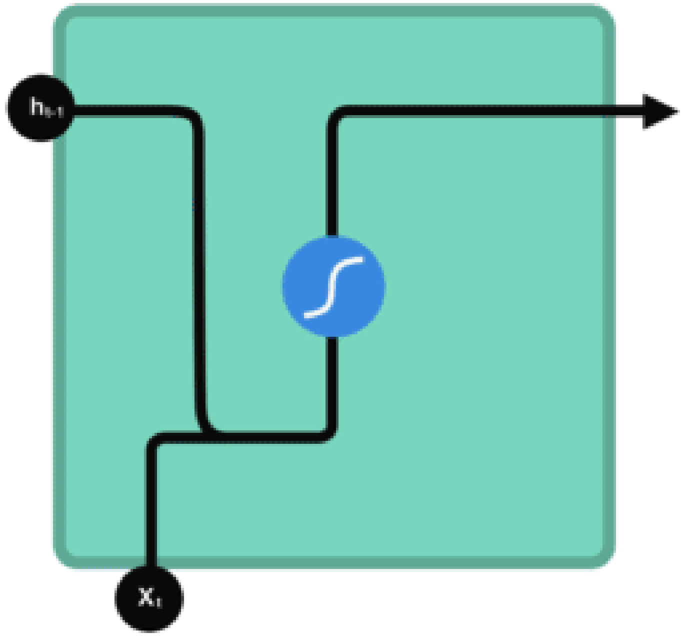
Vanilla RNN

Vanilla RNN



h_{t-1}
X_t
Tanh function
h_t
new hidden state
h_{t-1}
previous hidden state
X_t
input
concatenation

Vanilla RNN



Tanh function

new hidden state

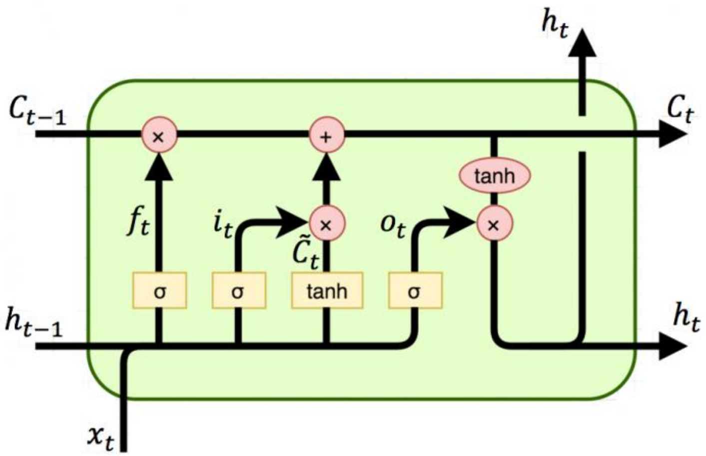
previous hidden state

input

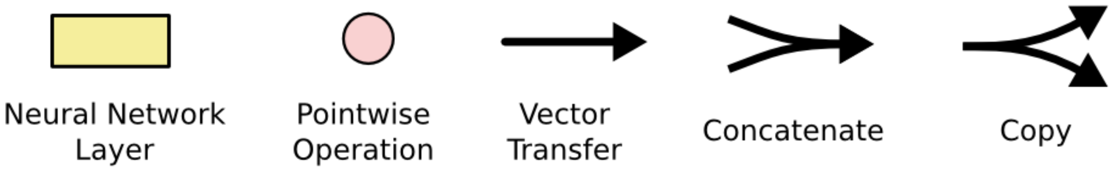
concatenation

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

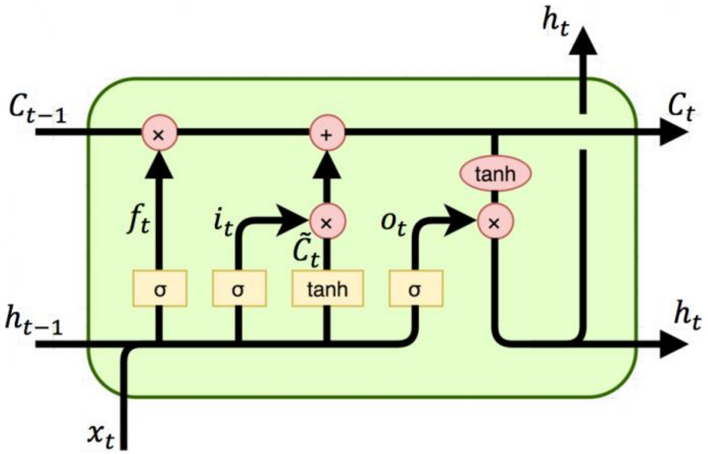
LSTM (Long Short-Term Memory)



LSTM

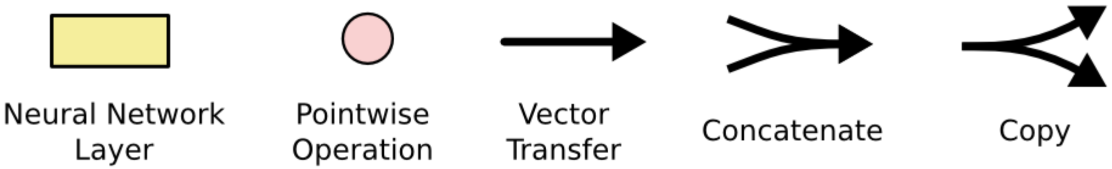


LSTM (Long Short-Term Memory)

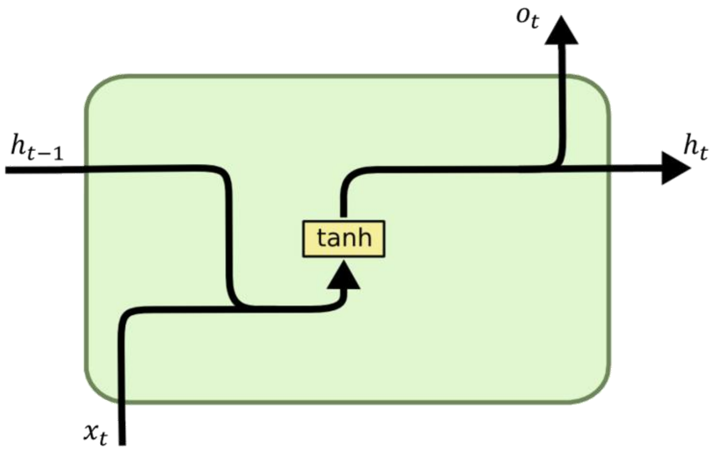


LSTM

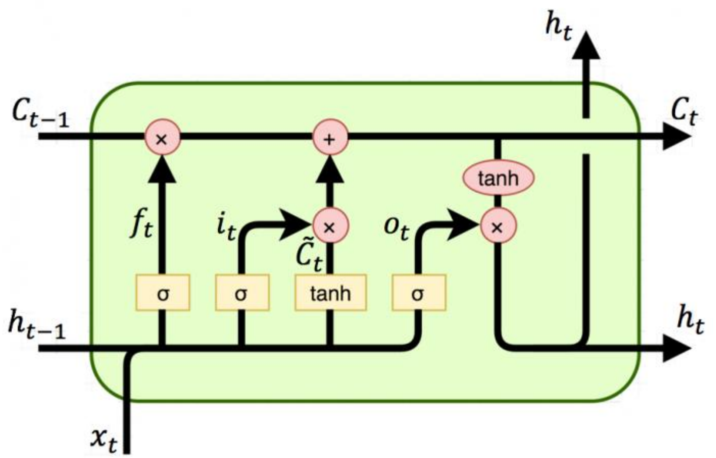
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$h_t = o_t \circ \sigma_h(c_t)$$



LSTM (Long Short-Term Memory)



Vanilla RNN



LSTM

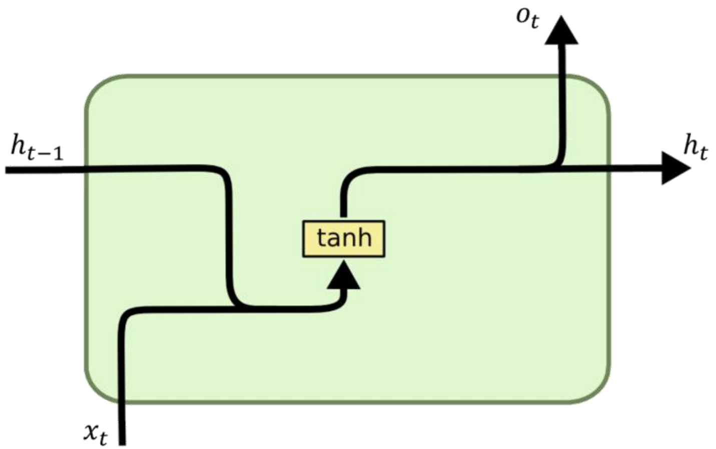
$$h_t = \sigma(wh_{t-1}).$$

$$\frac{\partial h_{t'}}{\partial h_t} = \prod_{k=1}^{t'-t} w\sigma'(wh_{t'-k})$$

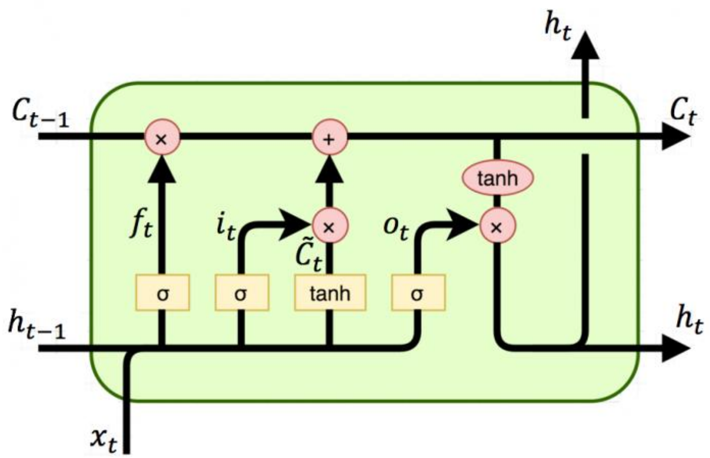
$$= \underbrace{w^{t'-t}}_{!!!} \prod_{k=1}^{t'-t} \sigma'(wh_{t'-k})$$

$$\frac{\partial c_{t'}}{\partial c_t} = \prod_{k=1}^{t'-t} \sigma(v_{t+k}).$$

LSTM (Long Short-Term Memory)



Vanilla RNN



LSTM

$$h_t = \sigma(wh_{t-1}).$$

$$\frac{\partial h_{t'}}{\partial h_t} = \prod_{k=1}^{t'-t} w\sigma'(wh_{t'-k})$$

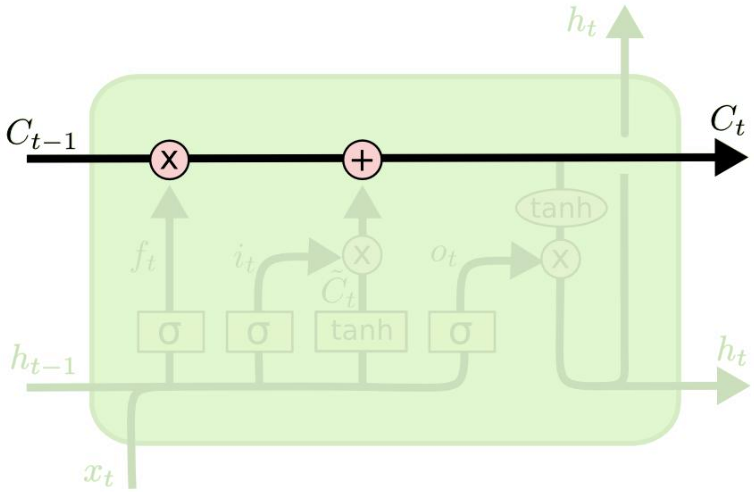
$$= \underbrace{w^{t'-t}}_{!!!} \prod_{k=1}^{t'-t} \sigma'(wh_{t'-k})$$

Gradient decays or grow exponentially if $w \neq 1$

$$\frac{\partial c_{t'}}{\partial c_t} = \prod_{k=1}^{t'-t} \sigma(v_{t+k}).$$

No exponential decay or growth term

LSTM: Detailed Architecture

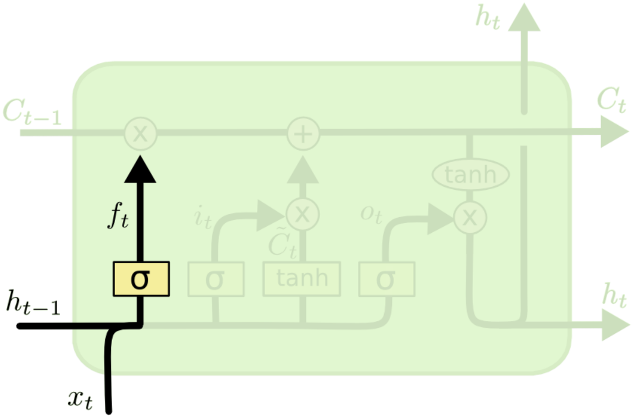


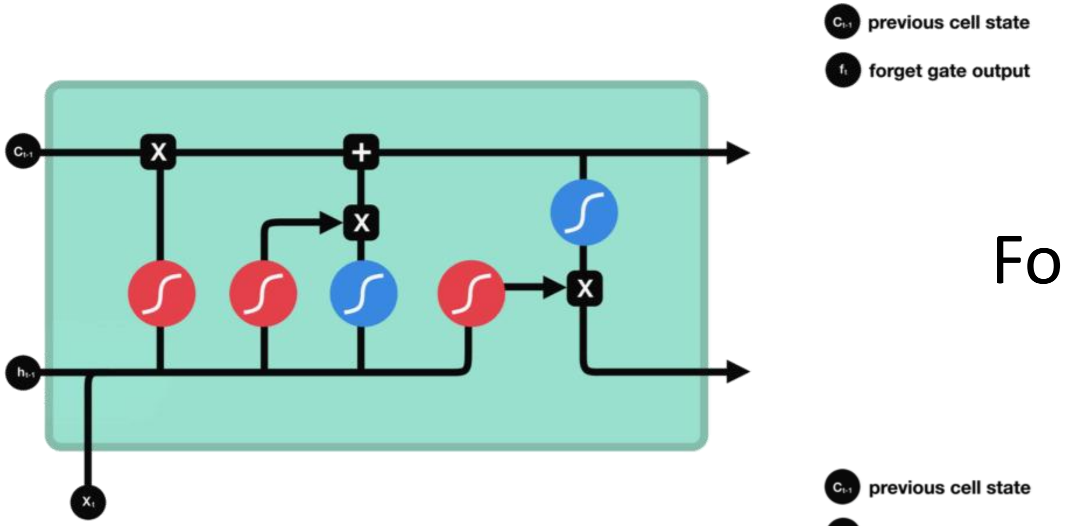
Cell state

- Unique to LSTM
- Long term memory of the model

LSTM: Detailed Architecture

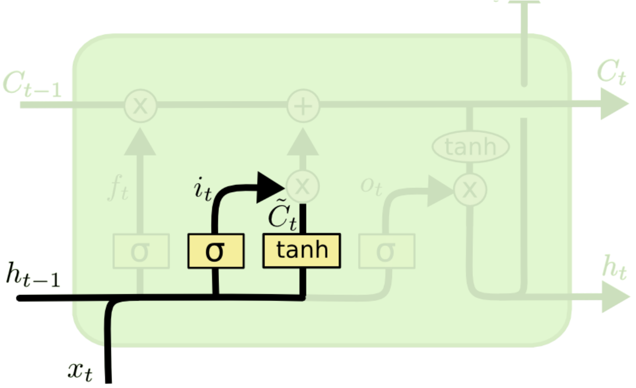
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

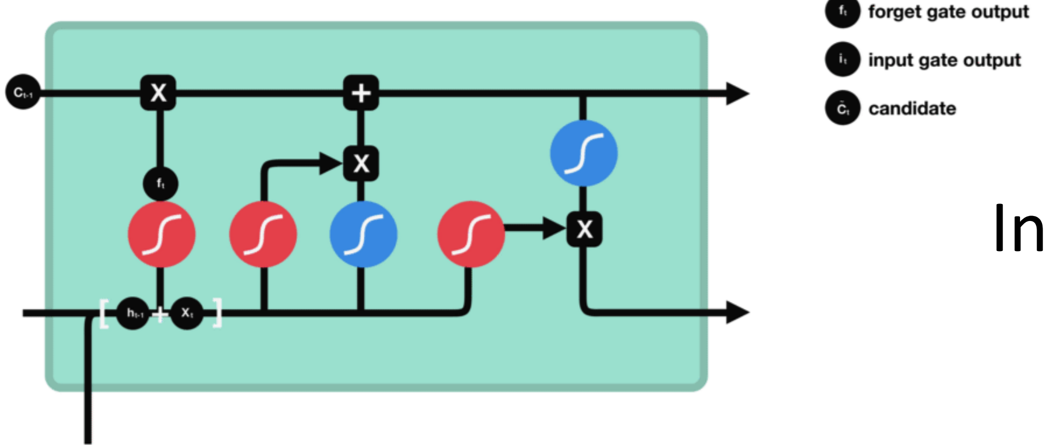




Forget gate layer

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

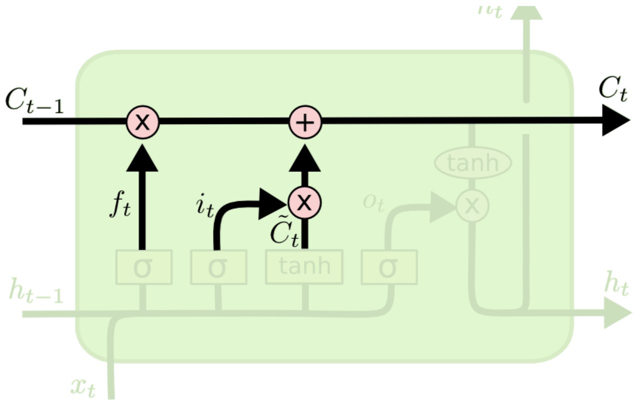


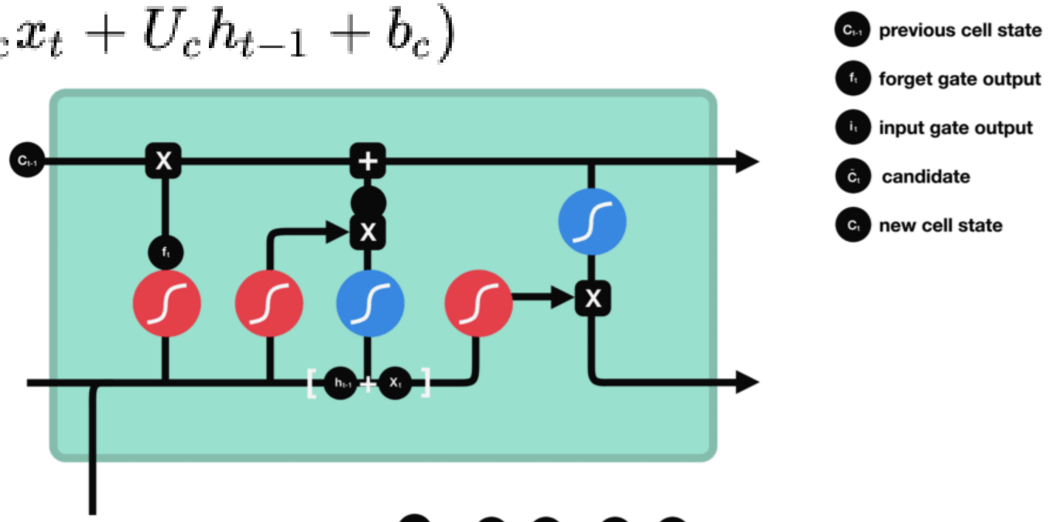


Input gate layer

LSTM: Detailed Architecture

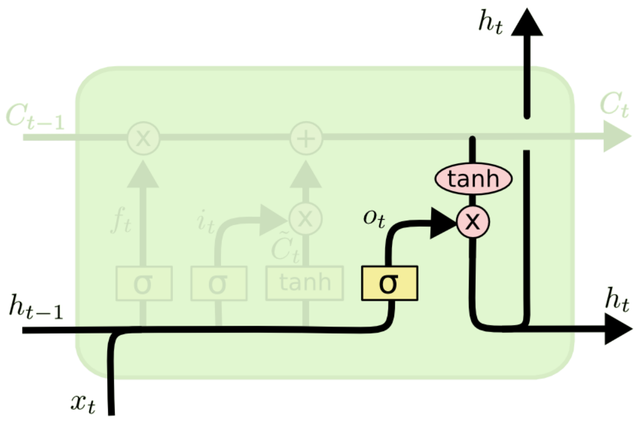
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

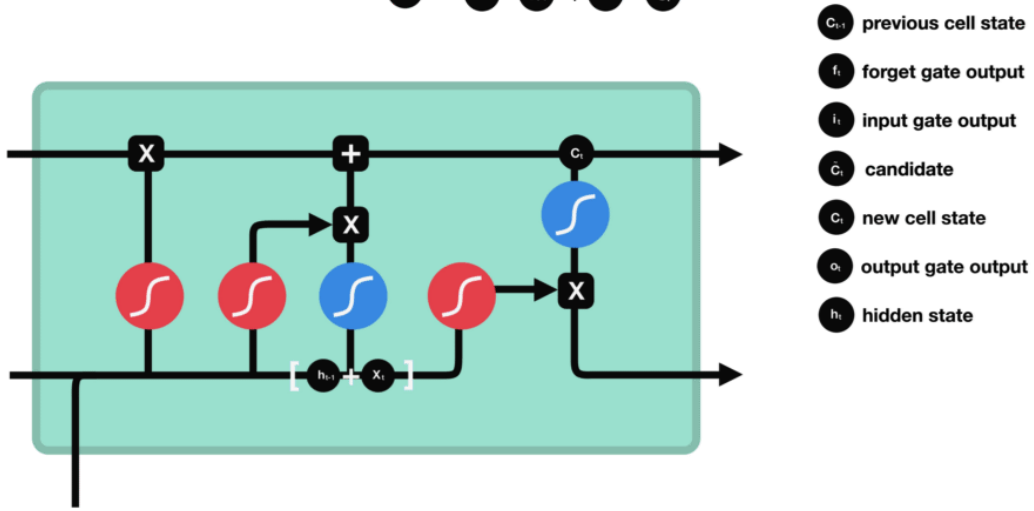




Update cell state

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$





Output gate layer

LSTM: Detailed Architecture

Forget gate

Decides what is relevant to keep from previous steps

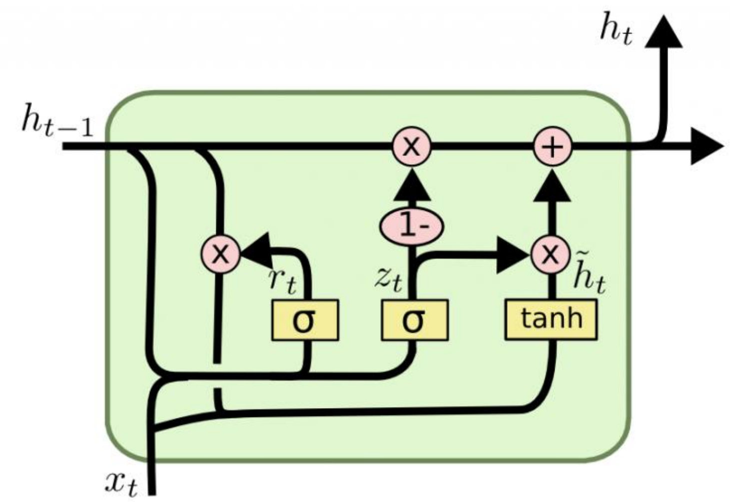
Input gate

Decides what information is relevant to add from the current step

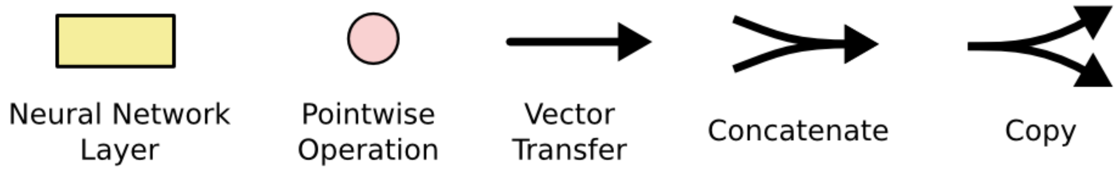
Output Gate

Determines what the next hidden state should be

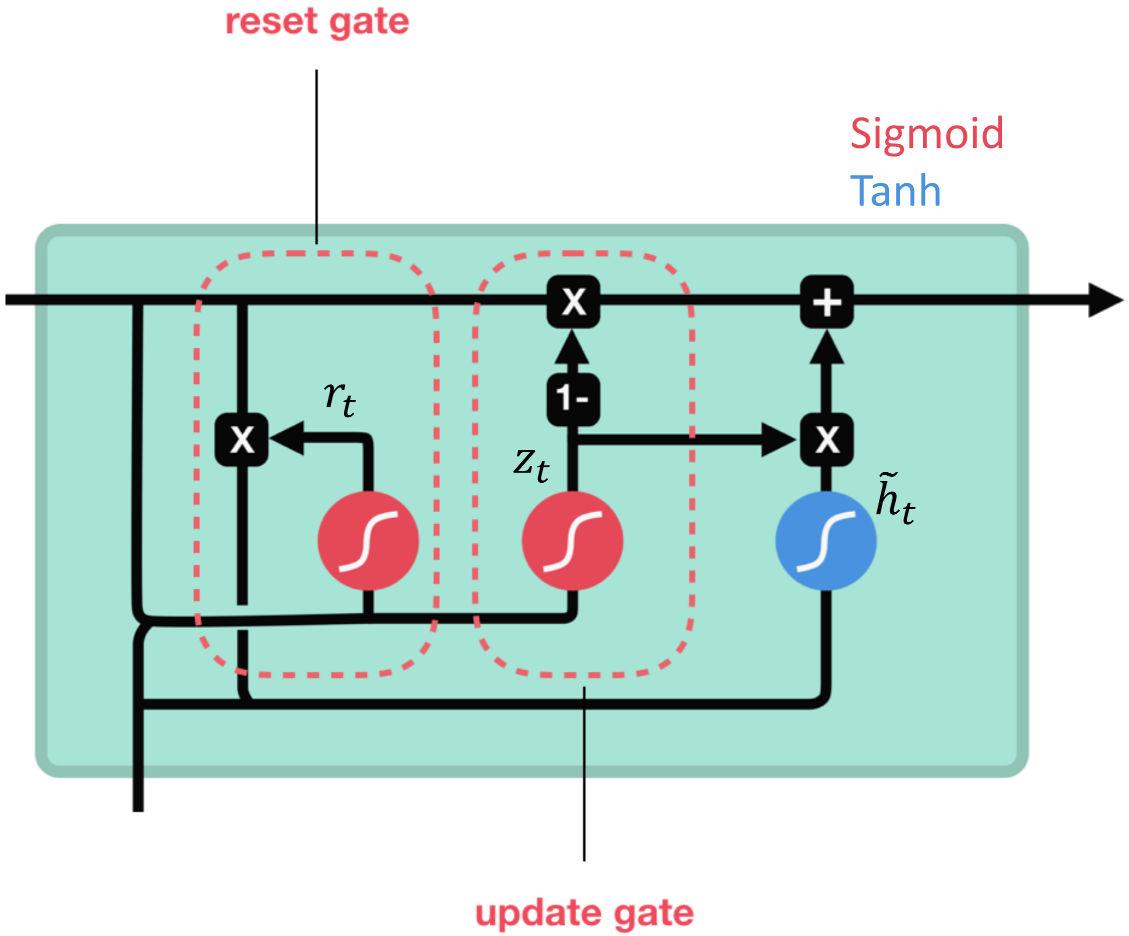
Gated RNNs



GRU



GRU: Detailed Architecture



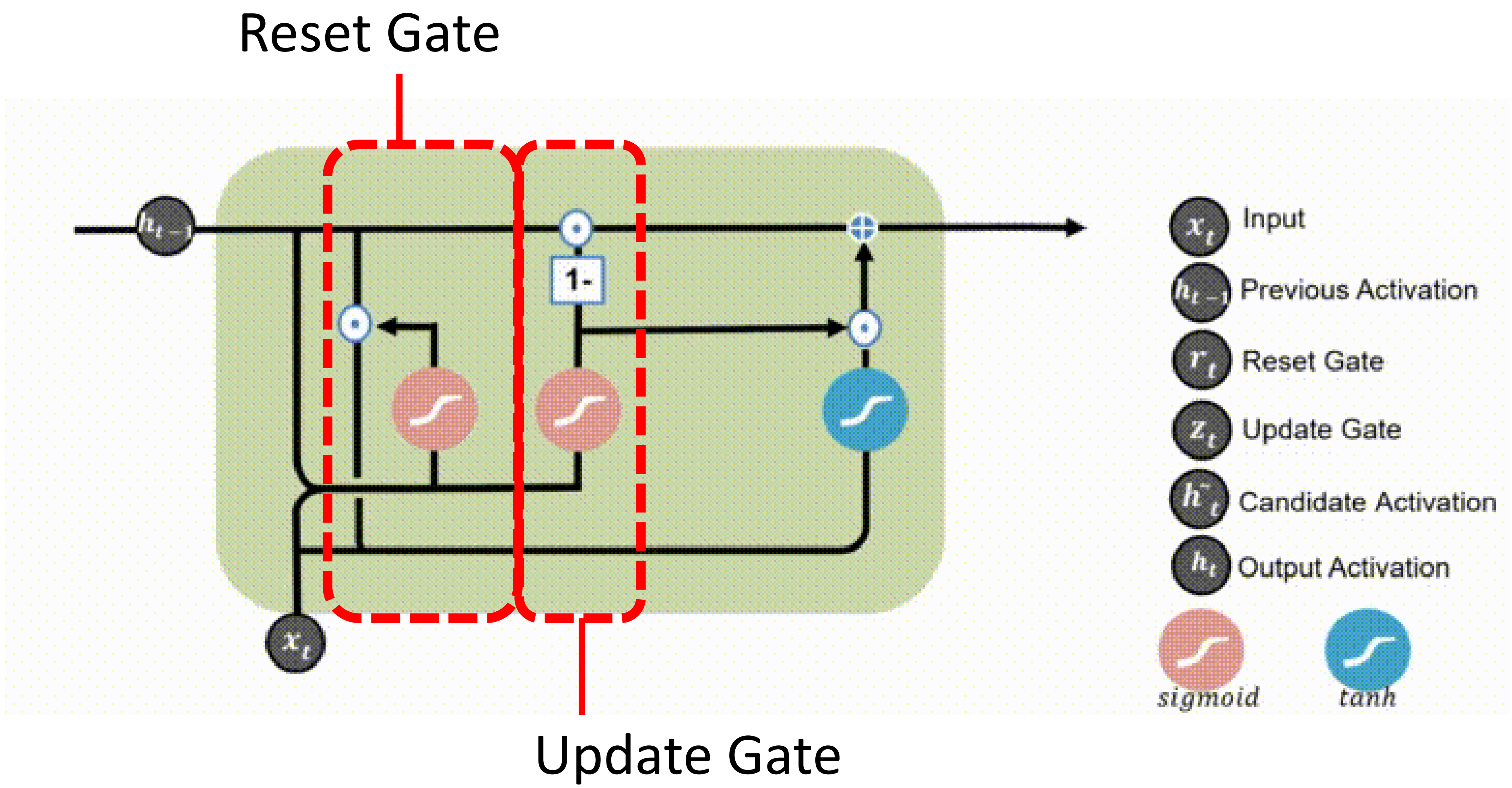
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Information Flow in GRU



GRU: Detailed Architecture

Update gate

How much of the past information needs to be retained

Reset gate

How much of the past information to forget

LSTM in PyTorch

torch.nn.LSTM(	Parameter description	Data type
- input_size	# of expected features in the input	int
- hidden_size	# of features in the hidden state	int
- num_layers	# of recurrent layers	Default = 1
)		

Official documentation: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

GRU in PyTorch

torch.nn.GRU(	Parameter description	Data type
- input_size	# of expected features in the input	int
- hidden_size	# of features in the hidden state	int
- num_layers	# of recurrent layers	Default = 1
)		

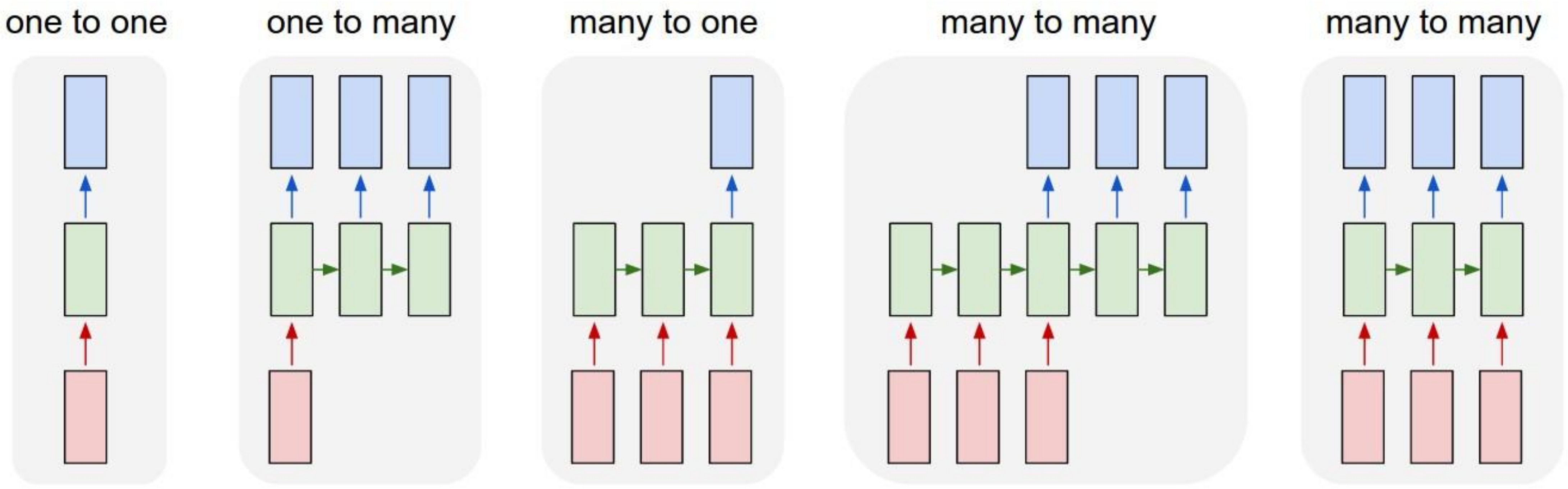
Official documentation: https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

RNN PROBLEM TYPES

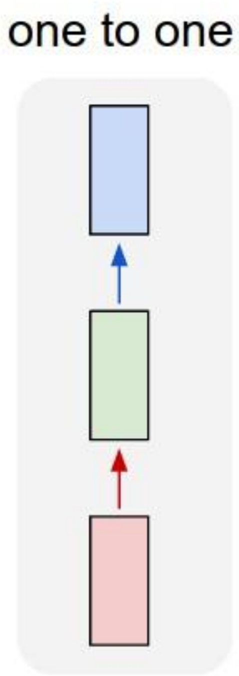
RNN Configurations

RNN Extensions

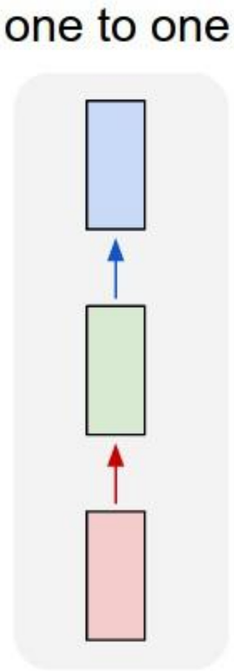
RNN Configurations



One to One

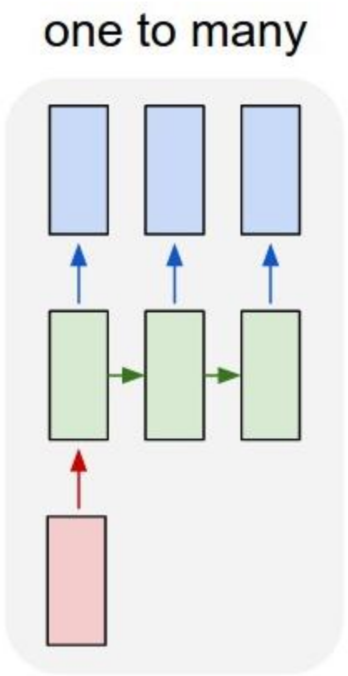


One to One



Identical to Feed Forward Network

One to Many



One to Many

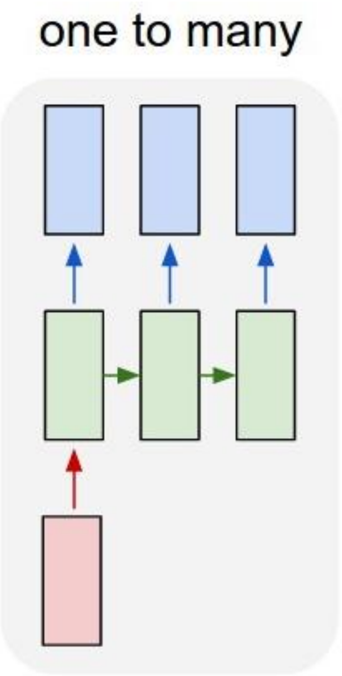
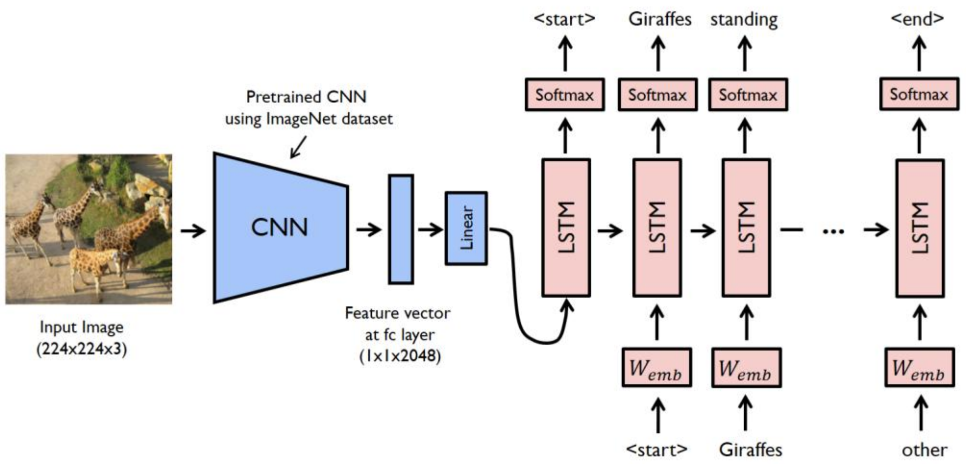
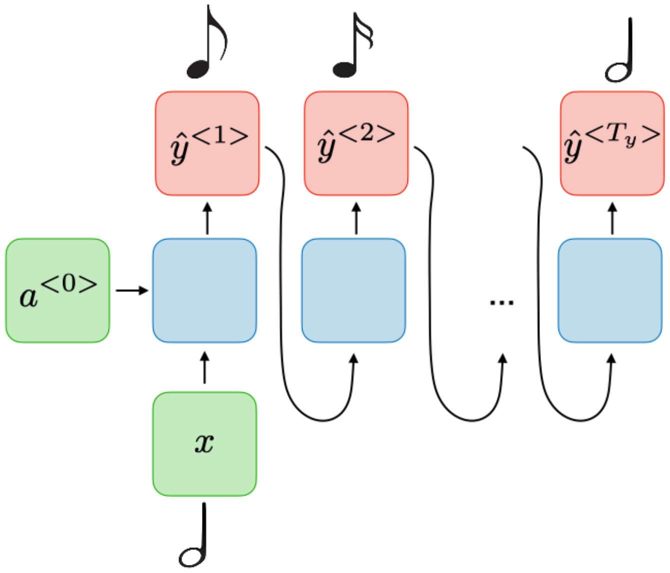


Image credit: www.analyticsvidhya.com

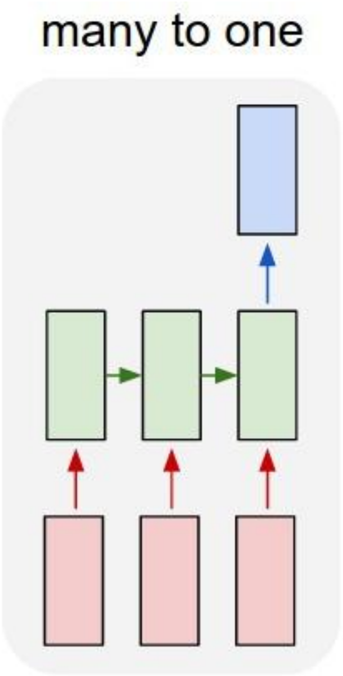
Image captioning



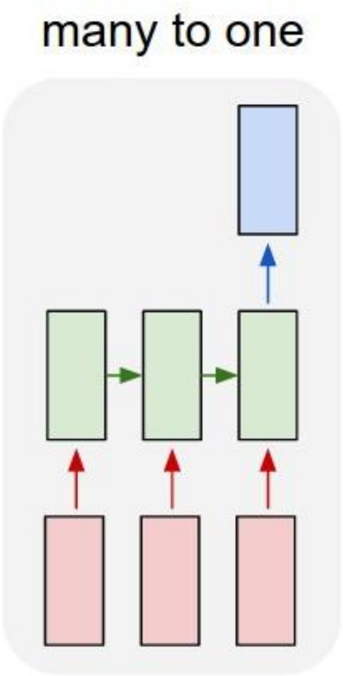
Music generation

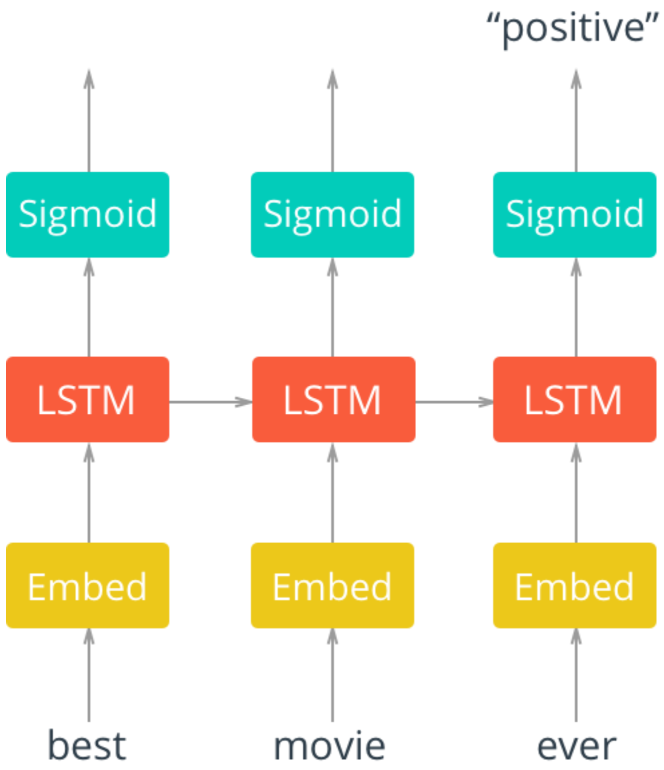


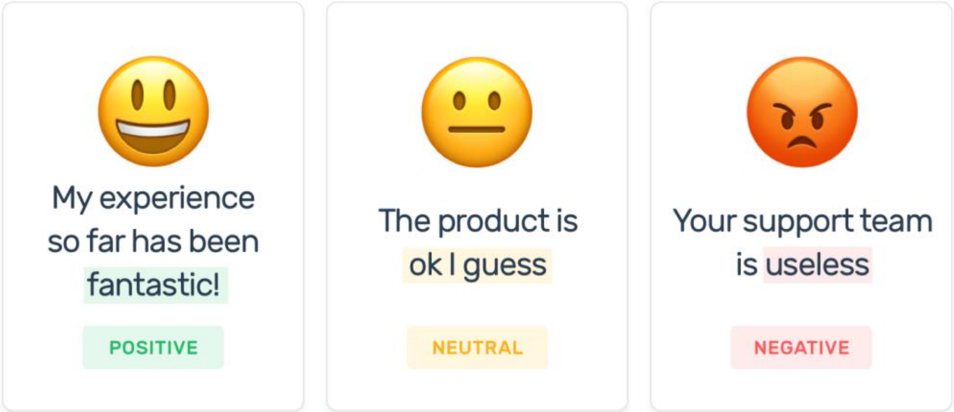
Many to One



Many to One

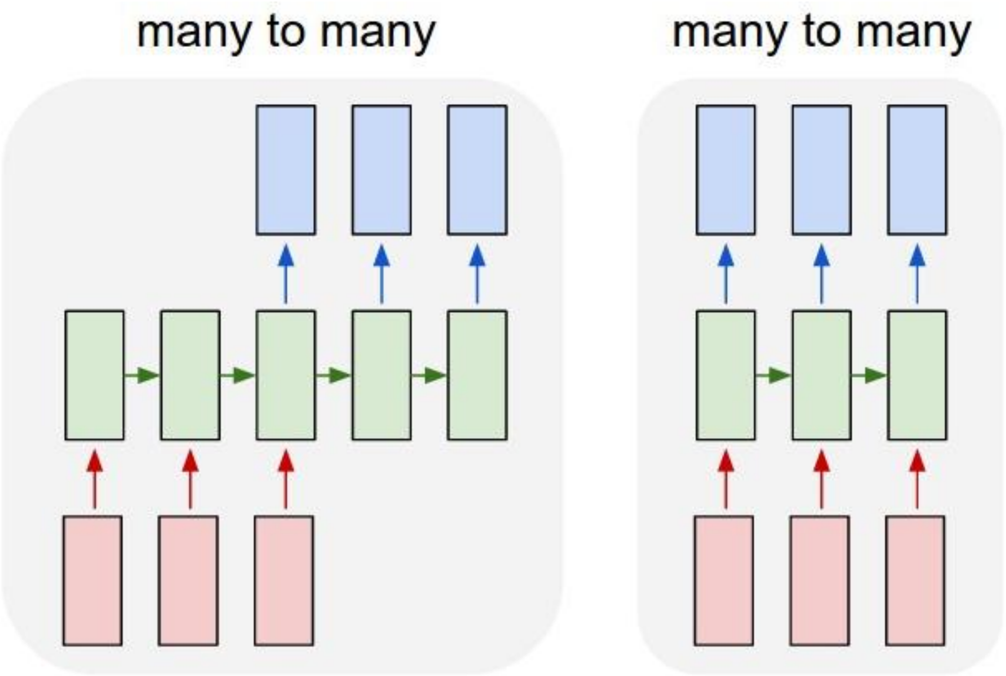




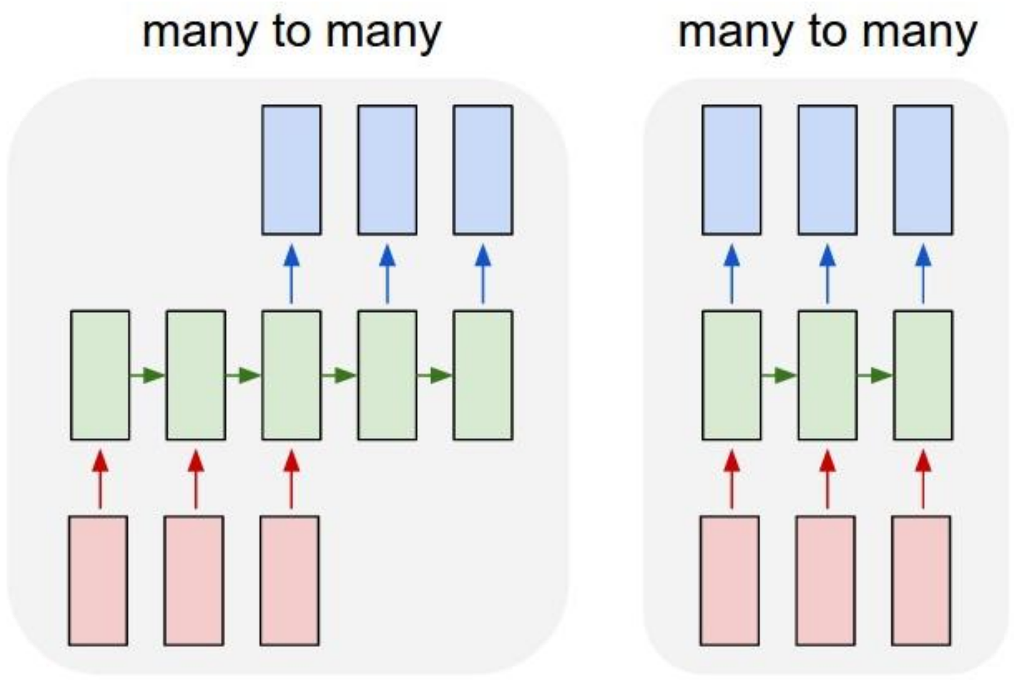


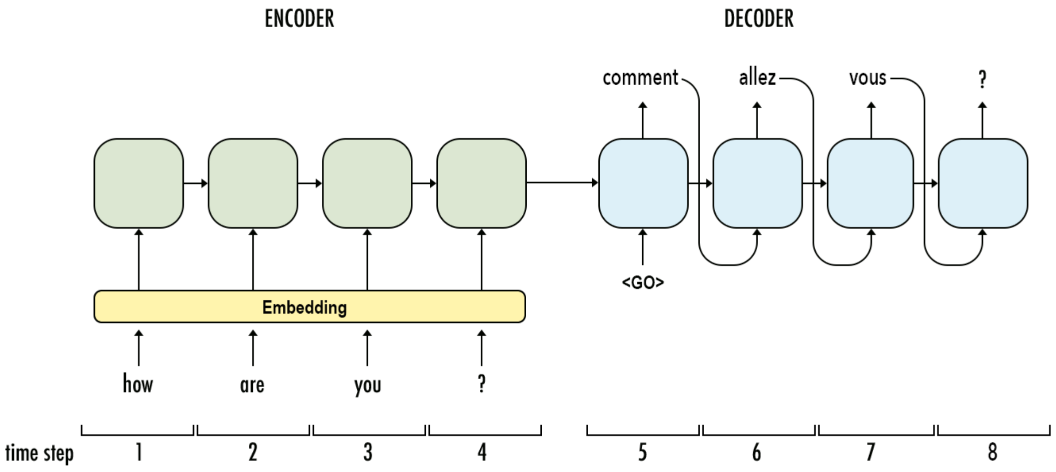
Sentiment Analysis

Many to Many

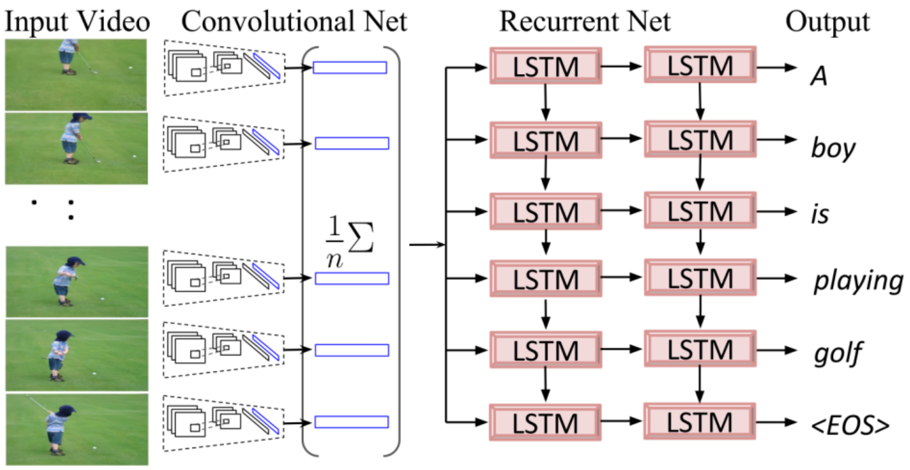


Many to Many





Machine Translation



Video Captioning

RNN IMPLEMENTATION IN PYTORCH

Character Level Text Generation using Shakespeare Dataset

Shakespeare Dataset

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
```

Full script of "**Tragedy of Coriolanus**" in .txt format

3801089 characters (including space)

66 unique characters

Prepare Data

```python
data_size_to_train = 10000

data = open('shakespeare.txt', 'r').read()[:data_size_to_train]
characters = sorted(list(set(data)))
data_size, vocab_size = len(data), len(characters)

print("Data has {} characters, {} unique".format(data_size, vocab_size))
```

Define length of training data to be used for training

Load .txt file and sort unique characters using set()

Print # of total and unique characters

```
Data has 10000 characters, 57 unique
```

```python
character_to_num = { ch:i for i,ch in enumerate(characters) }
num_to_character = { i:ch for i,ch in enumerate(characters) }
```

Create dictionaries that map each character to numbers and vice versa

```python
print(character_to_num)
```

```
{'\n': 0, ' ': 1, '!': 2, "'": 3, ',': 4, '-': 5, '.': 6, ':': 7, ';': 8, '?':
9, 'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15, 'H': 16, 'I': 17, 'J':
18, 'L': 19, 'M': 20, 'N': 21, 'O': 22, 'P': 23, 'R': 24, 'S': 25, 'T': 26,
'U': 27, 'V': 28, 'W': 29, 'Y': 30, 'a': 31, 'b': 32, 'c': 33, 'd': 34, 'e': 3
5, 'f': 36, 'g': 37, 'h': 38, 'i': 39, 'j': 40, 'k': 41, 'l': 42, 'm': 43, 'n':
44, 'o': 45, 'p': 46, 'q': 47, 'r': 48, 's': 49, 't': 50, 'u': 51, 'v': 52,
'w': 53, 'x': 54, 'y': 55, 'z': 56}
```

Prepare Data

```
data = list(data)

for i, ch in enumerate(data):
    data[i] = character_to_num[ch]
```

Convert data into Python list

Map each character in the data to a number

```
print(data[:10])
```

```
[17, 48, 57, 58, 59, 1, 14, 48, 59, 48]
```

First 10 characters of the data

Define Model

```python
class CharRNN(torch.nn.Module):

    def __init__(self, num_embeddings, embedding_dim, input_size, hidden_size, num_layers, output_size):

        super(CharRNN, self).__init__()

        self.embedding = torch.nn.Embedding(num_embeddings, embedding_dim)

        self.rnn = torch.nn.RNN(input_size=input_size, hidden_size=hidden_size,
                                num_layers=num_layers,
                                nonlinearity = 'relu')

        self.decoder = torch.nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, hidden_state):

        embedding = self.embedding(input_seq)

        output, hidden_state = self.rnn(embedding, hidden_state)

        output = self.decoder(output)

        return output, hidden_state.detach()
```

Define embedding layer

Define RNN cell

Define decoder layer

Input_seq -> embedding layer

Embedding, hidden_state -> RNN cell

RNN output -> decoder layer -> output

Return both **output** & **hidden state**

Select Hyperparameters

```python
torch.manual_seed(25)

rnn = CharRNN(num_embeddings = vocab_size, embedding_dim = 100,
              input_size = 100, hidden_size = 512, num_layers = 3,
              output_size = vocab_size)

lr = 0.001
epochs = 50
training_sequence_len = 50
validation_sequence_len = 200

loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(rnn.parameters(), lr=lr)

rnn
```

Define RNN specifics

- # of Embedding = vocab size
- Embedding dim = 100
- Input_size = 100
- Hidden_size = 512
- Num_layers = 3
- Output_size = vocab size

Define learning rate, epochs, length of training/validation text sequence

Define loss function and optimizer

Identify Tracked Values

```
train_loss_list = []
```

Python list to track training loss

Train Model

```python
data = torch.unsqueeze(torch.tensor(data), dim = 1)

# Training Loop -----------------------------------------------------------------

for epoch in range(epochs):

    character_loc = np.random.randint(100)
    iteration = 0
    hidden_state = None

    while character_loc + training_sequence_len + 1 < data_size:

        input_seq = data[character_loc : character_loc + training_sequence_len]
        target_seq = data[character_loc + 1 : character_loc + training_sequence_len + 1]

        output, hidden_state = rnn(input_seq, hidden_state)

        loss = loss_fn(torch.squeeze(output), torch.squeeze(target_seq))

        train_loss_list.append(loss.item())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        character_loc += training_sequence_len

        iteration += 1

    print("Averaged Training Loss for Epoch ", epoch,": ", np.mean(train_loss_list[-iteration:]))
```

Convert data into torch tensor in vertical orientation (data_length, 1)

For each epoch, randomly select a starting character from first 100 characters

input_seq =
character location -> training sequence size
target_seq =
character location + 1 -> training sequence size + 1

Retrieve output & hidden state from RNN cell and compute loss

Save training loss

Backpropagation in time & update network

Update the character location

Update the iteration count

Print the averaged training loss throughout an epoch

Train Model

```python
    # Sample and generate a text sequence after every epoch ------------------

    character_loc = 0
    hidden_state = None

    rand_index = np.random.randint(data_size-1)
    input_seq = data[rand_index : rand_index+1]

    print("----------------------------------------")
    with torch.no_grad():

        while character_loc < validation_sequence_len:

            output, hidden_state = rnn(input_seq, hidden_state)

            output = torch.nn.functional.softmax(torch.squeeze(output), dim=0)
            character_distribution = torch.distributions.Categorical(output)
            character_num = character_distribution.sample()

            print(num_to_character[character_num.item()], end='')

            input_seq[0][0] = character_num.item()

            character_loc += 1

    print("\n----------------------------------------")
```

Initialize character location and hidden state for validation

Pick a random character from the dataset as an initial input to RNN

Generate new output by using the previous RNN output as an input

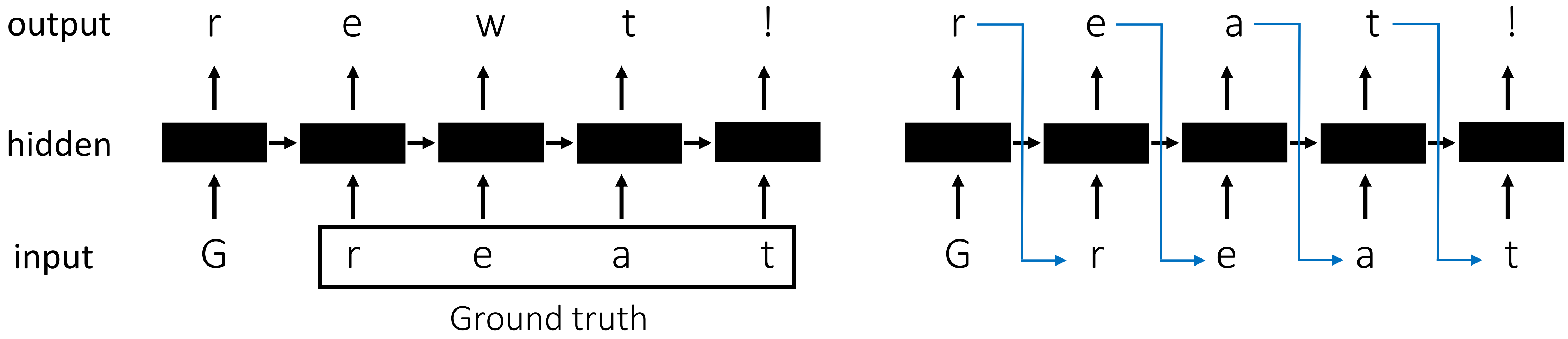
Convert the output into character number via sampling from the decoder layer output (probability distribution of characters)

Print the actual character from the number

New input_seq is the output character we just generated

Update the character location

Train Model



During Training
(Teacher Forcing)

During Validation
(RNN output is used as next input)

Train Model

```
Averaged Training Loss for Epoch  0 :  2.7497982840345365
----------------------------------------
cous zend yous leaogkal.

MENUNUNENNNNNENNENEUNNENNNNUENNNNNNUNNUNUNNNNENENENNNRNSESNNUNENNUNENN
NNNANN
Suwirs touy to
Ther'krn;

MNENUNUNN
NNNNNNINNNNNENYNNS:
NENUSNSNThat balt,
Theoud cogvrifing of
```

Generated text sequence after 1 epoch

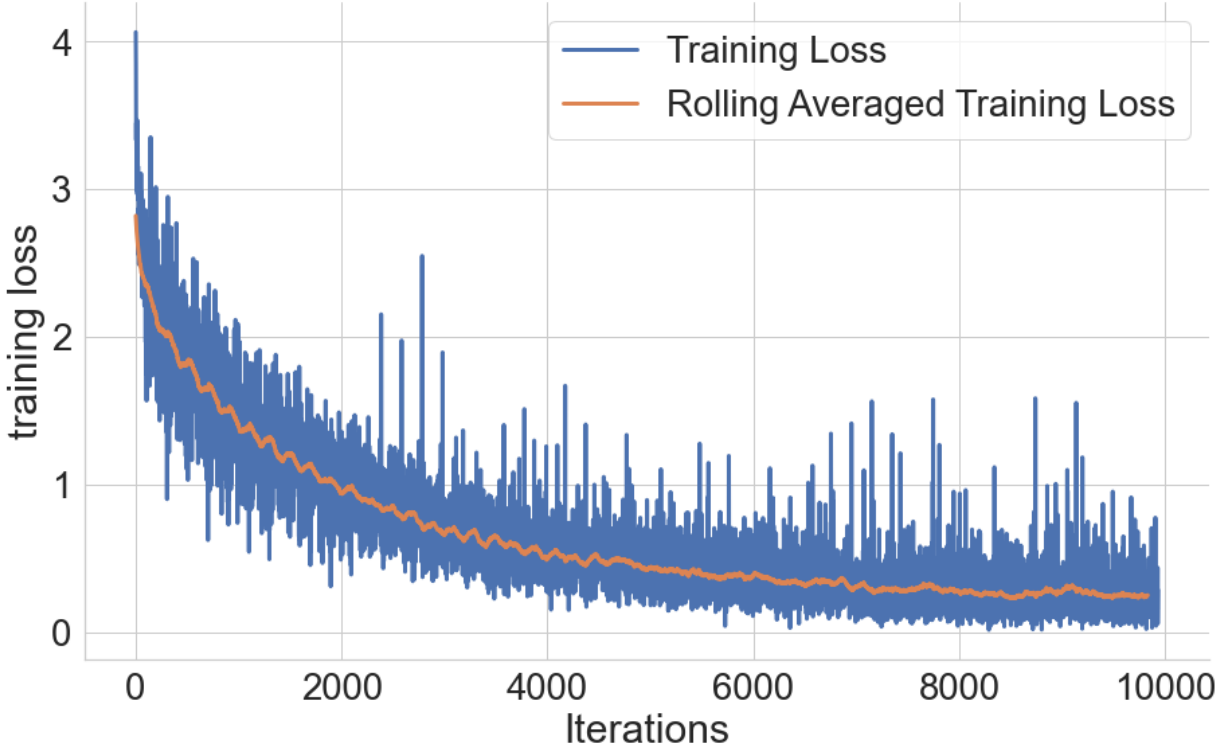
```
Averaged Training Loss for Epoch  49 :  0.26021135710741405
---------------------------------------
the Capitol; who's fide our trumpetersvile in awe, which else
Would feed on one another? What's their abundance; our
seike.
it takes, cracking ten thus--
For, look o' the moon,
Shouting their emulatio
---------------------------------------
```

Generated text sequence after 50 epoch

Validate & Evaluate Model

```
import seaborn as sns

sns.set(style = 'whitegrid', font_scale = 2.5)
```

```
plt.figure(figsize = (15, 9))

plt.plot(train_loss_list, linewidth = 3, label = 'Training Loss')
plt.plot(np.convolve(train_loss_list, np.ones(100), 'valid') / 100,
         linewidth = 3, label = 'Rolling Averaged Training Loss')
plt.ylabel("training loss")
plt.xlabel("Iterations")
plt.legend()
sns.despine()
```



Plot the training loss + rolling average training loss after training

LAB 4 ASSIGNMENT:

Create Arthur Conan Doyle AI with RNNs

Sherlock Holmes Dataset

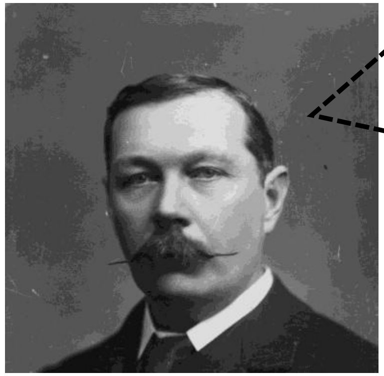
```
                              PART I

          (Being a reprint from the reminiscences of
                      John H. Watson, M.D.,
            late of the Army Medical Department.)



   CHAPTER I
   Mr. Sherlock Holmes

In the year 1878 I took my degree of Doctor of Medicine of the
University of London, and proceeded to Netley to go through the
course prescribed for surgeons in the army. Having completed my
studies there, I was duly attached to the Fifth Northumberland
Fusiliers as Assistant Surgeon. The regiment was stationed in India
at the time, and before I could join it, the second Afghan war had
broken out. On landing at Bombay, I learned that my corps had
advanced through the passes, and was already deep in the enemy's
country. I followed, however, with many other officers who were in
the same situation as myself, and succeeded in reaching Candahar in
safety, where I found my regiment, and at once entered upon my new
duties.

The campaign brought honours and promotion to many, but for me it had
nothing but misfortune and disaster. I was removed from my brigade
and attached to the Berkshires, with whom I served at the fatal
battle of Maiwand. There I was struck on the shoulder by a Jezail
bullet, which shattered the bone and grazed the subclavian artery. I
should have fallen into the hands of the murderous Ghazis had it not
been for the devotion and courage shown by Murray, my orderly, who
threw me across a pack-horse, and succeeded in bringing me safely to
the British lines.
```

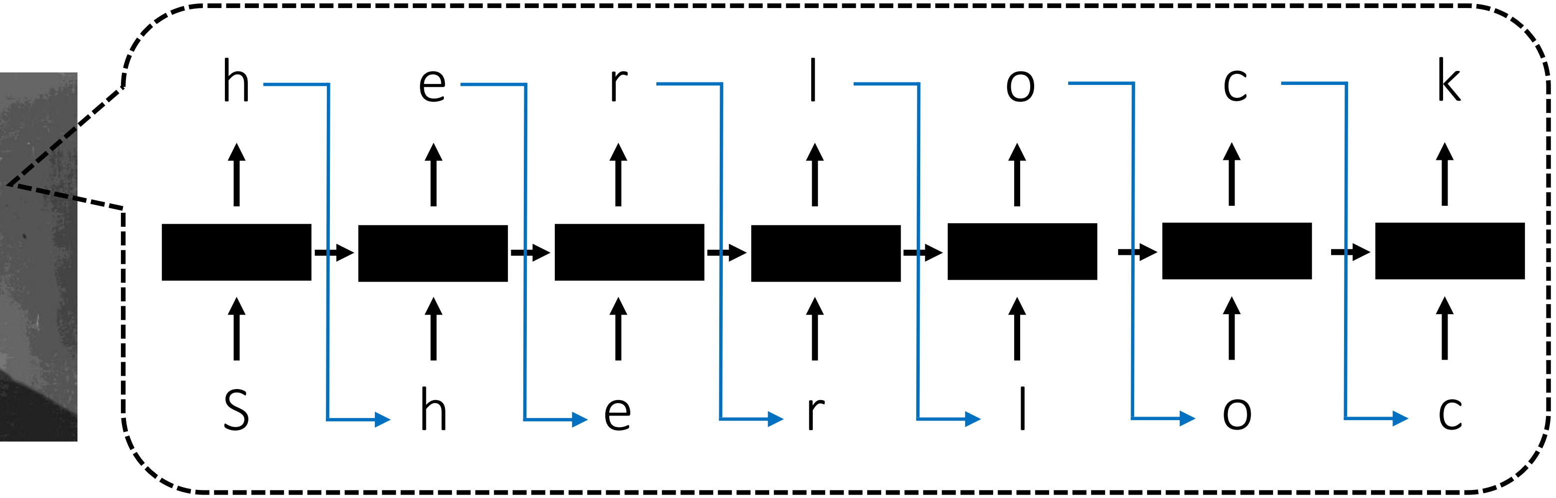
Full collection of **Sherlock Holmes series**

3011055 total characters (including space)

102 unique characters

Create Arthur Conan Doyle AI using RNN





In this exercise, you will implement **1. Vanilla RNN** and **2. GRU** to generate **Sherlock Holmes style sequence of texts.**

Prior to training, you can decide the **training size** you want to use for training.
(e.g., first 10k characters, 100k characters, etc)

For both Vanilla RNN and GRU models, design your own RNN architecture with your choice of embedding dimension, hidden state size, number of RNN layers, and training sequence size etc.

After training your RNN, **print a validation text sequence for both vanilla RNN and GRU** that most closely resembles Sherlock Holmes style in your opinion & plot the training curve to confirm the RNN successfully trained.

Describe which validation text do you like more – Vanilla RNN or GRU? What were the differences of two models during training?

Tips for Training Your RNN

First things to decide

- Training data size (# of characters)
- Embedding dimension & RNN input
- RNN hidden size
- (Vanilla RNN) Activation function (ReLU, Tanh)
- Decoder output size
- Learning rate
- Optimizer
- Number of training epochs
- Training input sequence length

Additional tips

- If you get ‘nan’ errors while training -> your training is unstable -> decrease lr or training input sequence length
- With ReLU you might be able to process longer sequence
- Choose your training data size according to your machine spec
- Higher ‘num_layers’ might give you better performance but longer training.