Introduction to Text Generation

Outline

- Problem Introduction
- Simple Examples
- Code Implementation

Problem Introduction

❖ Definition

Text generation is the process of using a computer program or algorithm to automatically create human-like text based on certain inputs. Techniques range from rule-based systems to advanced AI models like LSTMs and Transformers

INPUT

Inputs can vary from simple prompts or seed text to more complex data like keywords, structured information, or images, which the model uses to generate relevant text.

OUTPUT

The output is the coherent and contextually relevant text generated by the model, which aims to mimic human writing for various applications

Problem Introduction

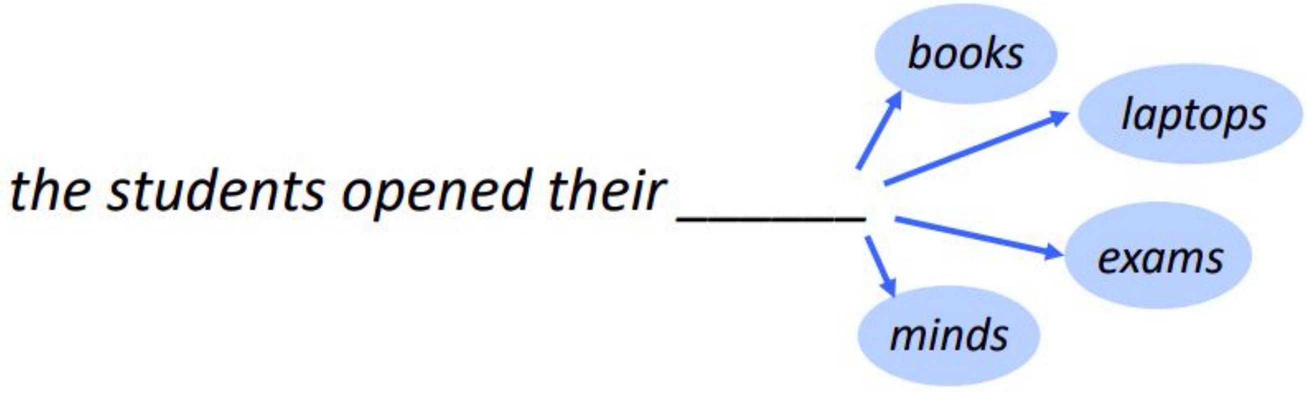
❖ Definition

The task is to predict what word comes next.

$$P(\boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

- **Input**: a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$,
- **Output**: The most probable next word $\boldsymbol{x}^{(t+1)}$

$$\boldsymbol{x}^{(t+1)} \in V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$$

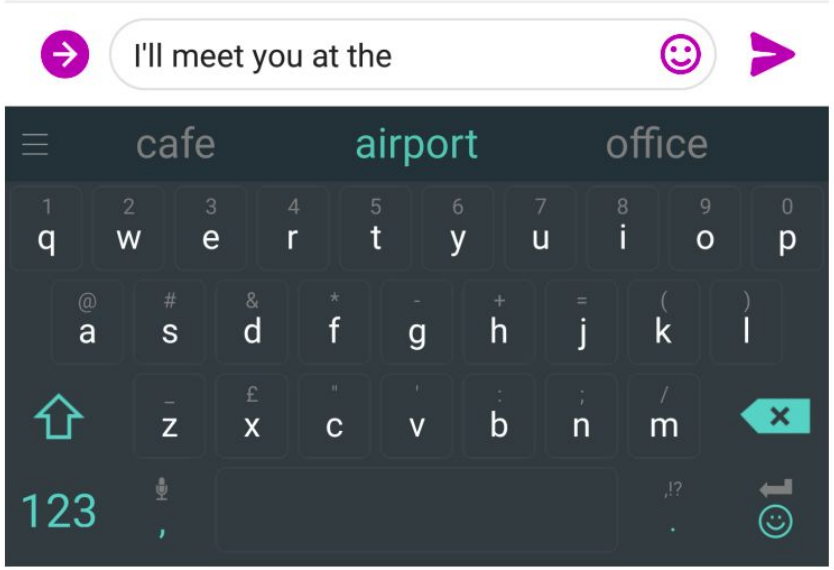


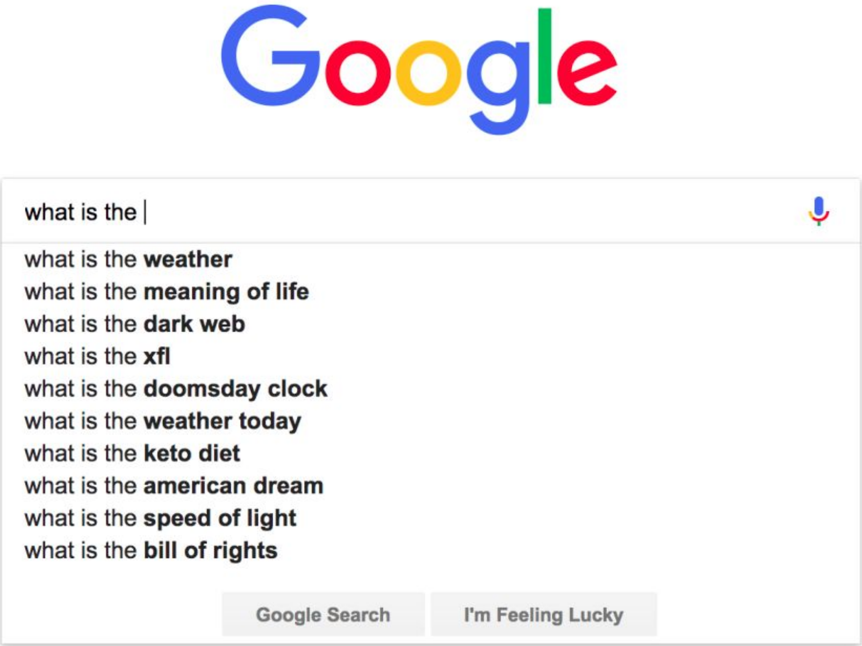
Example:

- The students opened their → books
- He was good at → basketball
- I enjoyed reading this → books
- He was a part→ of

Problem Introduction

❖ Why Text Generation?





Application Area	Description
Content Creation	Generating articles, blog posts, stories, and poetry to assist in creative writing and content development.
Customer Service Automation	Powering chatbots and virtual assistants to provide instant responses to customer inquiries, improving service efficiency.
Language Translation	Providing rough translations between languages, useful for less commonly spoken languages with scarce resources.
Programming Assistance	Generating code snippets, explaining code, and offering debugging suggestions to speed up software development.
Educational Tools	Creating educational content like summaries, quizzes, and explanatory notes tailored to students' understanding levels.
Email and Communication Assistance	Assisting in drafting emails, reports, and presentations with appropriate tone and style for the audience.

Problem Introduction

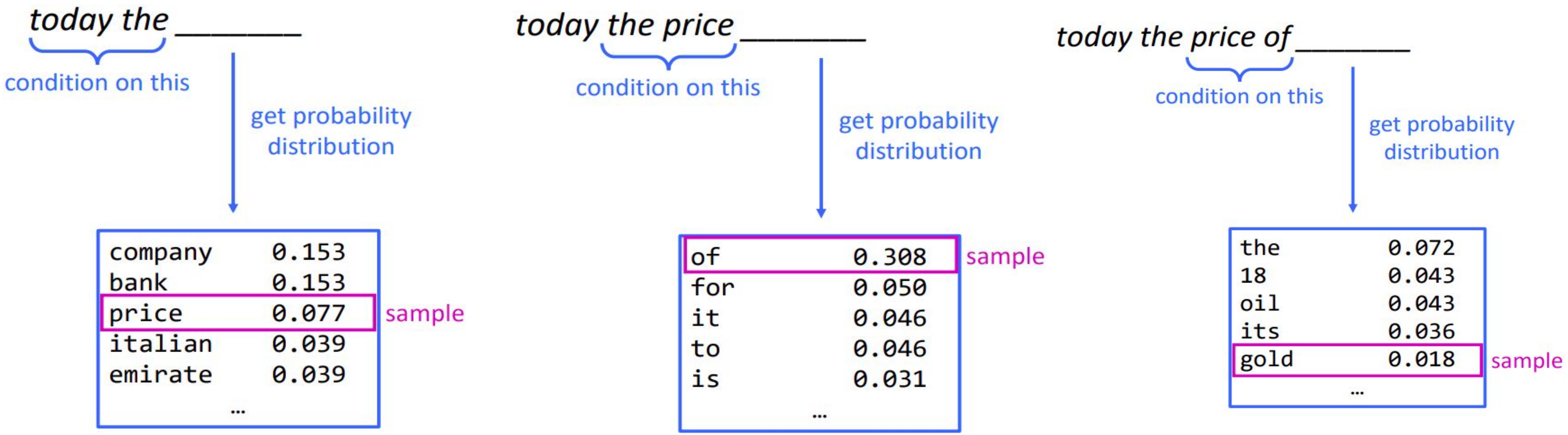
❖ n-gram

- **Definition**: A **n-gram** is a chunk of **n** consecutive words
  - unigrams: “the”, “students”, “opened”, ”their”
  - bigrams: “the students”, “students opened”, “opened their”
  - trigrams: “the students opened”, “students opened their”
  - 4-grams: “the students opened their”

- **Idea**: Collect statistics about how frequent different n-grams are, and use these to predict next word.

Problem Introduction

❖ n-gram



Simple Examples

❖ Text Generation Model

output distribution

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

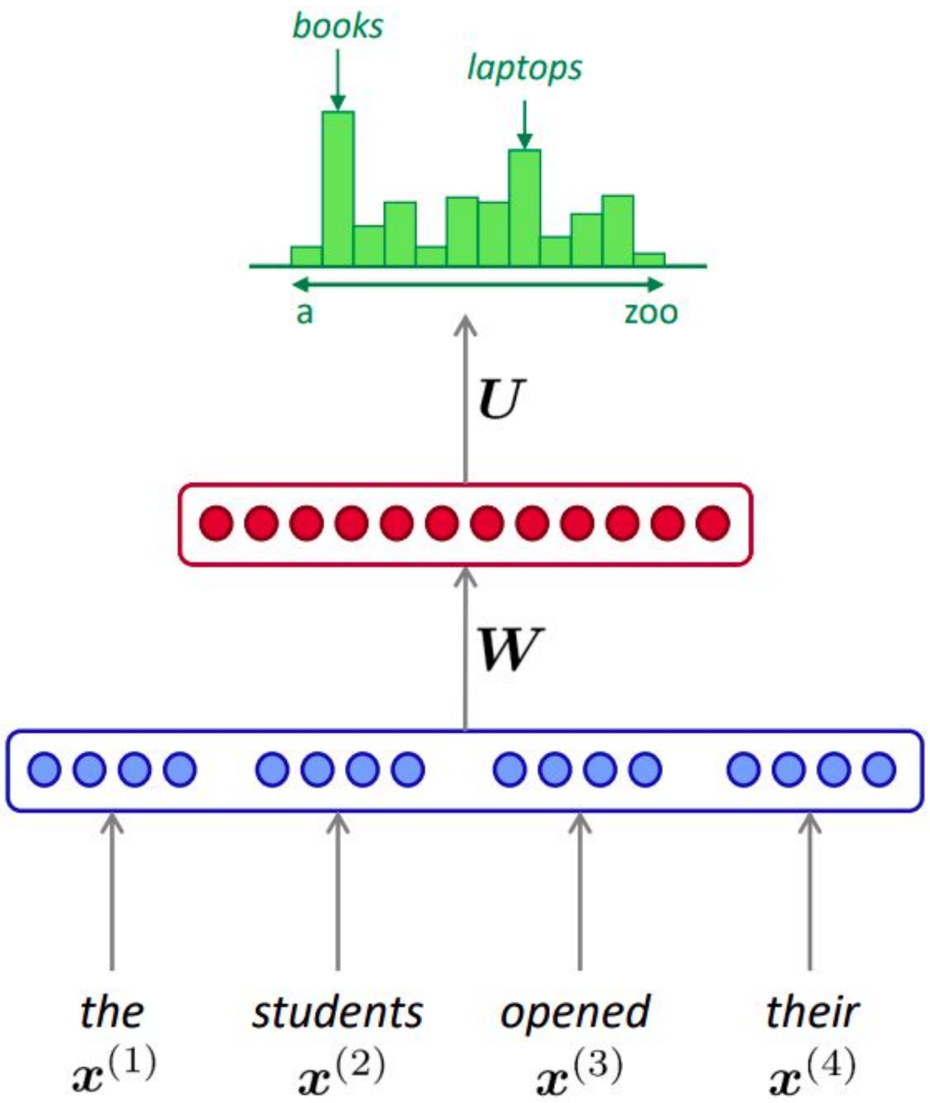
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + \boldsymbol{b}_1)$$

concatenated word embeddings

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

words / one-hot vectors

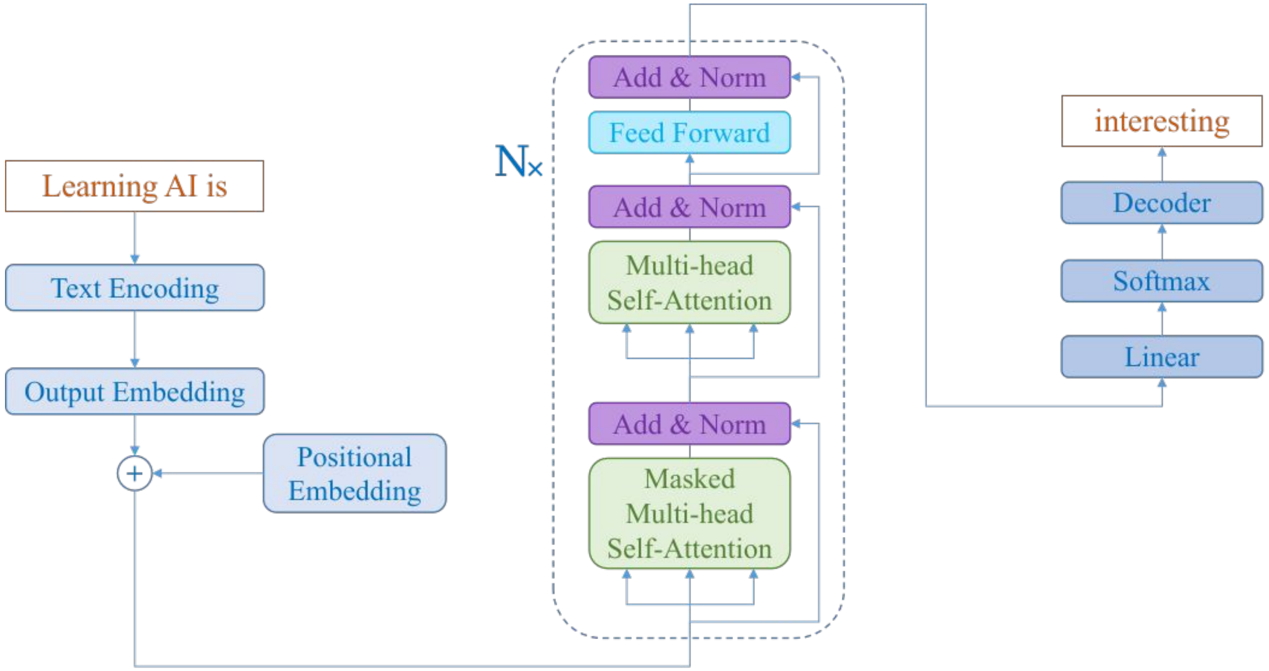
$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



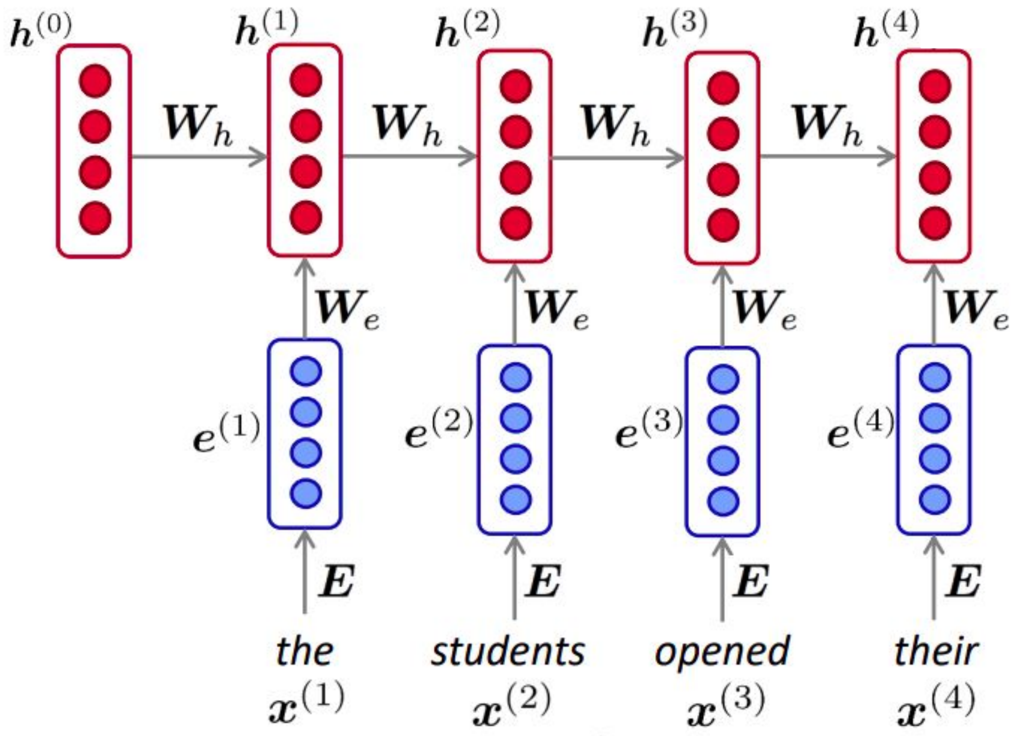
Simple Examples

❖ Text Generation Model

Transformer



RNN



“Both the brown fox and the brown dog slept.”

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown] }

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown],
(the, brown): [fox], }

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown],
(the, brown): [fox],
(brown,fox): [and], }

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown],
(the, brown): [fox],
(brown,fox): [and],
(fox, and): [the], }

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown],
(the, brown): [fox],
(brown,fox): [and],
(fox, and): [the],
(and, the): [brown],}

“Both the brown fox and the brown dog slept.”

{ (Both, the): [brown],
(the, brown): [fox, dog],
(brown,fox): [and],
(fox, and): [the],
(and, the): [brown],}

...

Simple Examples

❖ Evaluation Metric

- **Precision and Recall of Words**

Predict/Candidate/Output: Tôi học NLP của AI VIET NAM

Reference: Tôi đang học lớp AI của AI VIET NAM

Precision 1-gram

$$\frac{\text{correct}}{\text{candidate\_length}} = \frac{6}{7}$$

Recall 1-gram

$$\frac{\text{correct}}{\text{reference\_length}} = \frac{6}{9}$$

F1-score 1-gram

$$\frac{\text{precision} \times \text{recall}}{(\text{precision} + \text{recall})/2} = 0.75$$

Simple Examples

❖ Evaluation Metric

- **BLEU Score**

Precision 1-gram

$$\frac{\text{correct}}{\text{candidate\_length}} = \frac{6}{7}$$

Recall 1-gram

$$\frac{\text{correct}}{\text{reference\_length}} = \frac{6}{9}$$

N-gram overlap between machine translation candidate and reference translation

Compute precision for n-grams of size 1 to 4

With 4-gram and add brevity penalty (for too short translations):

$$\text{BLEU} = \min\left(1, \frac{\text{candidate\_length}}{\text{reference\_length}}\right)\left(\prod_{i=1}^{4} \text{Precision}_i\right)^{1/4}$$

Simple Examples

❖ Evaluation Metric

- **Precision and Recall of Words**

Predict/Candidate/Output: Tôi học NLP của AI VIET NAM

Reference: Tôi đang học lớp CV và NLP của AI

Precision	1-gram	2-gram	3-gram	4-gram
	6/7	3/6	2/5	1/4

Multiple reference: N-grams may match in any of the reference and closest reference length used

Brevity penalty = 7/9

BLEU = 0.35

$$\text{BLEU} = \min\left(1, \frac{\text{candidate\_length}}{\text{reference\_length}}\right)\left(\prod_{i=1}^{4} \text{Precision}_i\right)^{1/4}$$

Code Implementation

❖ Text Generation Code Overview

- **0: Import Library**
- **1: Data Preparation**
  1. Load Raw Data
  2. Split Data
  3. Tokenizer
  4. Pad and Truncate
- **2: Modeling**
  1. Config Model
  2. Build Model
- **3: Train and Evaluate**
  1. Train model
  2. Evaluate model
- **4: Plot Results**
  1. Train/Test Loss
  2. Train/Test Acc

Code Implementation

❖ Text Generation Code Overview

0: Import Library → 1: Data Preparation → 2: Modeling → 3: Train and Evaluate → 4: Plot Results

1: Data Preparation
1. Load Raw Data
2. ~~Split Data~~
3. Tokenizer
4. Pad and Truncate

2: Modeling
1. Config Model
2. Build Model

3: Train and Evaluate
1. Train model
2. Evaluate model

4: Plot Results
1. Train/Test Loss
2. ~~Train/Test Acc~~

Code Implementation

❖ Import Library

`import torch`: Imports PyTorch, a library for tensor computation and deep learning.

`import torch.nn as nn`: Brings in PyTorch's neural network module, aliased as `nn`, for building network layers.

```
1 import torch
2 import torch.nn as nn
3 from torchtext.data.utils import get_tokenizer
4 from torchtext.vocab import build_vocab_from_iterator
5 import matplotlib.pyplot as plt
```

`from torchtext.data.utils import get_tokenizer`: This function is used to split text into tokens (e.g., words or subwords), which is a common preprocessing step in NLP.

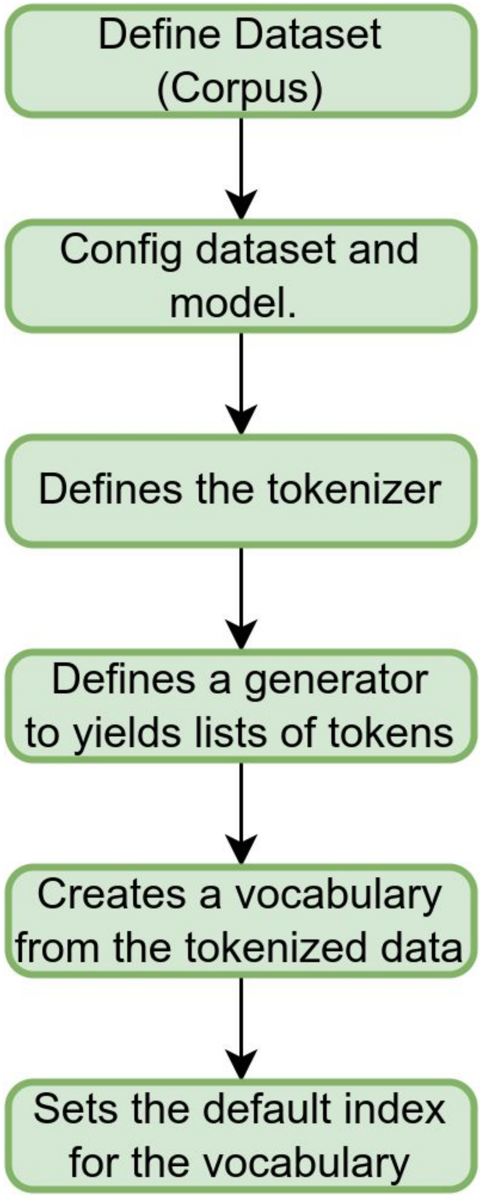
`import matplotlib.pyplot as plt`: is a library for creating static, interactive, and animated visualizations in Python

`from torchtext.vocab import build_vocab_from_iterator`: This function is used to build a vocabulary from an iterator. The vocabulary maps tokens (words) to indices and is used to convert text data into numerical form that can be processed by neural networks.

Code Implementation

❖ Data Preparation

```
SET corpus TO [
    "ăn quả nhớ kẻ trồng cây",
    "có chí thì nên"
]


SET data_size TO len(corpus)
SET vocab_size TO 12
SET sequence_length TO 5


SET tokenizer TO get_tokenizer('basic_english')


DEFINE FUNCTION yield_tokens(examples):
    FOR text IN examples:
        yield tokenizer(text)



SET vocab TO build_vocab_from_iterator(yield_tokens(corpus),
                                      max_tokens=vocab_size,
                                      specials=["<unk>", "<pad>"]
                                     )


vocab.set_default_index(vocab["<unk>"])
vocab.get_stoi()
```



Code Implementation

Data Preparation

```
SET corpus TO [
    "ăn quả nhớ kẻ trồng cây",
    "có chí thì nên"
]


SET data_size TO len(corpus)
SET vocab_size TO 12
SET sequence_length TO 5


SET tokenizer TO get_tokenizer('basic_english')


DEFINE FUNCTION yield_tokens(examples):
    FOR text IN examples:
        yield tokenizer(text)



SET vocab TO build_vocab_from_iterator(yield_tokens(corpus),
                                      max_tokens=vocab_size,
                                      specials=["<unk>", "<pad>"]
                                     )


vocab.set_default_index(vocab["<unk>"])
vocab.get_stoi()
```

Defines a list named `corpus` containing two phrases. This serves as the dataset for this example.

Configuration for dataset and modeling.
- Calculates the size of the corpus by counting the number of items in the list and stores it in `data_size`
- Sets the maximum size of the vocabulary to 12. This means only the first 12 unique tokens will be considered in the vocabulary.
- Defines the sequence length as 5, meaning that inputs/outputs will be processed or generated in chunks of 5 tokens at a time.

Defines the tokenizer: This tokenizer splits text into words, lowercasing, and removing punctuation.

Defines a generator function `yield_tokens` that takes an iterable of text examples (such as sentences) and yields lists of tokens for each example.

Code Implementation

❖ Data Preparation

```
SET corpus TO [
    "ăn quả nhớ kẻ trồng cây",
    "có chí thì nên"
]


SET data_size TO len(corpus)
SET vocab_size TO 12
SET sequence_length TO 5


SET tokenizer TO get_tokenizer('basic_english')


DEFINE FUNCTION yield_tokens(examples):
    FOR text IN examples:
        yield tokenizer(text)


SET vocab TO build_vocab_from_iterator(yield_tokens(corpus),
                                  max_tokens=vocab_size,
                                  specials=["<unk>", "<pad>"]
                                 )


vocab.set_default_index(vocab["<unk>"])
vocab.get_stoi()
```

Creates a vocabulary from the tokenized data and then configures the vocabulary to handle unknown tokens.
It limits the vocabulary to vocab_size most frequent tokens and includes special tokens <unk> for unknown words and <pad> for padding.

- Sets the default index for the vocabulary to be the index of the <unk> token (any token not found in the vocabulary will be treated as an unknown token)

- Retrieves the mapping of tokens to indices from the vocabulary and prints it.

Code Implementation

❖ Data Preparation

```python
corpus = [
    "ăn quả nhớ kẻ trồng cây",
    "có chí thì nên"
]
```

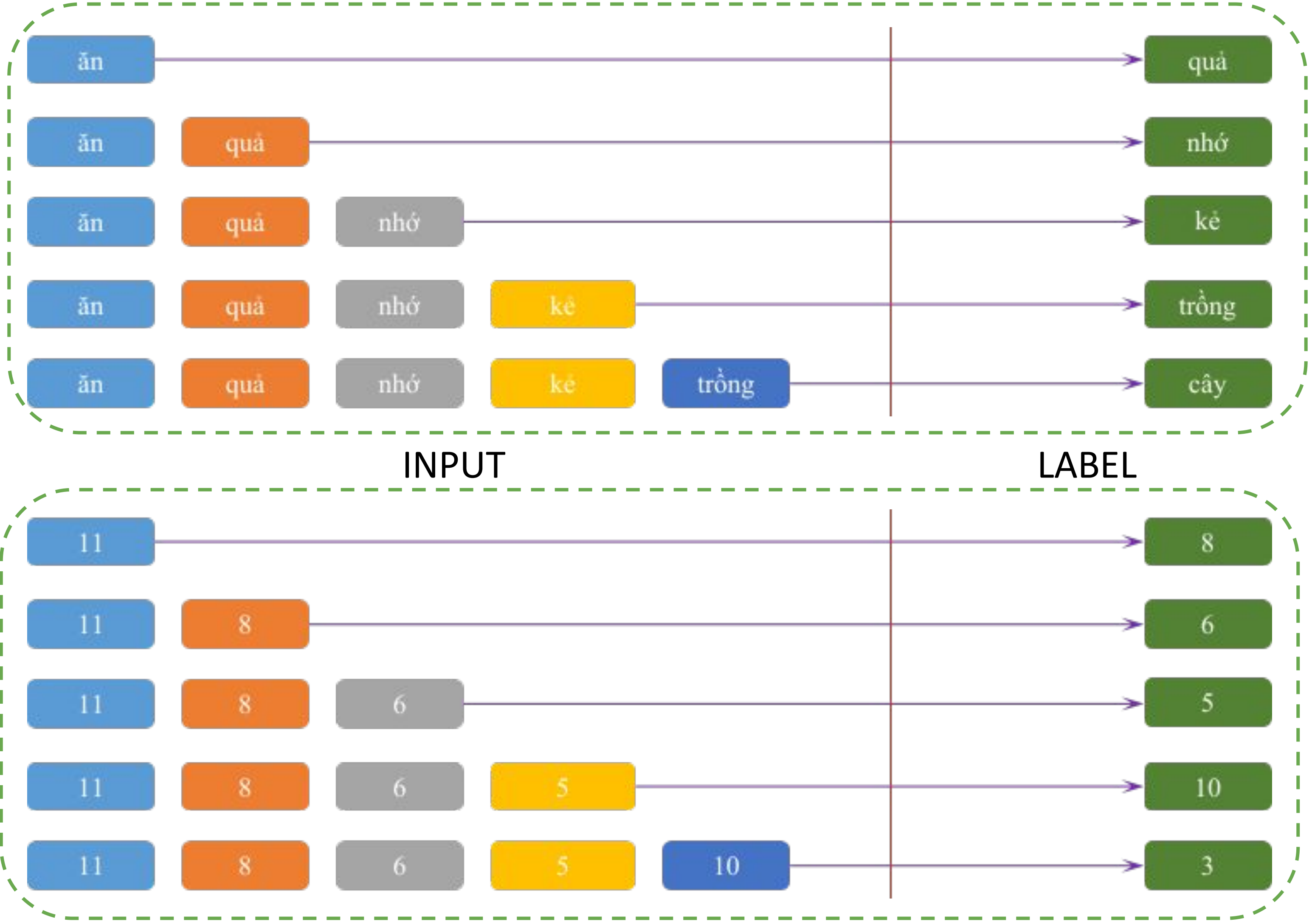
Vocab

```python
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

❖ Text Generation Example

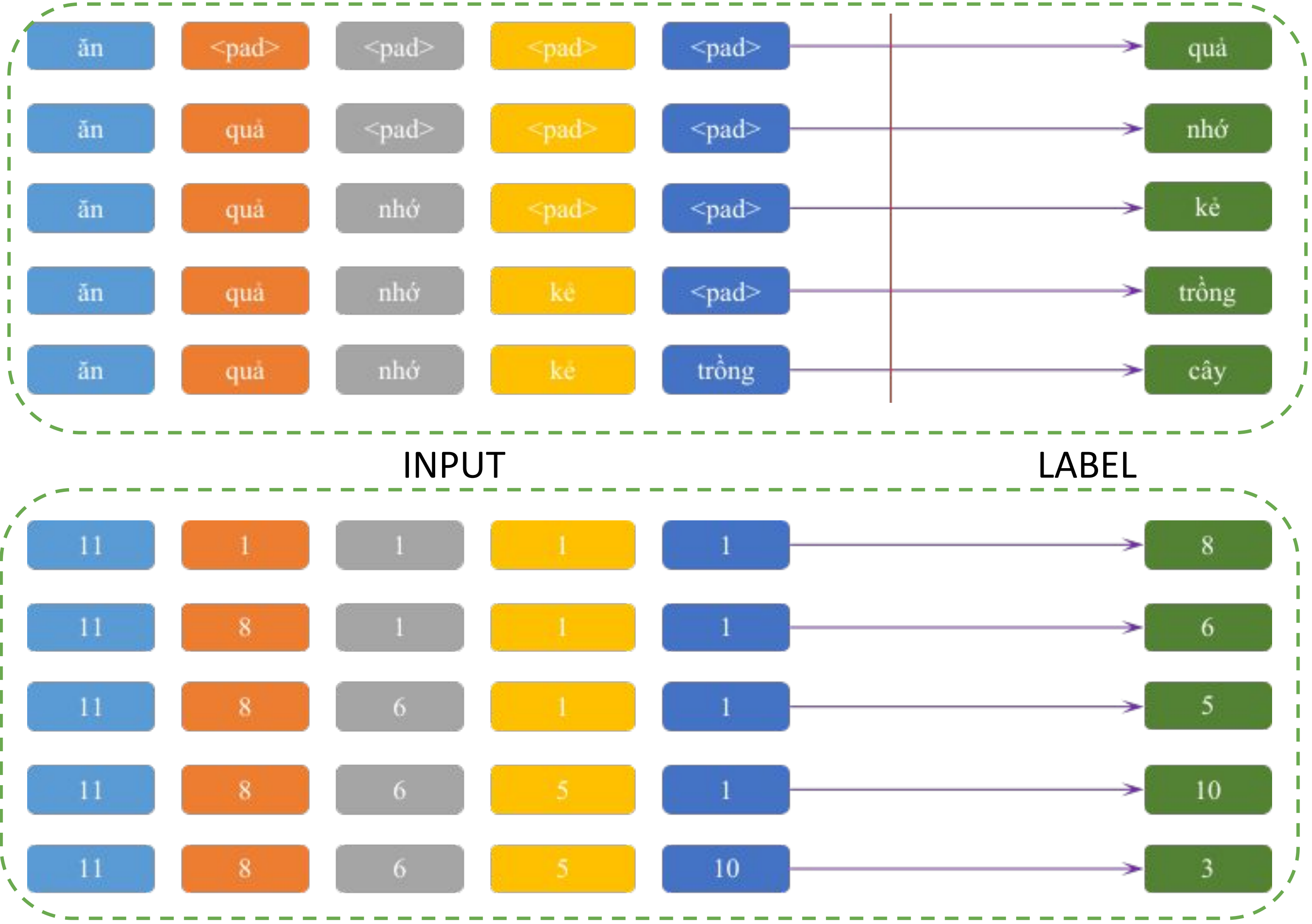
ăn quả nhớ kẻ trồng cây




```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

❖ Text Generation Example



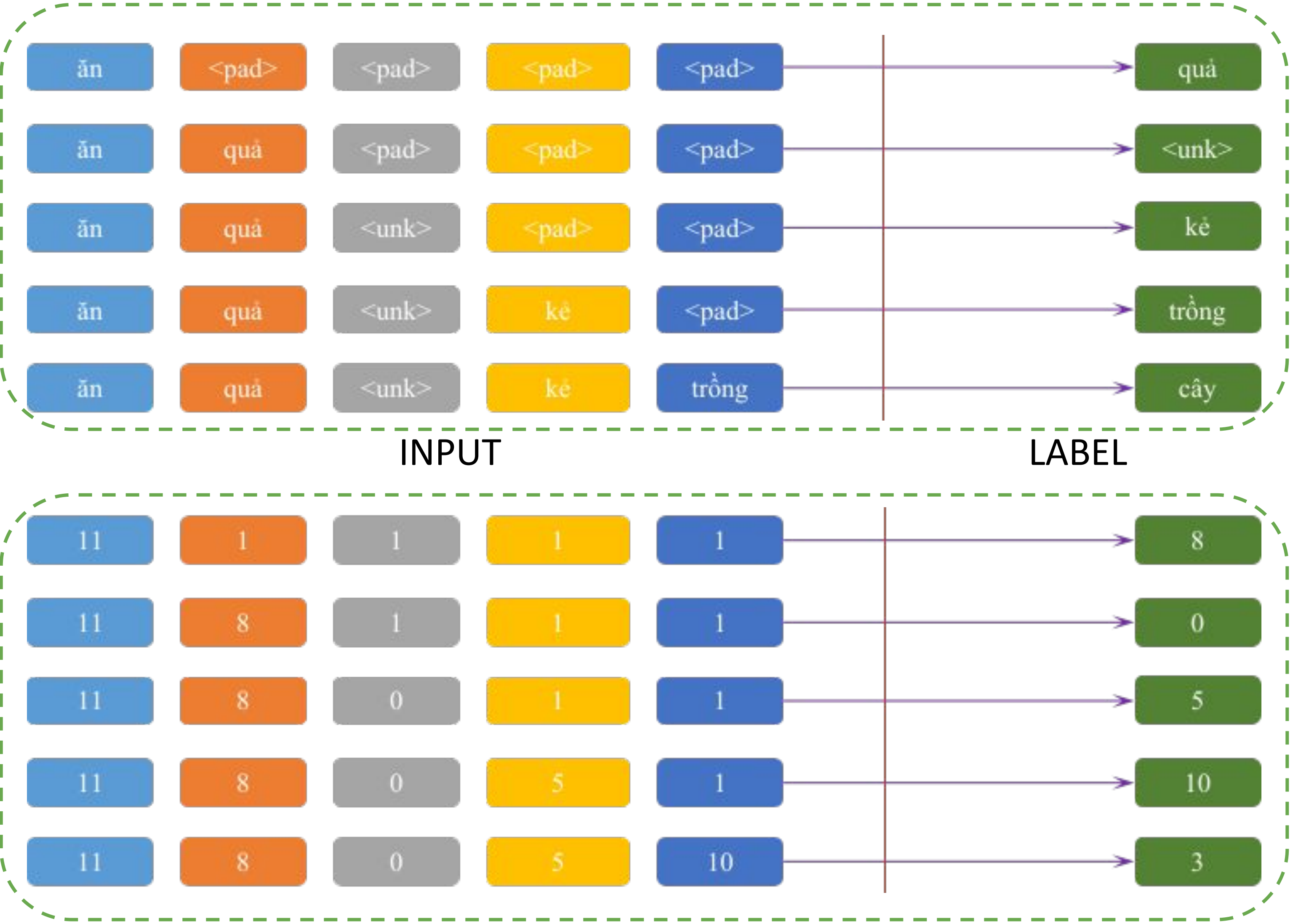
ăn quả nhớ kẻ trồng cây

```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

❖ Text Generation Example

ăn quả quên kẻ trồng cây

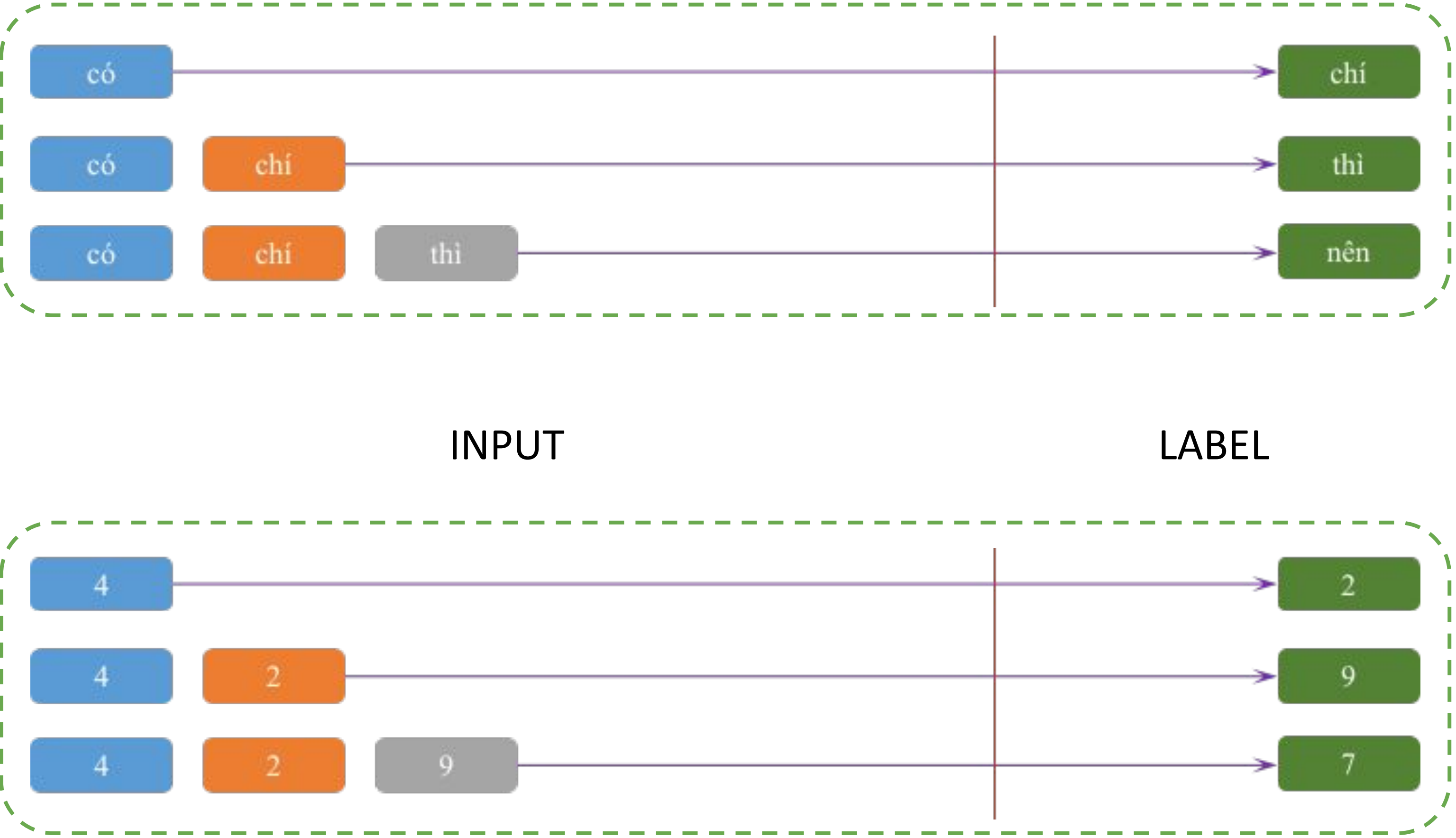



```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

❖ Text Generation Example

có chí thì nên

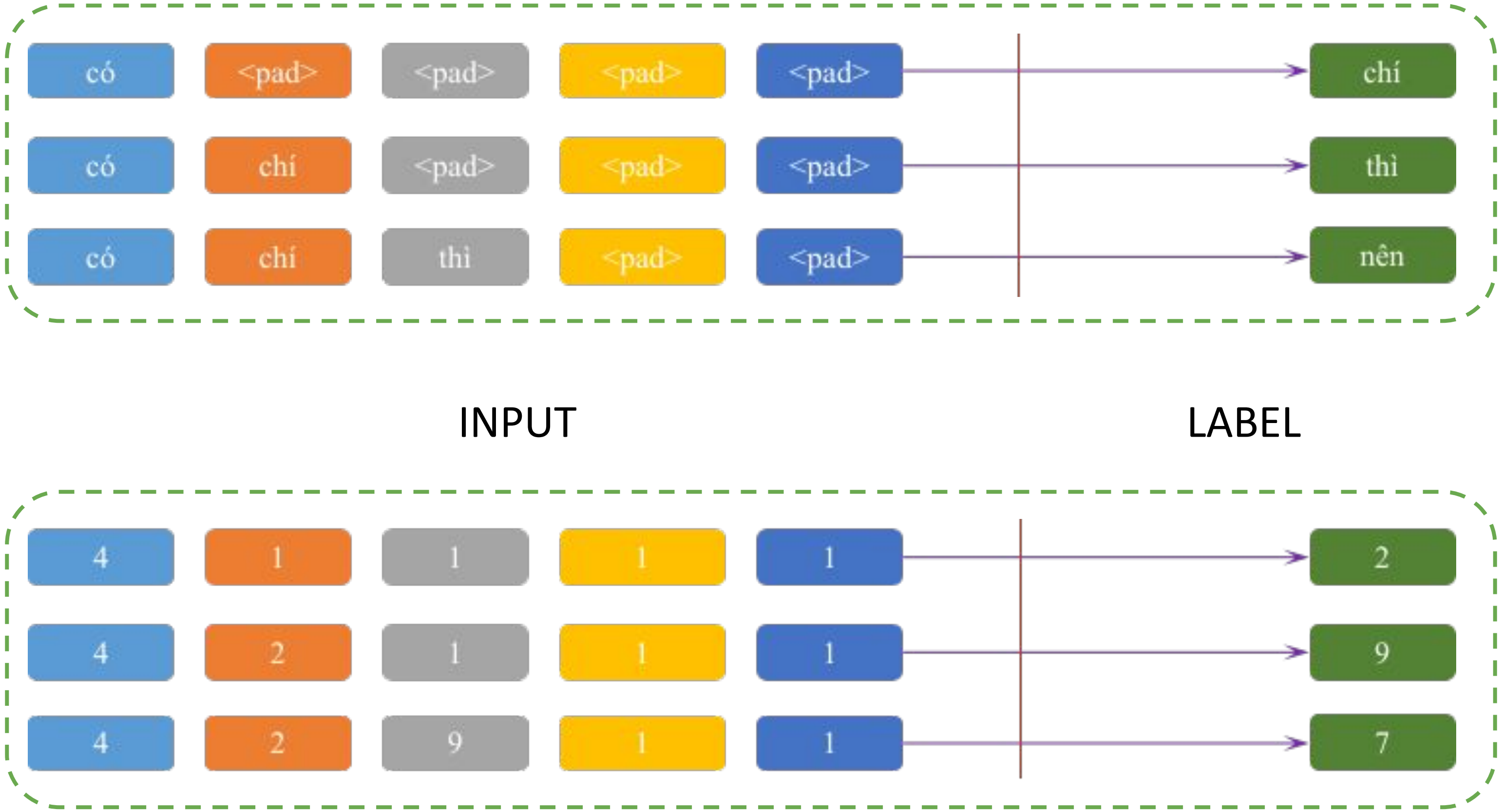



```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

❖ Text Generation Example

có chí thì nên



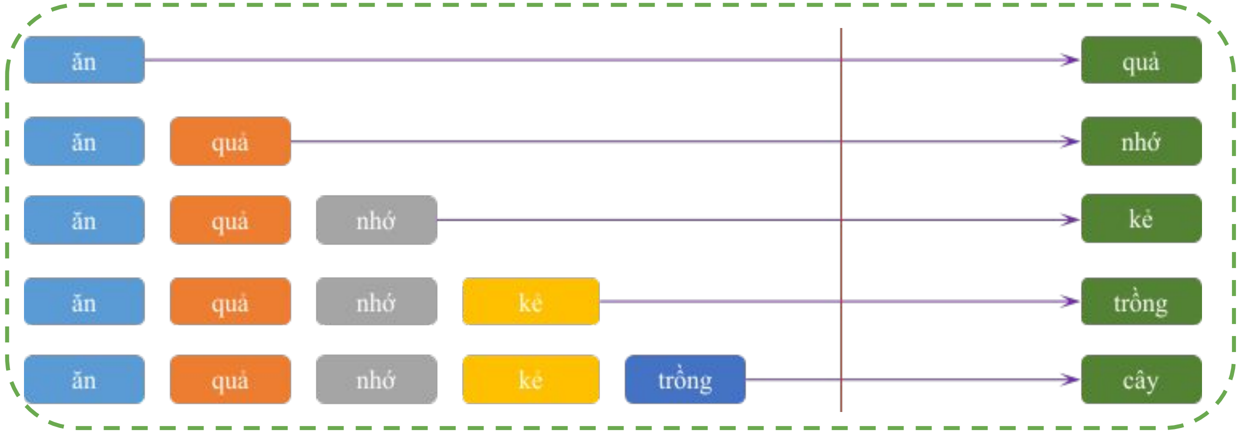

```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Code Implementation

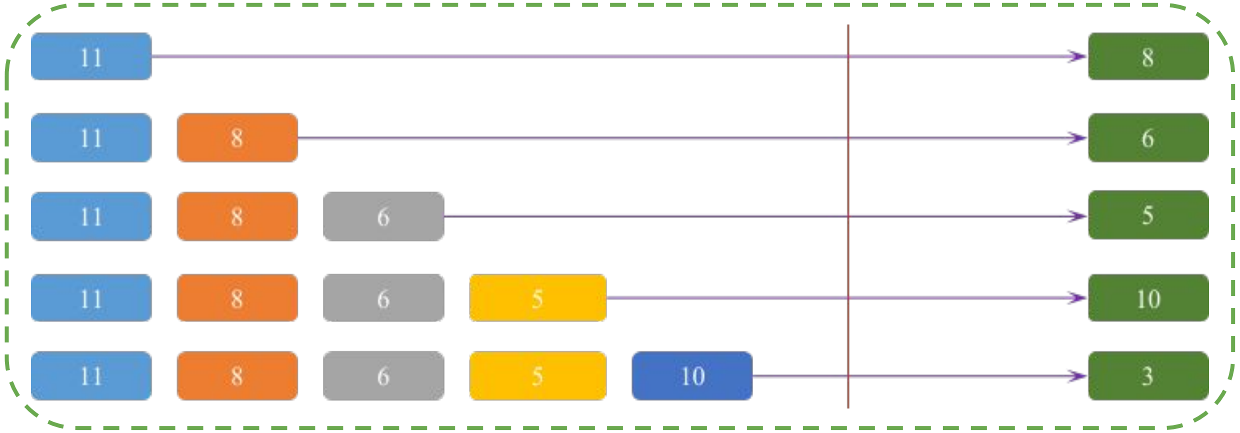
❖ Text Generation Example

ăn quả nhớ kẻ trồng cây

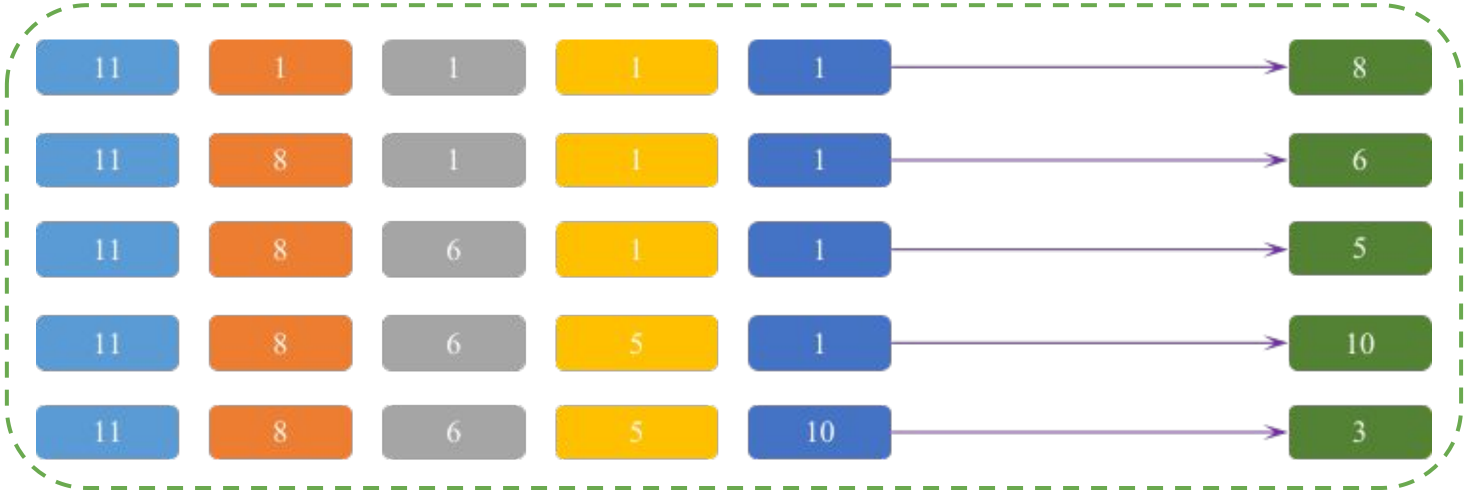
Step1: Build Training Data



Step2: Convert to ID



Step3: Pad and Truncate



Code Implementation

❖ Data Preparation

```
SET data_x TO []
SET data_y TO []

FOR vector IN corpus:
    SET vector TO vector.split()
    SET x TO vector[:-1]
    SET y TO vector[1:]

    FOR i IN range(len(x)):
        data_x.append(x[:i+1])
        data_y.append(y[i])


DEFINE FUNCTION vectorize(x, y, vocab, sequence_length):
    SET x_ids TO [vocab[token] FOR token IN x][:sequence_length]
    SET x_ids TO x_ids + [vocab["<pad>"]] * (sequence_length - len(x))
    RETURN x_ids, vocab[y]


SET data_x_ids TO []
SET data_y_ids TO []

FOR x, y IN zip(data_x, data_y):
    SET x_ids, y_ids TO vectorize(x, y, vocab, sequence_length)
    data_x_ids.append(x_ids)
    data_y_ids.append(y_ids)
```

Step1: Build Training Data

- Processes the corpus to create training data (data_x and data_y) for a sequence prediction task
- Each sentence in the corpus is split into words
- For each prefix of words in a sentence, it creates an input sequence (data_x) and the corresponding next word in the sequence (data_y)

Step2 & 3: Convert to ID & Pad and Truncate

- Defines a function vectorize to convert input sequences (x) and target words (y) into their numerical representations using a given vocabulary (vocab)
- All input sequences are padded or truncated to a specified length (sequence_length)

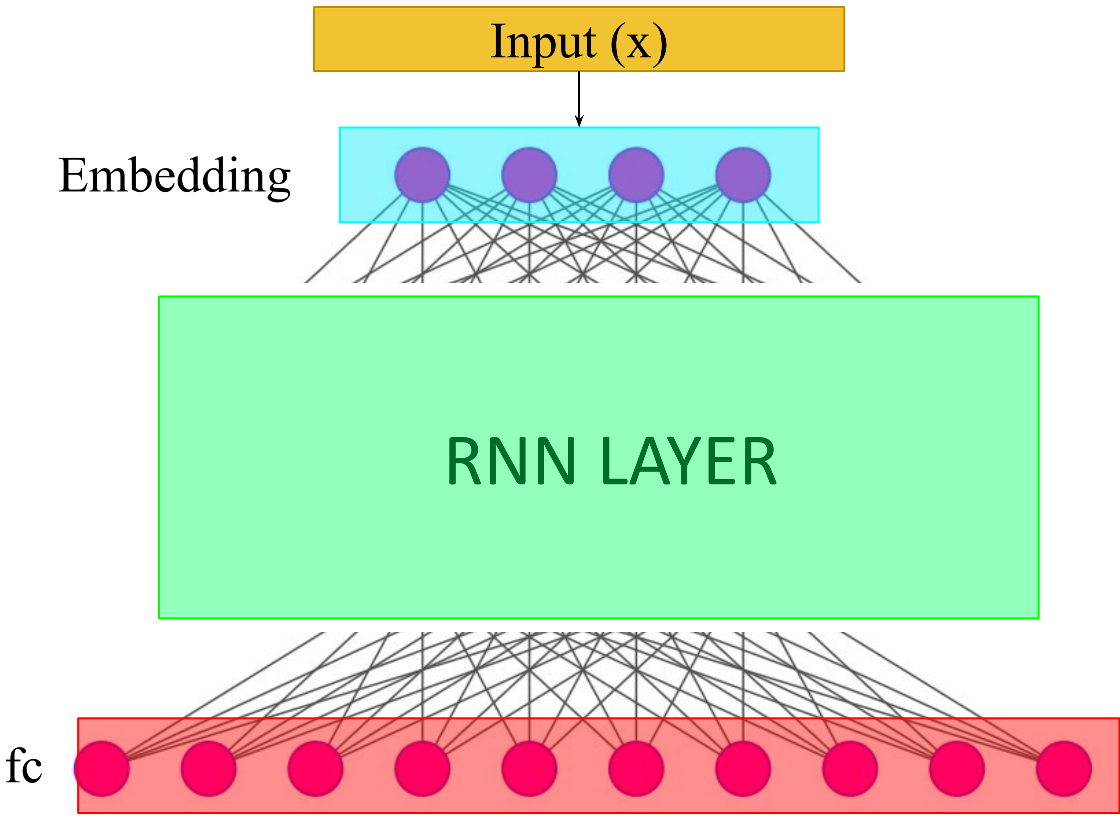
Applies this function to each pair of input sequence and target word in data_x and data_y, creating lists of numericalized inputs (data_x_ids) and targets (data_y_ids).

Code Implementation

❖ Text Generation Model

```
SET data_size TO len(corpus)
SET vocab_size TO 12
SET sequence_length TO 5
```

```python
class Text_Generation_Model(nn.Module):
    def __init__(self, vocab_size, sequence_length):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, 4)
        self.recurrent = nn.RNN(4, 4, batch_first=True)
        self.linear = nn.Linear(sequence_length*4, vocab_size)

    def forward(self, x):
        x = self.embedding(x)
        x,_ = self.recurrent(x)
        x = nn.Flatten()(x)
        x = self.linear(x)
        return x

model = Text_Generation_Model(vocab_size, sequence_length)
```



Code Implementation

❖ Text Generation Model

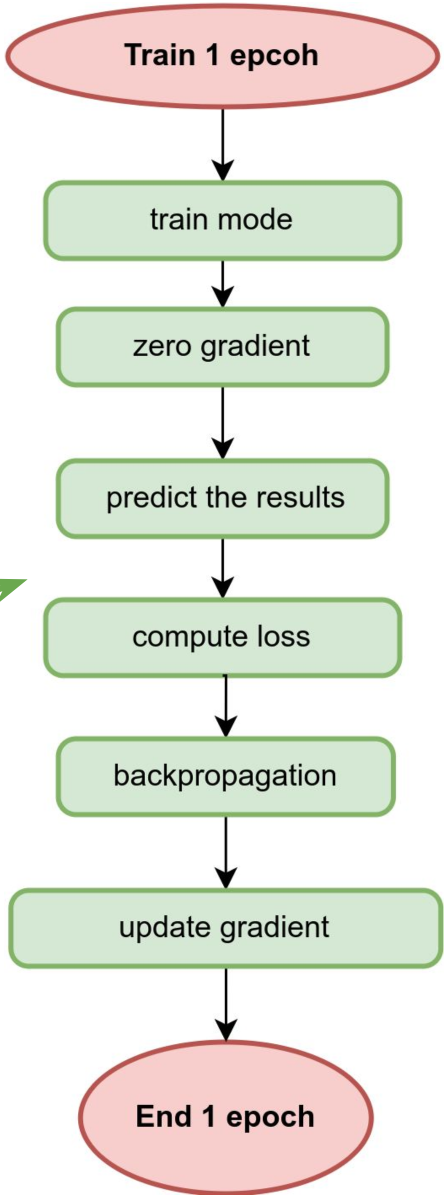
```python
from torchinfo import  summary
input_data = torch.randint(low=0, high=vocab_size-1, size=(8, sequence_length))
summary(model, input_data = input_data)
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Text_Generation_Model                    [8, 12]                   --
├─Embedding: 1-1                         [8, 5, 4]                 48
├─RNN: 1-2                               [8, 5, 4]                 40
├─Linear: 1-3                            [8, 12]                   252
==========================================================================================
Total params: 340
Trainable params: 340
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.01
==========================================================================================
```

Code Implementation

❖ Compile and Train

```
SET criterion TO nn.CrossEntropyLoss()
SET optimizer TO torch.optim.Adam(model.parameters(), lr=0.05)


SET EPOCH TO 35
SET losses TO []

FOR i IN range(EPOCH):
    model.train()

    optimizer.zero_grad()

    SET outputs TO model(data_x_ids)

    SET loss TO criterion(outputs, data_y_ids)
    losses.append(loss.item())
    OUTPUT(losses[i])

    loss.backward()

    optimizer.step()
```



Code Implementation

❖ Inference

```
{'ăn': 11,
 'thì': 9,
 'nhớ': 6,
 'kẻ': 5,
 'trồng': 10,
 'quả': 8,
 'cây': 3,
 'chí': 2,
 '<pad>': 1,
 'nên': 7,
 'có': 4,
 '<unk>': 0}
```

Processing a prompt example
>> OUTPUT: [4, 2, 1, 1, 1]

```
SET promt TO 'có chí'
SET promt TO promt.split()
SET promt_ids TO [vocab[token] FOR token IN promt][:sequence_length]
SET promt_ids TO promt_ids + [vocab["<pad>"]] * (sequence_length - len(promt))


FOR i IN range(sequence_length - len(promt)):
    SET promt_tensor TO torch.tensor(promt_ids, dtype=torch.long).reshape(1, -1)
    SET outputs TO model(promt_tensor)
    SET next_id TO torch.argmax(outputs, axis=-1)
    SET promt_ids[len(promt)+i] TO next_id.item()
```

Inference
>> OUTPUT: [4, 2, 9, 1, 1]
>> OUTPUT: [4, 2, 9, 7, 1]
>> OUTPUT: [4, 2, 9, 7, 10]

Code Implementation

❖ Plot Results

```
plt.plot(losses, label='Train')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```