Introductory Seminar of PyTorch for Deep Learning

Daniele Angioni, Cagliari Digital Lab 2024 - Day 2

Sequential Data

Working with sequences

Initially, we treated a set of samples as a matrix with shape (n_samples, n_features)

In computer vision instead we treat images as a 3-dimensional vectors, so a set of images in a tensor notation have shape (n_samples, n_channels, width, depth).

However, many tasks deal instead with **sequential data** as input, as output or both

With sequential data, the **output** should not depend only on one input , but it **depends on all the inputs seen before**.

The text is one of the most known types of sequential data, but we can find similar situations with audio, videos or numerical data that varies with time (e.g. in weather forecasting).

Example

Take for example the task of predicting the next word: if we process each word separately it's almost impossible to infer the next word as we would miss the all the context.

Example

Take for example the task of predicting the next word: if we process each word separately it's almost impossible to infer the next word as we would miss the all the context.

is ___

Example

Take for example the task of predicting the next word: if we process each word separately it's almost impossible to infer the next word as we would miss the all the context.

is ___

language is ___

Example

Take for example the task of predicting the next word: if we process each word separately it's almost impossible to infer the next word as we would miss the all the context.

is ___

language is ___

my native language is ___

Example

Take for example the task of predicting the next word: if we process each word separately it's almost impossible to infer the next word as we would miss the all the context.

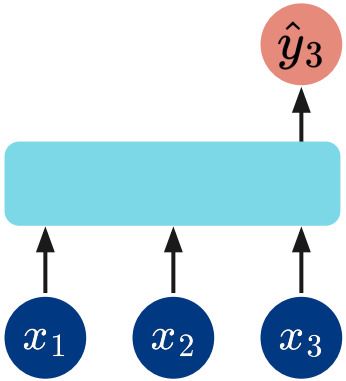
is ___

language is ___

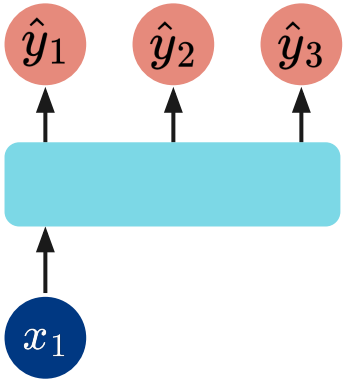
my native language is ___

I'm italian, my native language is ___

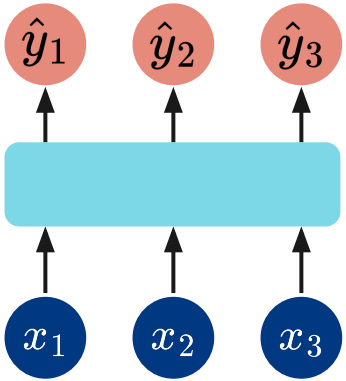
Sequence modelling



Many-to-One



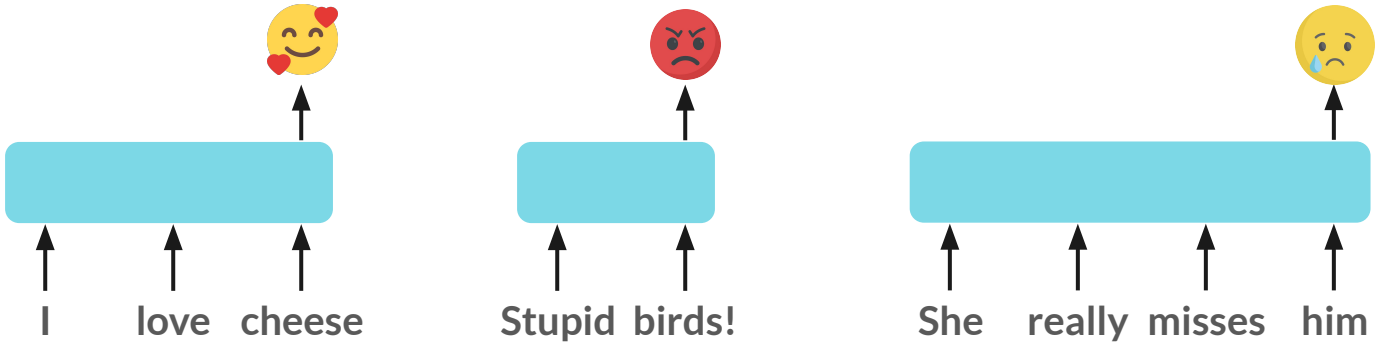
One-to-Many



Many-to-Many

Many-to-One

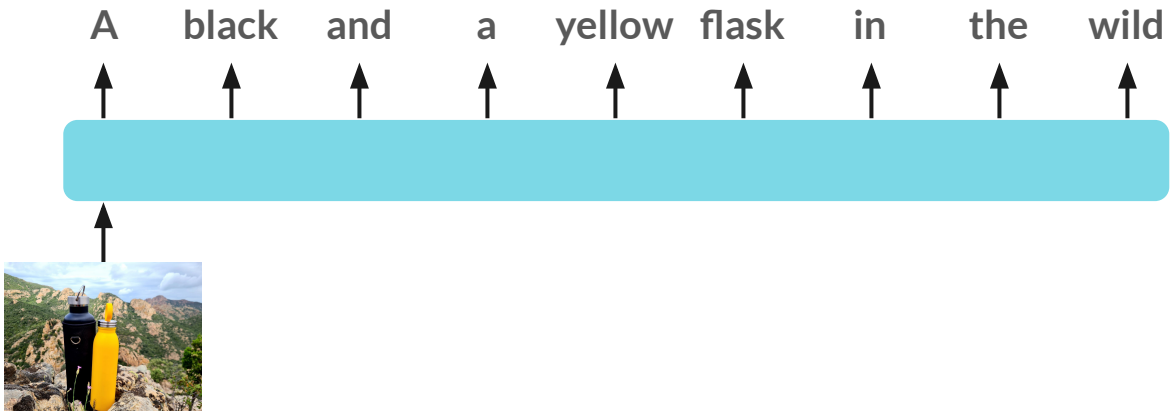
- In **many-to-one** sequence modelling the **input represents a Serie** and the **output represent a single value** (e.g., Sentiment Analysis)
- The **input** has an **arbitrary length** (not specified a priori)



One-to-Many

- In one-to-many sequence modelling the **input represents a single 'object'** and the **output represent a Serie** (e.g., Image Captioning)

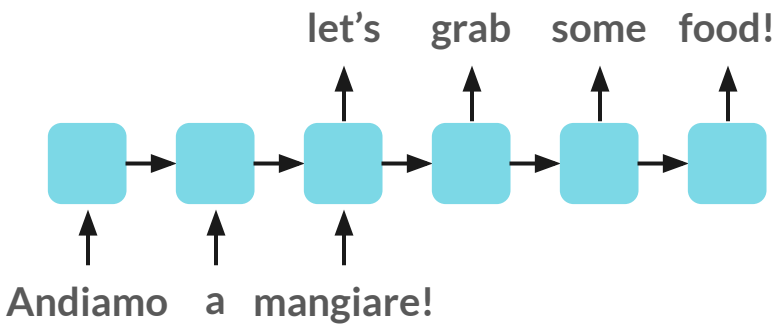
- The **output** has an **arbitrary length** (not specified a priori)



Many-to-Many

- In one-to-many sequence modelling the **both the input and the output represent Series** (e.g., Machine Translation)

- **Both the output and the input has arbitrary length**



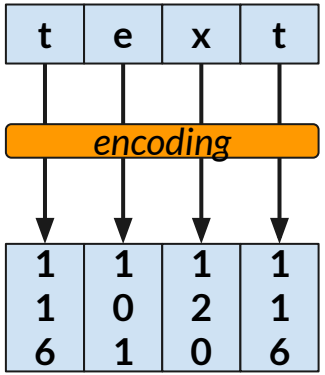
Representing Text

Representing Text

In a computer, a text is represented as a **sequence of characters**, each one with a **unique binary representation**.

Depending on the convention, each character is represented with a certain number of bits, which determines the overall number of characters that can be represented.

For example, while the *American Standard Code for Information Interchange* (**ASCII**) leverage up to 8 bits to represent up to 256 english characters (including letters, numbers, and simple punctuations), the *Unicode Transformation Format* (**UTF**) is a universal character-encoding standard that supports more than 149.000 symbols.

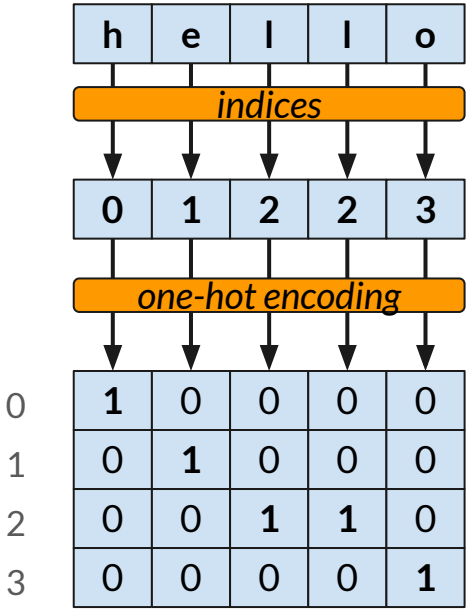


Character-level representation

Passing to the network the raw text is inconvenient as the characters are conventionally ordered, but this order is not useful for text understanding.

Similar to what we did to represent the output classes, we can represent a single character as a one-hot encoded vector.

- Find a set of N characters, e.g. [H, E, L, O]
- Retrieve their indices on, e.g. [0, 1, 2, 2, 3]
- Create an N dimensional vector of all zero
- Put a 1 in the position of the character index.



Recurrent Neural Networks (RNNs)

How can we learn from sequential data?

$$\hat{y}_t = f(x_t)$$

Problems with Neural Networks:

- fixed input dimension
- current output only depends on current input

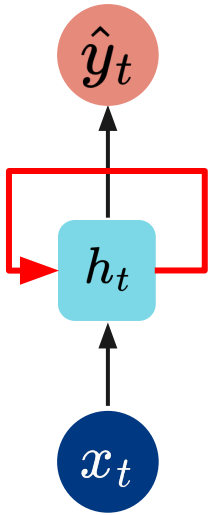
To model sequences we need a model that:

- can deal with **sequences of arbitrary dimension**
- can **remember past input** to produce **contextual predictions**

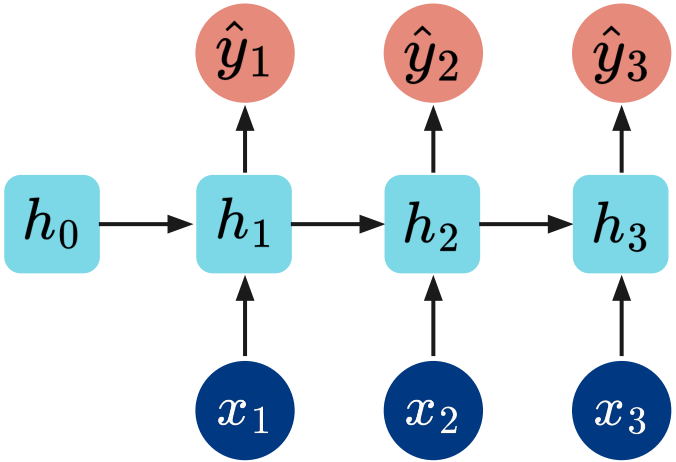
Recurrent Neural Networks (!!!)

$$\hat{y}_t = f(x_t, h_{t-1})$$

- Here, a fully connected layer (FC) with parameters W_t looks at some input x_t, and output a value h_t, called **hidden state**.
- Another FC creates an **internal loop** to pass information from the current hidden state to the next.
- Another FC process the current hidden state (which contains information also about past inputs) to produce an output contextual to the previous inputs.



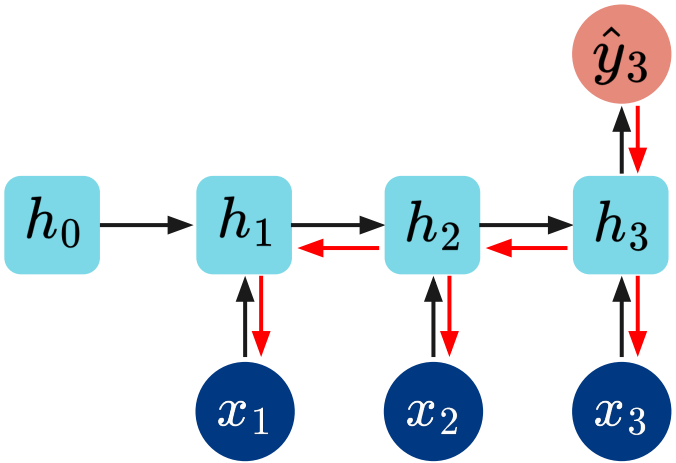
Unfolding the loop



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

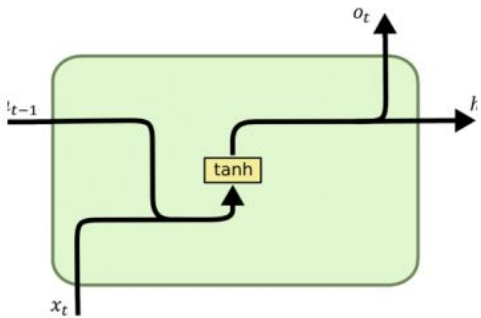
Backpropagation Through Time (BPTT)

$$\frac{\partial \mathcal{L}(\hat{y}_k, y)}{\partial h_k} \cdot \frac{\partial h_k}{\partial h_{k-1}} \cdot \frac{\partial h_{k-1}}{\partial h_{k-2}} \cdots \frac{\partial h_1}{\partial W_h}$$

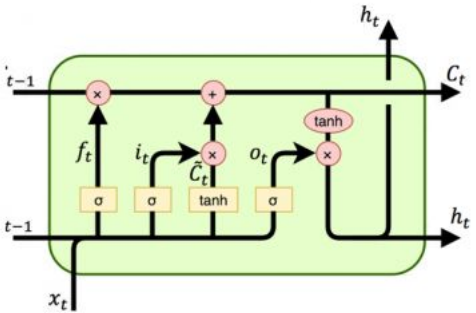


- The gradients propagate throughout all the time-steps
- High risk of vanishing or exploding gradients due to the serie of multiplication from the derivative chain rule
- Many solutions try to develop recurrent units which combine wisely the activation functions for avoiding exploding/vanishing gradients

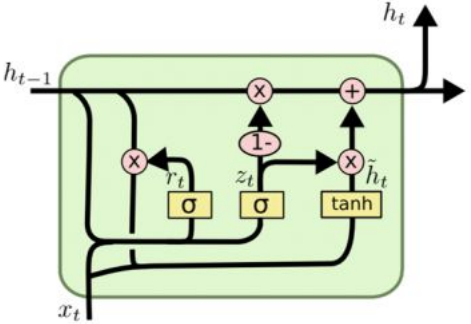
Different types of RNNs



Regular RNN



LSTM



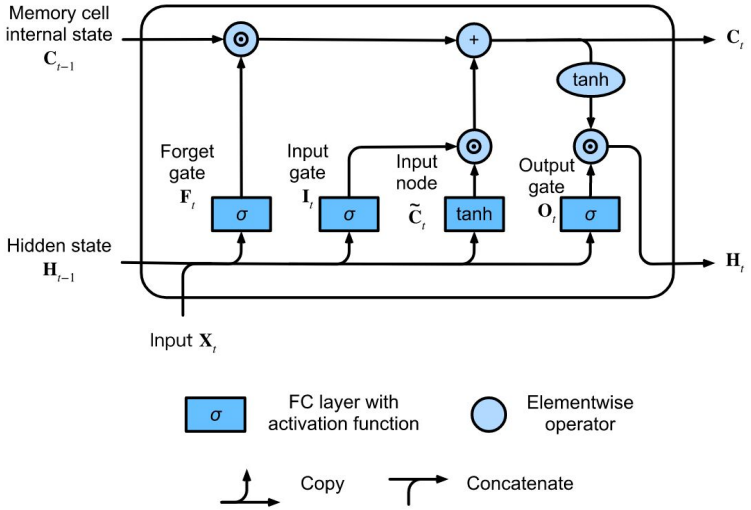
GRU

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory Cell (LSTM)

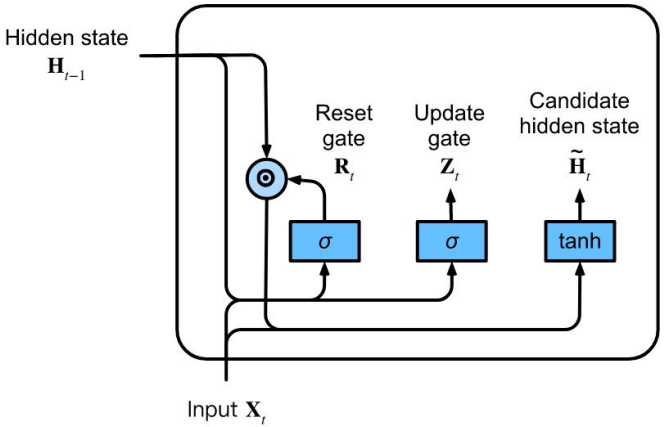
The LSTM are special types of RNNs that are capable of learning long term relationships between the inputs (e.g., very distant words in a sentence)

The key difference is that the LSTM have different *gates* that controls the hidden state, with a more complex mechanism to *update* or *reset* it.



Gated Recurrent Units (GRU)

GRU is another type of RNN which takes the idea of the LSTM and simplify the architecture to run faster.

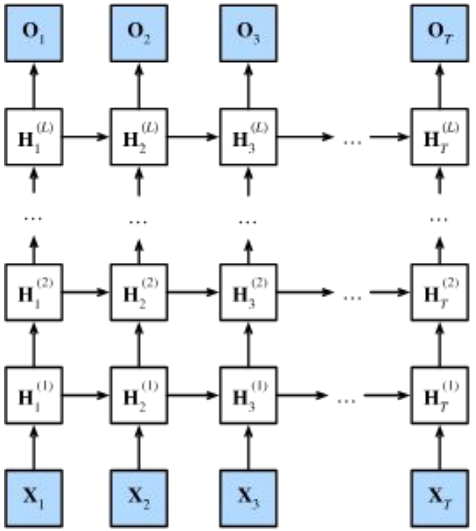


Deep Recurrent Neural Network

We can also stack RNNs one after another.

In this way the network will learn hierarchical information during the sequence processing.

With L layers, the hidden state have shape (L, hidden_dim)



Problem with going only one direction

For some task, such as predicting the missing words, we can also look the sequence from the end instead of just relying on the already processed tokens.

For example:

- I am ___. (happy?)

Problem with going only one direction

For some task, such as predicting the missing words, we can also look the sequence from the end instead of just relying on the already processed tokens.

For example:

- I am ___.
- I am ___ hungry. (ah ok, maybe ‘not hungry’?)

Problem with going only one direction

For some task, such as predicting the missing words, we can also look the sequence from the end instead of just relying on the already processed tokens.

For example:

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a watermelon. (nevermind, it should be ‘very hungry’)

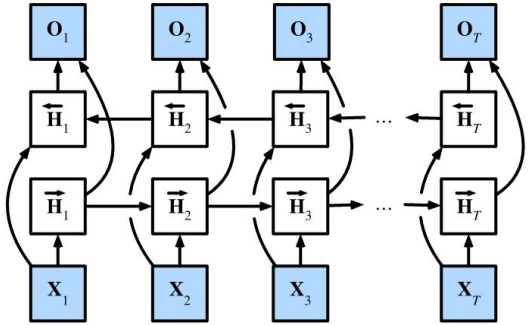
Here the possible outcome can change dramatically depending on the tokens that comes after the missing ones.

Bidirectional Recurrent Neural Networks

In this case there is an RNN for each direction

This is easily done in PyTorch by setting `bidirectional=True`.

The shape of the hidden state is now `(2*L, hidden_dim)` for a network with L layers.



LSTM and GRU in PyTorch

The torch.nn PyTorch module comes to help again, as RNN, LSTM and GRU are already implemented.

- Here we can specify the `input_size`, i.e., the dimension of the single input in the sequence
- The `hidden_size`, i.e., the size of the hidden vector we use to make the network recurrent
- The `num_layers`, i.e., the number of recurrent layers

RNNs in PyTorch

CLASS torch.nn.RNN(*input_size*, *hidden_size*, *num_layers=1*, *nonlinearity='tanh'*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *device=None*, *dtype=None*) [SOURCE]

CLASS torch.nn.LSTM(*input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *proj_size=0*, *device=None*, *dtype=None*) [SOURCE]

CLASS torch.nn.GRU(*input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *device=None*, *dtype=None*) [SOURCE]

Tensor shapes for unbatched inputs

If we are working **without batching the data**, i.e. one sample at a time:

- **Input**: (sequence_length, input_size)
- **Hidden state**: (D*num_layers, hidden_size), where D=2 if bidirectional=True, 1 otherwise *

The output of the RNN will be:

- **New hidden state**: same shape as the initial hidden state given as input.
- **Output**: this is the collection of all the hidden state from the last layer produced for each token in the sequence, so it has shape (sequence_length, hidden_size)

* *The hidden state, if not initialized by hand, will be initialized internally to a zero tensor with proper shape*

Tensor shapes for batched inputs

If we are working **with batches**, i.e. more than one sample at a time (setting batch_first=True in the RNN class):

- **Input**: (batch_size, sequence_length, input_size)
- **Hidden state**: (D*num_layers, batch_size, hidden_size), where D=2 if bidirectional=True, 1 otherwise *

The output of the RNN will be:

- **New hidden state**: same shape as the initial hidden state given as input.
- **Output**: this is the collection of all the hidden state from the last layer produced for each token in the sequence, so it has shape (batch_size, sequence_length, hidden_size)

* *The hidden state, if not initialized by hand, will be initialized internally to a zero tensor with proper shape*

Character-level generation with RNNs

Character-level generation with RNNs

Goal: generate new names of a specified nationality

The Dataset

The dataset with the category (the nationalities) and names can be downloaded from this url.

After downloading and a little bit of processing we end up with a python dictionary containing a list of words for every category.

We can see here that the dataset is highly imbalanced.

	Category	# Names
0	Vietnamese	73
1	German	724
2	Arabic	2000
3	Dutch	297
4	Chinese	268
5	Spanish	298
6	Portuguese	74
7	Italian	709
8	Czech	519
9	Scottish	100
10	Japanese	991
11	English	3668
12	Russian	9408
13	Polish	139
14	Irish	232
15	Greek	203
16	Korean	94
17	French	277

Preprocessing

1. Defining a vocabulary of characters: we choose to consider all the printable ASCII characters
2. Converting all the names from unicode to ASCII
3. Defining some functions to transform a given name to a sequence of one-hot encoded tensors

Designing the optimization problem

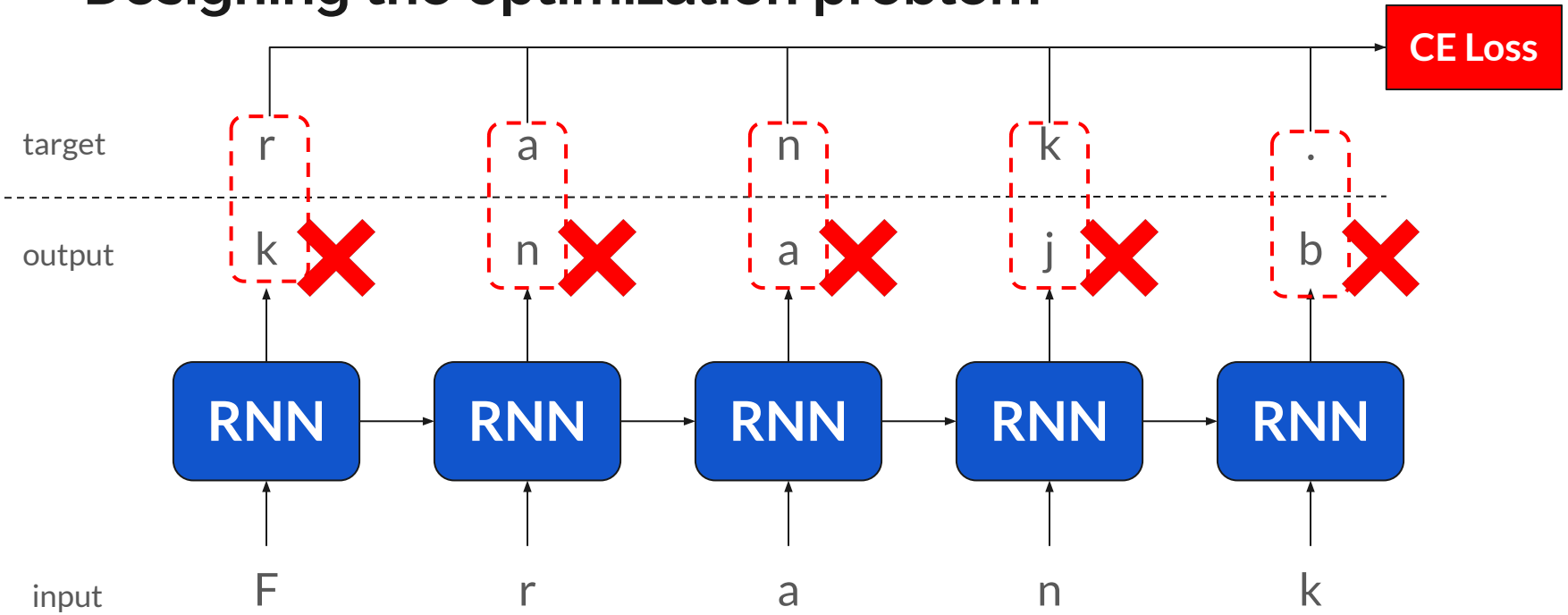
Idea: predicting the next character, e.g. if the objective is to produce the name “Frank”, the RNN should learn to produce consecutively the character ‘r’, ‘a’, ‘n’, and ‘k’, after feeding him with the first character ‘F’.

The output layer is a classifier that select the most likely character from a given set.

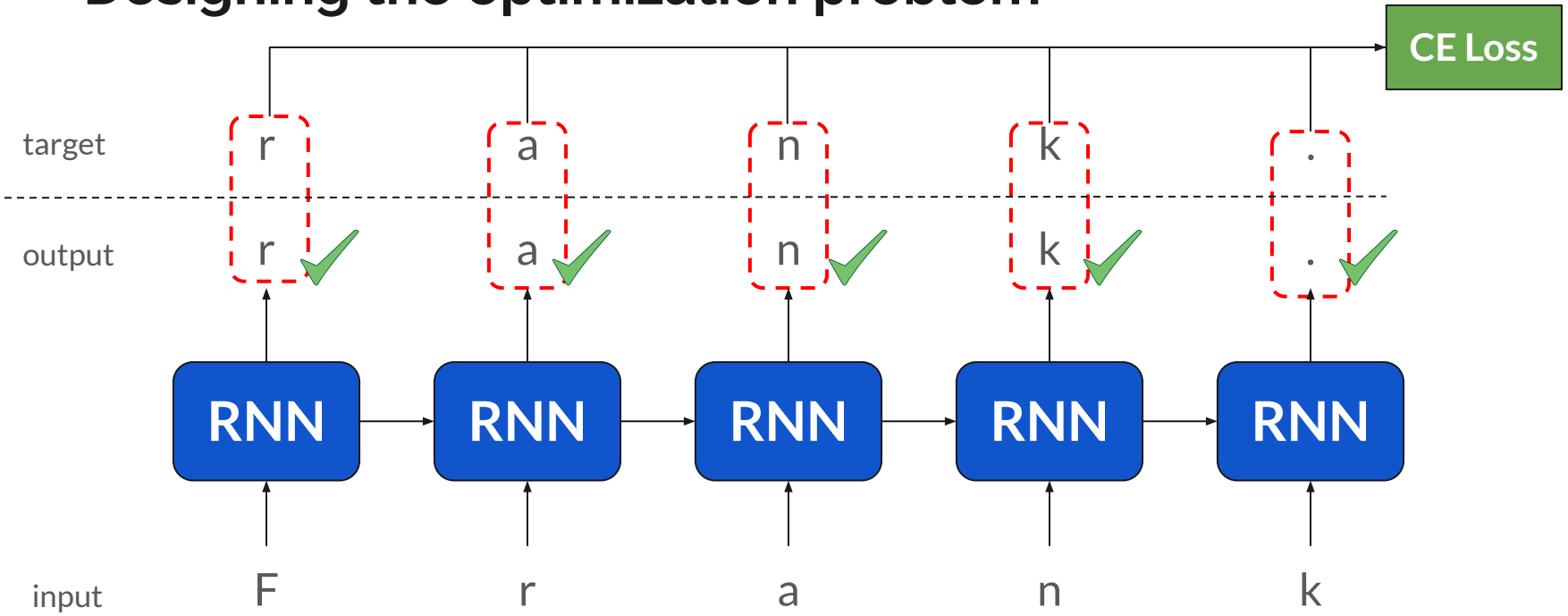
We can set the target as:

- the same name but shifted left
- with a special character that we will use as an End of Sequence signal, for example the dot symbol.

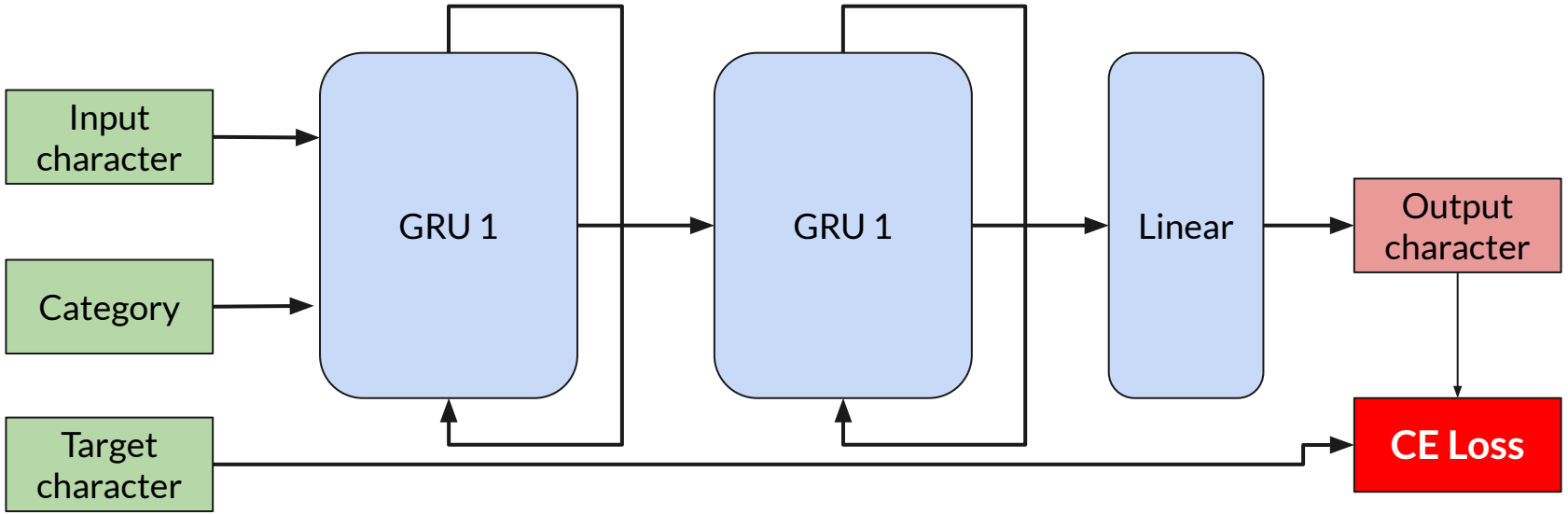
Designing the optimization problem



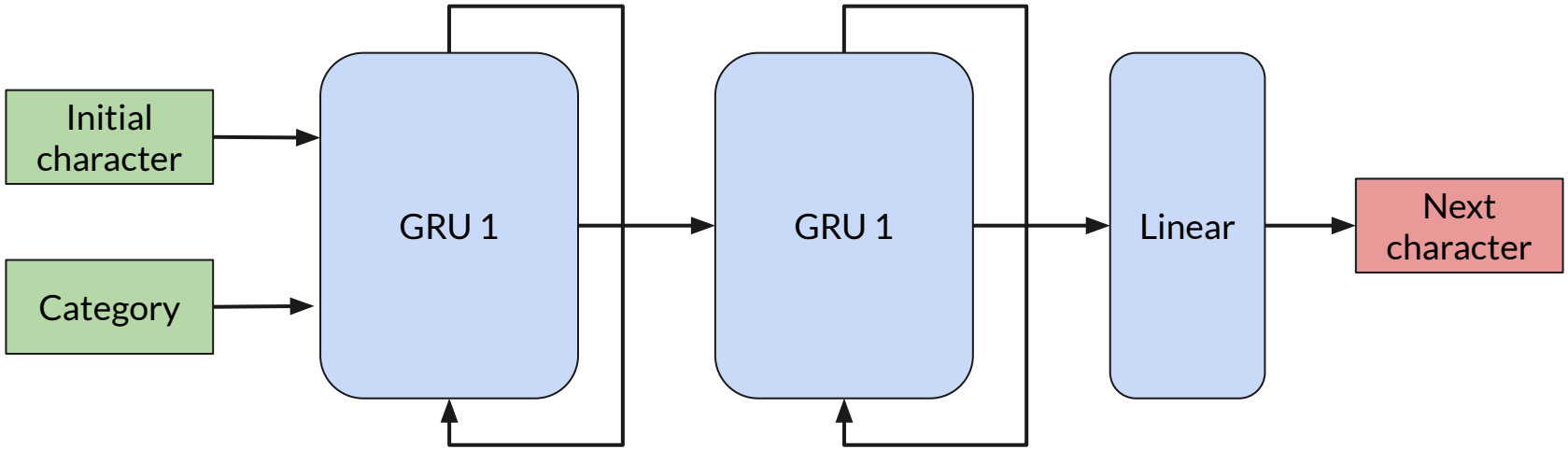
Designing the optimization problem



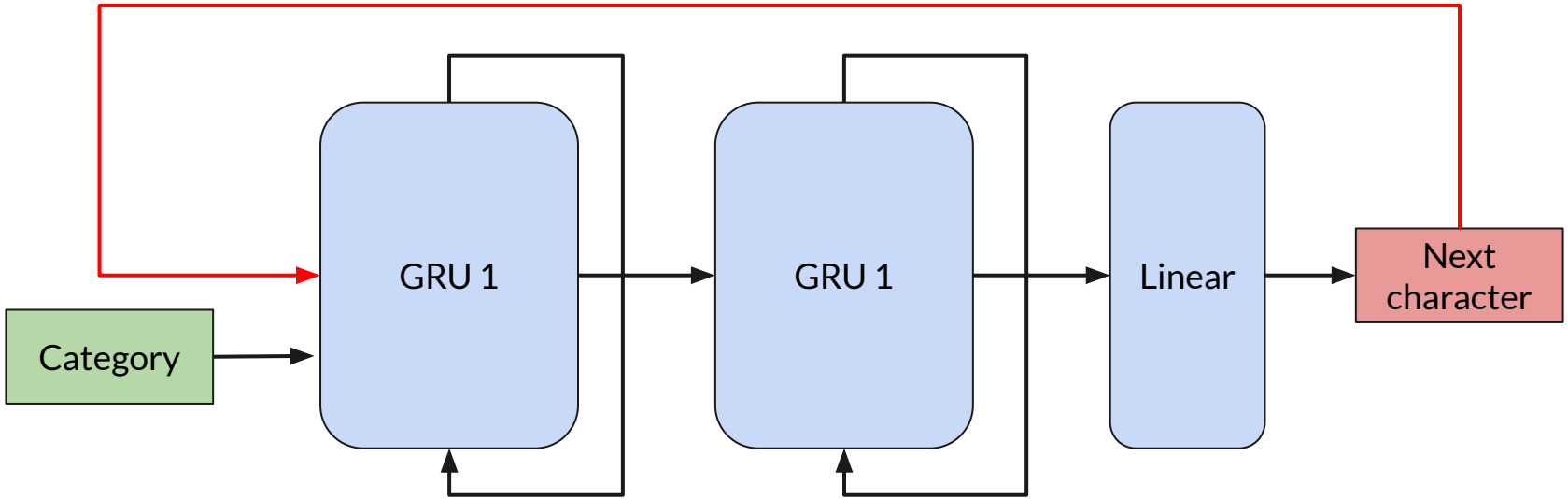
Network Architecture (for training)



Network Architecture (for generating)



Network Architecture (for generating)



RNN Architecture

```
import torch.nn as nn
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
                 n_categories, num_recurrent_layers=2):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.n_categories = n_categories
        self.num_recurrent_layers = num_recurrent_layers
        self.rnn = nn.GRU(input_size=input_size + n_categories,
                          hidden_size=hidden_size,
                          num_layers=num_recurrent_layers)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, category_tensor, input_tensor, hidden_tensor):
        input_combined = torch.cat((category_tensor, input_tensor), 1)
        output, hidden = self.rnn(input_combined, hidden_tensor)
        output = self.linear(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(self.num_recurrent_layers, self.hidden_size)
```

Training

```python
from torch.optim import Adam

def train(category_tensor, input_line_tensor, target_line_tensor,
          criterion, optimizer, device):
    category_tensor = category_tensor.to(device)
    input_line_tensor = input_line_tensor.to(device)
    target_line_tensor.unsqueeze_(-1)
    hidden = rnn.init_hidden().to(device)

    loss = 0
    for i in range(input_line_tensor.size(0)):  # iterate the sequence
        output, hidden = rnn(category_tensor, input_line_tensor[i], hidden)
        # Here I take into account each loss at each iteration
        loss += criterion(output, target_line_tensor[i].to(device))
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    return output, loss.item() / input_line_tensor.size(0)

```

```
21  device = 'cuda' if torch.cuda.is_available() else 'cpu'
22  assert device == 'cuda'
23
24  category_lines, all_categories = load_data()
25  n_categories = len(all_categories)
26
27  n_iters = 10_000
28  learning_rate = 1e-3
29  rnn = RNN(input_size=N_LETTERS,
30            hidden_size=128,
31            output_size=N_LETTERS,
32            n_categories=n_categories,
33            num_recurrent_layers=3).to(device)
34  optimizer = Adam(rnn.parameters(), lr=learning_rate)
35  criterion = nn.CrossEntropyLoss()
36
```

Training

```
current_loss = 0
all_losses = []
print_steps = 100
for iter in range(n_iters):
    data = random_training_example(category_lines, all_categories)
    category_tensor, line_tensor, target_tensor = data

    output, loss = train(category_tensor, line_tensor, target_tensor,
                         criterion, optimizer, device)

    current_loss += loss
    if (iter + 1) % print_steps == 0:
        current_loss /= print_steps
        all_losses.append(current_loss)
        print(f"[{(iter + 1)/n_iters * 100:.2f}%] -  Loss: {current_loss:.4f}")
        current_loss = 0

torch.save(rnn.state_dict(), 'data/rnn_gen.pt')
```

Generating new names

```
from torch.nn import Softmax
# Sample from a category and starting letter
def sample(rnn, category, start_letter='A', temperature=0.8):
    with torch.no_grad():
        category_idx = all_categories.index(category)   # pick the index of the category

        # make one-hot of category
        category_tensor = torch.zeros(1, n_categories)
        category_tensor[0][category_idx] = 1

        input = line_to_tensor(start_letter)
        hidden = rnn.init_hidden()

        output_name = start_letter  # initialize the output string
        for i in range(max_length):
            output, hidden = rnn(category_tensor, input[0], hidden)

            # obtain the probability distribution over the alphabet
            probs = Softmax(dim=1)(output / temperature)

            # instead of picking the argmax here we sample from the probability
            # distribution to obtain more variability in the generation
            next_char = torch.multinomial(probs, num_samples=1)

```

Generating new names

```
31          # instead of picking the argmax here we sample from the probability
32          # distribution to obtain more variability in the generation
33          next_char = torch.multinomial(probs, num_samples=1)
34
35          # here next char is a tensor of shape (1, 1)
36          # we pick the item and compare the predicted index with the index of the EOS character
37          if next_char.item() == letter_to_index(EOS):
38              break   # break if the model predicted the EOS
39          else:
40              letter = ALL_LETTERS[next_char]
41              output_name += letter
42          input = line_to_tensor(letter)
43
44      check = 'NEW' if output_name not in category_lines[category] else 'EXISTING'
45      print(f"{output_name} ({check})")
46      return output_name
```

Generating new names

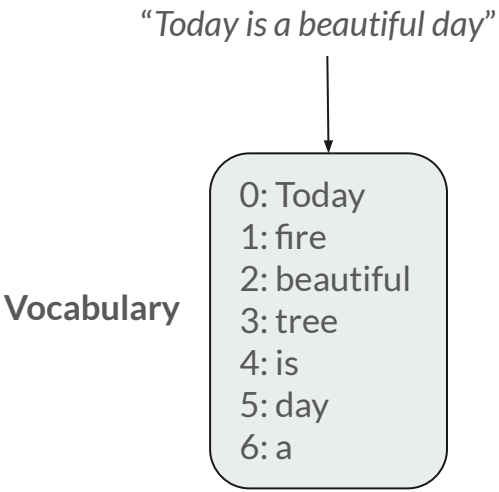
```python
def samples(rnn, category, start_letters):
    print("### Category: ", category)
    for start_letter in start_letters:
        sample(rnn, category, start_letter=start_letter)

samples(rnn, 'Italian', 'ABC')
samples(rnn, 'English', 'ABC')
samples(rnn, 'Chinese', 'ABC')
samples(rnn, 'Russian', 'ABC')
```

```
### Category:  Italian
Alvioto (NEW)
Banzerti (NEW)
Cambro (NEW)
### Category:  English
Altilen (NEW)
Beury (NEW)
Clatlan (NEW)
### Category:  Chinese
Anla (NEW)
Ban (EXISTING)
Ceoh (NEW)
### Category:  Russian
Abadol (NEW)
Bacharov (NEW)
Chruzek (NEW)
```

Word-level representation

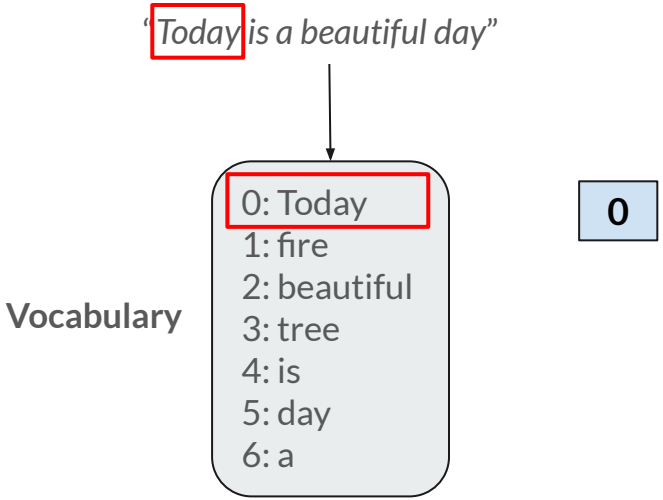
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



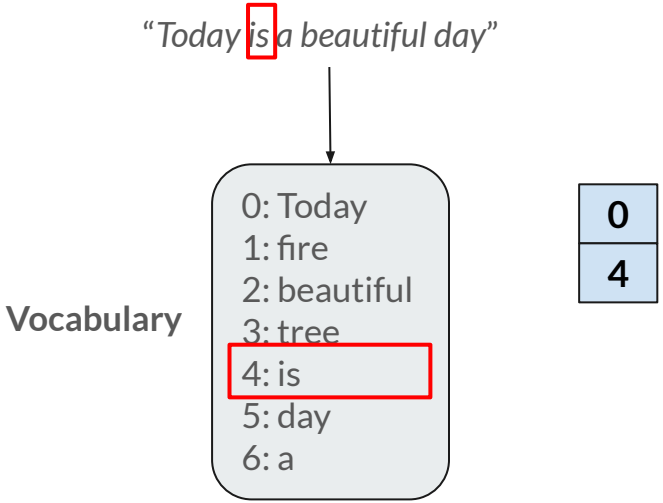
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



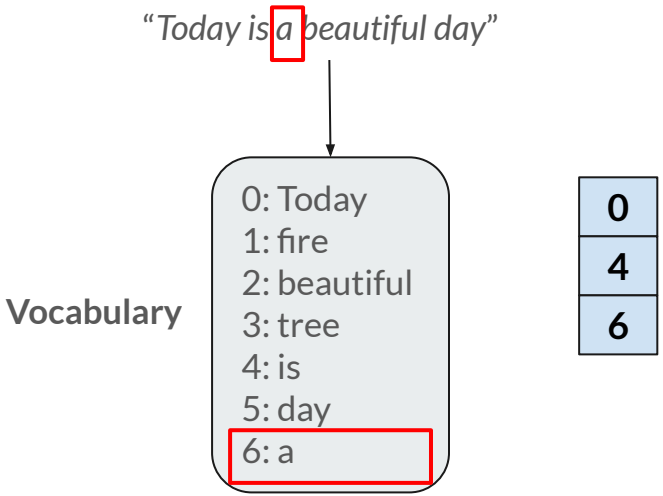
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



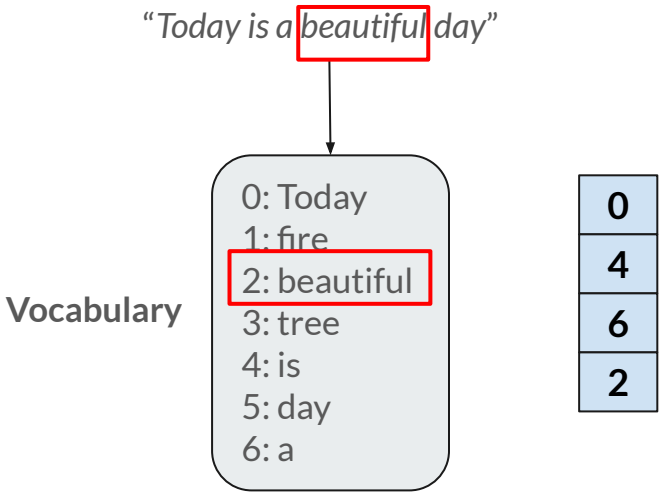
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



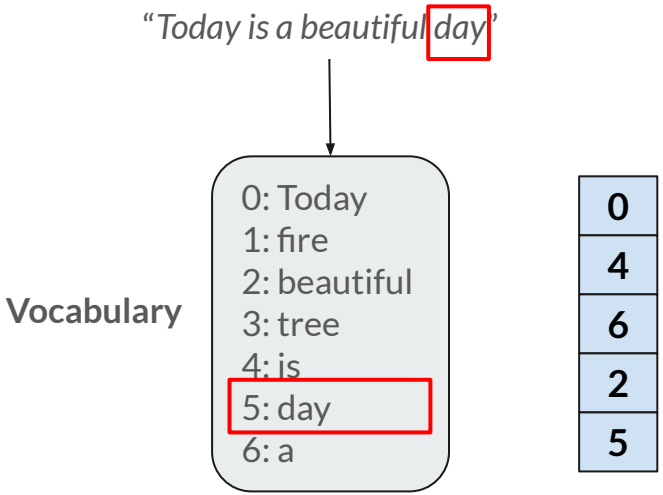
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



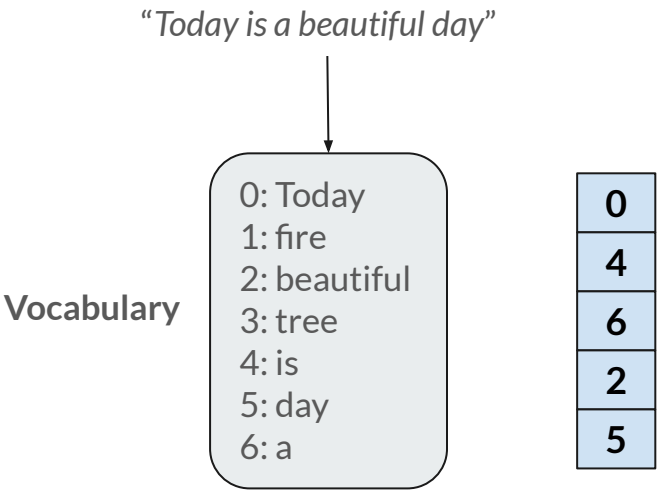
Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary



Word-level representation

- Similarly to character-level representation we can split the sentence at a word level (**tokenization**)
- Each word is called also called **token**
- We can create a vocabulary of words by processing a given text corpus
- And we can transform a given sentence by assigning to each word its corresponding index in the vocabulary
- From this indices we can easily derive the one-hot encoding for each word



Issues with one-hot encoding

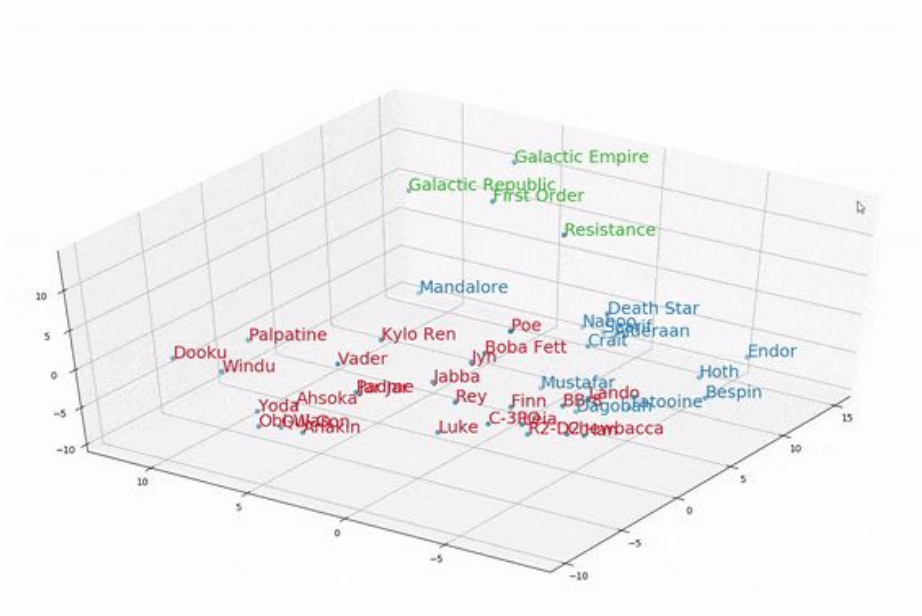
However, for some complex tasks, using one-hot encoding is not convenient as we cannot exploit the relashionship between different words, e.g. the model should recognize that the word “car” is more similar to “truck” than the word “flower”.

One of the most common similarity measure is the **cosine similarity** $S(w_1, w_2) = \frac{w_1 \cdot w_2}{||w_1|| \cdot ||w_2||}$

If any pair of one-hot words w_1 and w_2 are orthogonal, it means that their scalar product will always be zero, and which also make the cosine similarity always zero.

Word Embedding

One of the most effective solution is to **represent words in a high-dimensional space** so that semantically similar words are closer, while very different ones are more distant.



Word Embedding

PyTorch provides the nn.Embedding layer, which transform the index of a word to a tensor that, once trained, is expected to lie in the same region of similar words.

```
embed = nn.Embedding(num_embeddings=vocab_len,
                     embedding_dim=128,
                     padding_idx=1
                     )
```

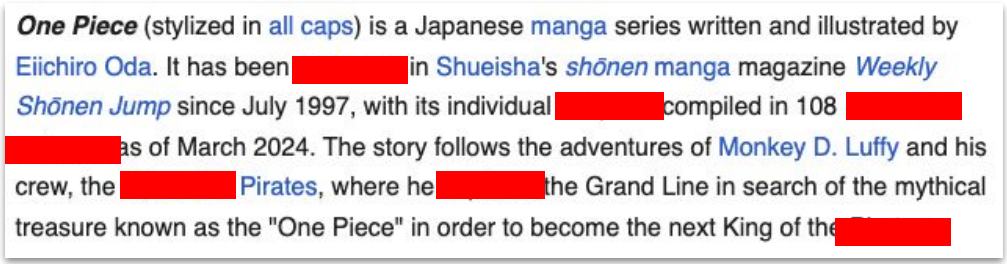
To instantiate an embedding layer we have to specify the number of embeddings (usually the number of words in the vocabulary) and the embedding dimensions.

We can also specify a special index called **padding index**, that acts as a dummy index on which we don't want to calculate gradients (more on padding later)

Pre-trained Word Embedding

Generally, we use standard pre-computed embeddings which are dictionaries mapping words in multidimensional vectors.

These pre-trained embedding models are typically trained on a large text corpus to perform tasks such as filling the blanks in a sentence, which require the model to understand both the meaning and the context inside a text.



Processing sequences parallelly

In the last example we processed one name at a time manually.

However, in PyTorch we can just grab the whole sequence (expressed by the tensor of word indices) and feed it to the network.

We also want to exploit parallelism by passing an entire batch of sequences in a single shot.

With images doing this was simple as each image have the same dimension, making it possible to create a 4-dimensional tensor from them with a consistent shape.

How can we create consistent tensors when the sequence in a batch have different lengths?

Padding

A solution is to choose a special token called **padding token**, that will have a specific index in the vocabulary: the **padding index**.

We can add a certain numbers of this token at the end of each sequence of a batch to make their length match.

Preprocessing: loading the batch

batch size = 3

- *“This is the first sequence”*
- *“Second sequence here”*
- *“This is the last sequence, I promise”*

Load a batch of samples (sequences)

Preprocessing: tokenization

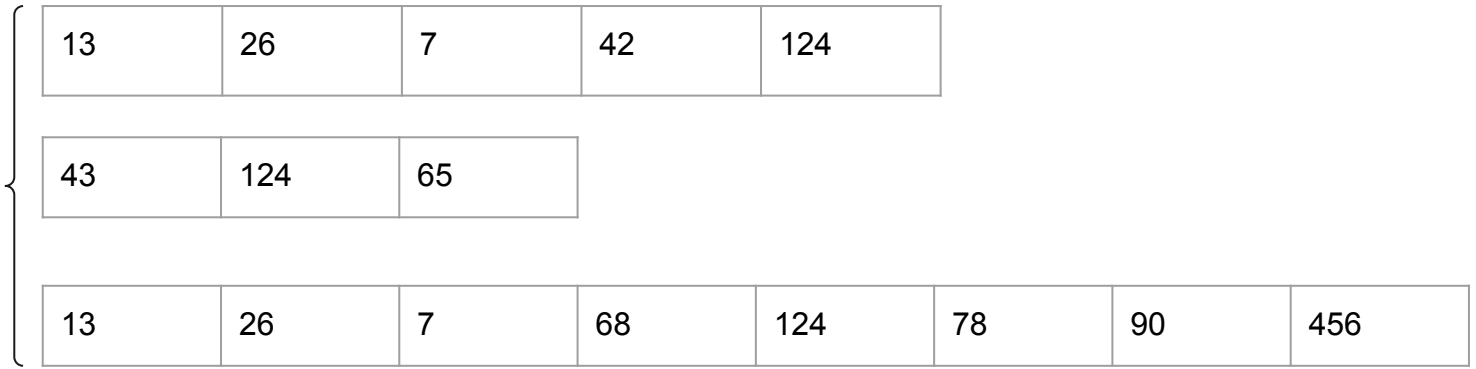
batch size = 3

This	is	the	first	sequence			
Second	sequence	here					
This	is	the	last	sequence	,	I	promise

Tokenizing each sequence (i.e., splitting the sentence in a sequence of tokens)

Preprocessing: finding vocabulary indices

batch size = 3



Finding the indices of each token from a given vocabulary

Preprocessing: padding

batch size = 3

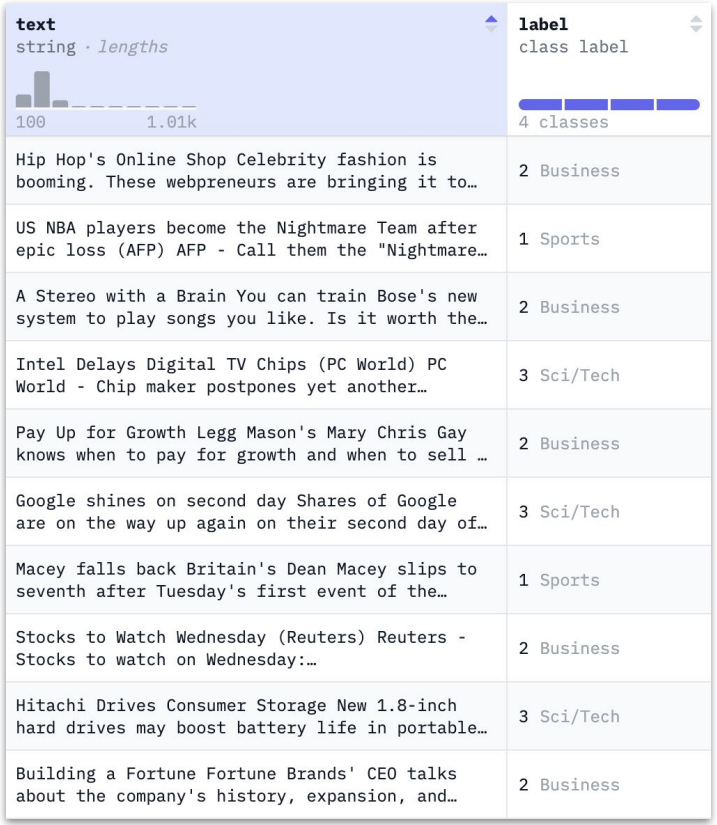
13	26	7	42	124	0	0	0
43	124	65	0	0	0	0	0
13	26	7	68	124	78	90	456

Using the padding index (for example the index 0) to make all the sequences dimension match

Word-level Text Classification with RNNs

The AG News dataset

- AG News (AG's News Corpus) is a subdataset of AG's corpus of news articles constructed by assembling titles and description fields of articles
- Each article belongs to one of 4 classes:
  - World
  - Sports
  - Business
  - Sci/Tech
- The AG News contains 30,000 training and 1,900 test samples per class.



text string · *lengths* 100 1.01k	label class label 4 classes
Hip Hop's Online Shop Celebrity fashion is booming. These webpreneurs are bringing it to…	2 Business
US NBA players become the Nightmare Team after epic loss (AFP) AFP - Call them the "Nightmare…	1 Sports
A Stereo with a Brain You can train Bose's new system to play songs you like. Is it worth the…	2 Business
Intel Delays Digital TV Chips (PC World) PC World - Chip maker postpones yet another…	3 Sci/Tech
Pay Up for Growth Legg Mason's Mary Chris Gay knows when to pay for growth and when to sell …	2 Business
Google shines on second day Shares of Google are on the way up again on their second day of…	3 Sci/Tech
Macey falls back Britain's Dean Macey slips to seventh after Tuesday's first event of the…	1 Sports
Stocks to Watch Wednesday (Reuters) Reuters - Stocks to watch on Wednesday:…	2 Business
Hitachi Drives Consumer Storage New 1.8-inch hard drives may boost battery life in portable…	3 Sci/Tech
Building a Fortune Fortune Brands' CEO talks about the company's history, expansion, and…	2 Business

Prerequisites

Before proceeding it's necessary to manually install some libraries and fix some internal bugs.

The torchtext library (which is the equivalent of torchvision for text processing) is still in a beta version, so there are some compatibility problems.

```
!pip install torchtext torchdata portalocker
```

```
import torch
torch.utils.data.datapipes.utils.common.DILL_AVAILABLE = torch.utils._import_utils.dill_available()
import torchdata
```

Loading the datasets

Differently from the already seen datasets which are mappings from an index to the (sample, label) pair, the AG_NEWS dataset returns an iterator, so we treat it as we treated a DataLoader object.

```python
import torch
from torchtext.datasets import AG_NEWS

train_iter = iter(AG_NEWS(split="train"))

label, input = next(train_iter)
print(f"input: {input}")
print(f"label: {label}")
```

```
input: Wall St. Bears Claw Back Into the Black (Reuters) Reuters - Short-sellers, Wall Street's dwindlin
label: 3
```

Tokenization

Using the function torchtext.data.utils.get_tokenizer we can easily tokenize sentences.

By setting the “basic_english” keyword, this function returns a tokenizer that before splitting by spaces it performs some normalization of the text, such as lowercasing, removing special characters, or adding spaces between them.

```
1  from torchtext.data.utils import get_tokenizer
2  tokenizer = get_tokenizer("basic_english")
3  tokenized_input = tokenizer(input)
4
5  print(len(tokenized_input))
6  print(tokenized_input)
```

```
29
['wall', 'st', '.', 'bears', 'claw', 'back', 'into', 'the', 'black', '(', 'reuters', ')', 'reuters', '-', 'short-sellers',
```

Building the vocabulary

The function torchtext.vocab.build_vocab_from_iterator function takes as input an iterator (in our case the AG News training iterator) of strings and construct a vocabulary of all the seen words.

We can also specify some special tokens, such as the “unkown token” and “padding token”.

```
from torchtext.vocab import build_vocab_from_iterator
train_iter = AG_NEWS(split="train")
def yield_tokens(data_iter):
    for _, text in data_iter:
        yield tokenizer(text)
tokens_iterator = yield_tokens(train_iter)
UNK_TOKEN = '<unk>'
PAD_TOKEN = '<pad>'
specials = [UNK_TOKEN, PAD_TOKEN]
vocab = build_vocab_from_iterator(tokens_iterator, specials=specials)
# set the index of the out-of-vocab token (<unk>) as default (=0)
vocab.set_default_index(vocab[UNK_TOKEN])
```

Using the vocabulary

We can also retrieve the vocabulary to go from a specific token to its corresponding index and viceversa:

```
1  index_to_tokens_dict = vocab.get_itos()
2  tokens_to_index_dict = vocab.get_stoi()
```

We can find the indices of a sentence by passing the tokenized sentence as argument to the vocabulary object:

```
1  vocab(tokenizer("This is how we find indices from a sentence"))
```

```
[53, 22, 358, 508, 747, 18963, 30, 6, 2994]
```

Padding

The function torch.nn.utils.rnn.pad_sequence take a list of tensors containing a sequence of indices and create a consistent batched tensors by padding the sequences with a given padding value:

```
1  from torch.nn.utils.rnn import pad_sequence
2  input1 = "This is the first sentence"
3  input2 = "This is the second sentence, which is longer"
4  tensor1 = torch.tensor(vocab(tokenizer(input1)))
5  tensor2 = torch.tensor(vocab(tokenizer(input2)))
6  inputs = [tensor1, tensor2]
7  padding_value = tokens_to_index_dict[PAD_TOKEN]
8  padded = pad_sequence(inputs,
9                        padding_value=padding_value, # fill with <pad> tokens
10                       batch_first=True)
11 print(padded)
```

Adding the collate_fn **to the** DataLoader

To efficiently exploit the DataLoader class, we can just define a collate_fn function that can be passed as argument when instantiating the DataLoader.

In this function we can define how to process the batch of (text, label) pair when are loaded from the memory.

In our case, we specify all the already seen preprocessing steps to eventually return two elements:

- The input tensor, with shape (batch_size, sequence_length)
- The label, with shape (batch_size,)

Network Architecture

- **Embedding Layer**, containing a vocab_size hidden tensors of size 128
- **GRU** with 3 recurrent layers, with an hidden size of 128
- A **Linear layer** that maps the last hidden state of GRU to the corresponding 4 labels

```python
from torch import nn

class TextClassificationModel(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_classes,
                 num_recurrent_layers=3, bidirectional=False):
        super(TextClassificationModel, self).__init__()
        self.embed_dim = embed_dim
        self.embedding = nn.Embedding(num_embeddings=vocab_size,
                                      embedding_dim=embed_dim,
                                      padding_idx=tokens_to_index_dict[PAD_TOKEN])
        self.num_recurrent_layers = num_recurrent_layers
        self.rnn = nn.GRU(input_size=embed_dim,
                          hidden_size=embed_dim,
                          num_layers=num_recurrent_layers,
                          bidirectional=bidirectional,
                          batch_first=True)
        n_directions = 2 if bidirectional else 1
        n_out_neurons = embed_dim * n_directions
        self.fc = nn.Linear(n_out_neurons, num_classes)

```

```
20
21      def forward(self, x):
22          # x.shape = (batch_size, seq_len)
23          embedded = self.embedding(x)
24          # embedded.shape = (batch_size, seq_len, embed_dim)
25          # the initial hidden state is initialized to zero by default
26          out, _ = self.rnn(embedded)
27          # out.shape = (batch_size, seq_len, n_out_neurons)
28          out = out[:, -1, :]   # pick only the last output
29          out = self.fc(out)
30          # out.shape = (batch_size, num_classes)
31          return out
```

Training

```python
import time
from torch.utils.data.dataset import random_split
from torchtext.data.functional import to_map_style_dataset

def evaluate(model, dataloader, device):
    model.eval()
    total_correct, total_samples = 0, 0
    with torch.no_grad():
        for idx, (x, y) in enumerate(dataloader):
            x, y = x.to(device), y.to(device)
            ypred = model(x).argmax(dim=1)
            total_correct += (ypred == y).sum().item()
            total_samples += x.shape[0]
    return total_correct / total_samples

# Hyperparameters
EPOCHS = 5  # epoch
LR = 1e-3  # learning rate
BATCH_SIZE = 64  # batch size for training
RANDOM_SEED = 42
embedding_size = 64
```

```
23  device = 'cuda' if torch.cuda.is_available() else 'cpu'
24  assert device == 'cuda'
25
26  # Prepare the train, validation and test dataloaders
27  train_iter, test_iter = AG_NEWS()
28  # Here we convert from iterable dataset to a mapping style
29  # which simply mean that we can access each sample with an index
30  train_dataset = to_map_style_dataset(train_iter)
31  test_dataset = to_map_style_dataset(test_iter)
32
33  train_size = int(len(train_dataset) * 0.95)
34
35  torch.manual_seed(RANDOM_SEED)
36  split_train_, split_valid_ = random_split(
37      train_dataset, [train_size, len(train_dataset) - train_size]
38  )
39
40  train_dataloader = DataLoader(
41      split_train_, batch_size=BATCH_SIZE, shuffle=True, collate_fn=pad_collate_batch
42  )
43  valid_dataloader = DataLoader(
44      split_valid_, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate_batch
45  )
46  test_dataloader = DataLoader(
47      test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=pad_collate_batch
48  )
49
```

```
model = TextClassificationModel(vocab_size, embedding_size, num_class).to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LR)

print_every = 100
accuracy_checkpoint = None

for epoch in range(EPOCHS):
    epoch_start_time = time.time()
    model.train()
    start = time.time()
    total_correct, total_samples = 0, 0
    for idx, (x, y) in enumerate(train_dataloader):
        x, y = x.to(device), y.to(device)
        out = model(x)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

```

```
70
71          ypred = out.argmax(dim=1)
72          total_correct += (ypred == y).sum().item()
73          total_samples += x.shape[0]
74          if idx % print_every == 0:
75              train_accuracy = total_correct / total_samples
76              print(f"Epoch: [{epoch + 1}/{EPOCHS}], "\
77                    f"Batch: [{idx + 1}/{len(train_dataloader)}], "\
78                    f"Train. Accuracy: {train_accuracy:.8f}"
79                    )
80              total_correct, total_samples = 0, 0
81      end = time.time()
82      total_time = end - start
83
84      valid_accuracy = evaluate(model, valid_dataloader, device)
85      accuracy_checkpoint = valid_accuracy
86      print("-" * 59)
87      print(f"> End of epoch {epoch + 1}, took {total_time} seconds")
88      print(f"> Valid. Accuracy: {valid_accuracy}")
89      print("\n")
```

```
Epoch: [1/5], Batch: [1/1782], Train. Accuracy: 0.21875000
Epoch: [1/5], Batch: [101/1782], Train. Accuracy: 0.24703125
Epoch: [1/5], Batch: [201/1782], Train. Accuracy: 0.25421875
Epoch: [1/5], Batch: [301/1782], Train. Accuracy: 0.25062500
Epoch: [1/5], Batch: [401/1782], Train. Accuracy: 0.26203125
Epoch: [1/5], Batch: [501/1782], Train. Accuracy: 0.26609375
Epoch: [1/5], Batch: [601/1782], Train. Accuracy: 0.28203125
Epoch: [1/5], Batch: [701/1782], Train. Accuracy: 0.25062500
Epoch: [1/5], Batch: [801/1782], Train. Accuracy: 0.24593750
Epoch: [1/5], Batch: [901/1782], Train. Accuracy: 0.27968750
Epoch: [1/5], Batch: [1001/1782], Train. Accuracy: 0.43390625
Epoch: [1/5], Batch: [1101/1782], Train. Accuracy: 0.61406250
Epoch: [1/5], Batch: [1201/1782], Train. Accuracy: 0.73171875
Epoch: [1/5], Batch: [1301/1782], Train. Accuracy: 0.78031250
Epoch: [1/5], Batch: [1401/1782], Train. Accuracy: 0.80937500
Epoch: [1/5], Batch: [1501/1782], Train. Accuracy: 0.81781250
Epoch: [1/5], Batch: [1601/1782], Train. Accuracy: 0.84312500
Epoch: [1/5], Batch: [1701/1782], Train. Accuracy: 0.85125000
-----------------------------------------------------------
> End of epoch 1, took 14.830350875854492 seconds
> Valid. Accuracy: 0.8598333333333333


Epoch: [2/5], Batch: [1/1782], Train. Accuracy: 0.84375000
Epoch: [2/5], Batch: [101/1782], Train. Accuracy: 0.87671875
Epoch: [2/5], Batch: [201/1782], Train. Accuracy: 0.88343750
Epoch: [2/5], Batch: [301/1782], Train. Accuracy: 0.88531250
Epoch: [2/5], Batch: [401/1782], Train. Accuracy: 0.88265625
Epoch: [2/5], Batch: [501/1782], Train. Accuracy: 0.89265625
Epoch: [2/5], Batch: [601/1782], Train. Accuracy: 0.89343750
Epoch: [2/5], Batch: [701/1782], Train. Accuracy: 0.89375000
Epoch: [2/5], Batch: [801/1782], Train. Accuracy: 0.89968750
Epoch: [2/5], Batch: [901/1782], Train. Accuracy: 0.89453125
```

Evaluting

```python
print("Checking the results of test dataset.")
test_accuracy = evaluate(model, test_dataloader, device)
print(f"Test accuracy: {test_accuracy:.4f}")
```

```
Checking the results of test dataset.
Test accuracy: 0.9084
```

```python
ag_news_label = {1: "World", 2: "Sports", 3: "Business", 4: "Sci/Tec"}
def predict(text, text_pipeline):
    with torch.no_grad():
        text = torch.tensor(text_pipeline(text)).unsqueeze(0)
        output = model(text)
        ypred = output.argmax(dim=1).item() + 1
        return output.argmax(1).item() + 1

ex_text_str = "MEMPHIS, Tenn. – Four days ago, Jon Rahm was \
    enduring the season's worst weather conditions on Sunday at The \
    Open on his way to a closing 75 at Royal Portrush, which \
    considering the wind and the rain was a respectable showing. \
    Thursday's first round at the WGC-FedEx St. Jude Invitational \
    was another story. With temperatures in the mid-80s and hardly any \
    wind, the Spaniard was 13 strokes better in a flawless round. \
    Thanks to his best putting performance on the PGA Tour, Rahm \
    finished with an 8-under 62 for a three-stroke lead, which \
    was even more impressive considering he'd never played the \
    front nine at TPC Southwind."

model.to("cpu")
pred = predict(ex_text_str, text_pipeline)
print(f"This is a {ag_news_label[pred]} news")
```

```
This is a Sports news
```

End of part 4

Summary:

- Character-level representation
- Word-level representation
- Text preprocessing
- Recurrent Neural Networks