FIT3181/5215 Deep Learning

Recurrent Neural Networks

Trung Le and Tutor Team

Department of Data Science and AI
Faculty of Information Technology, Monash University
Email: **trunglm@monash.edu**

Outline

- Fundamental of RNNs (Tute 7a) (*****)
- Sentiment Analysis (Tute 7b) (*****)
- Text Generation (Tute 7c – Additional Reading) (***)

Fundamental in RNNs

Manual RNNs

```
import numpy as np

X0 = np.array([[0.0, 1.0, -2.0],
               [-3.0, 4.0, 5.0],
               [6.0, 7.0, -8.0],
               [6.0, -1.0, 2.0]], dtype= np.float32) # t = 0
X1 = np.array([[9.0, 8.0, 7.0],
               [0.0, 0.0, 0.0],
               [6.0, 5.0, 4.0],
               [1.0, 2.0, 3.0]], dtype= np.float32) # t = 1
```

```
hidden_size = 5
input_size = 3

# Creating the parameters
U = torch.nn.Parameter(torch.randn(input_size, hidden_size, dtype=torch.float32))
W = torch.nn.Parameter(torch.randn(hidden_size, hidden_size, dtype=torch.float32))
b = torch.nn.Parameter(torch.zeros(1, hidden_size, dtype=torch.float32))

X0 = torch.tensor(X0)
X1 = torch.tensor(X1)
# Implementing the operations
h0 = torch.tanh(torch.matmul(X0, U) + b)
h1 = torch.tanh(torch.matmul(X1, U) + torch.matmul(h0, W) + b)
```

The computation process is as follows:

- $h_0 = tanh(X_0 \times U + b)$.
- $h_1 = tanh(X_1 \times U + h_0 \times W + b)$.

In the following code, $X0$ is a mini-batch with batch size 4 consisting of the data of time step 0.

- $X0$'s shape is $[batch\_size \times input\_size]$

$X1$ is a mini-batch with batch size 4 consisting of the data of time step 1.

- $X1$'s shape is $[batch\_size \times input\_size]$

```
print("h0= {}".format(h0.detach().numpy()))
```

```
h0= [[ 0.77583134 -0.87714946  0.99381113 -0.98397243 -0.5037168 ]
 [-0.9999998   0.9999607  -1.          0.9448576   0.99701405]
 [ 0.99991125 -0.99994725  1.         -1.         -1.        ]
 [ 0.941103    0.6392131   1.          0.85120136 -1.        ]]
```

```
print("h1= {}".format(h1.detach().numpy()))
```

```
h1= [[-1.          0.9999641   1.         -0.9899772  -1.        ]
 [ 0.9993845   0.91682416 -0.889713    0.94020694 -0.8458057 ]
 [-0.9999997   0.9808309   1.         -0.9970271  -1.        ]
 [-0.99985224  0.9462382   0.9698116  -0.94209725 -0.996053  ]]
```

Standard RNN cells

RNN

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]

Apply a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence. For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then ReLU is used instead of $\tanh$.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Inputs: input, h_0

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for the input sequence batch. Defaults to zeros if not provided.

where:

$$\begin{aligned} N &= \text{batch size} \\ L &= \text{sequence length} \\ D &= 2 \text{ if bidirectional=True otherwise } 1 \\ H_{in} &= \text{input\_size} \\ H_{out} &= \text{hidden\_size} \end{aligned}$$

Outputs: output, h_n

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the RNN, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

```
# Manually initialize h0
inputs = np.random.random([32, 10, 8]).astype(np.float32)
inputs = torch.tensor(inputs)

simple_rnn = torch.nn.RNN(input_size=8, hidden_size=20, num_layers=1, batch_first=True)
h0 = torch.randn(1, 32, 20)

output, h_n = simple_rnn(inputs, h0)  # The output has shape `[32, 10, 20]` and the h_n
```

```
print(output.shape)
print(h_n.shape)
```

```
torch.Size([32, 10, 20])
torch.Size([1, 32, 20])
```

LSTM cells

LSTM

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]

Apply a multi-layer long short-term memory (LSTM) RNN to an input sequence. For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, h_{t-1} is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

Inputs: input, (h_0, c_0)

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.
- **c_0**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the initial cell state for each element in the input sequence. Defaults to zeros if (h_0, c_0) is not provided.

where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{cell} &= \text{hidden\_size} \\
H_{out} &= \text{proj\_size if proj\_size} > 0 \text{ otherwise hidden\_size}
\end{aligned}
$$

Outputs: output, (h_n, c_n)

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the LSTM, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. When `bidirectional=True`, *output* will contain a concatenation of the forward and reverse hidden states at each time step in the sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the sequence. When `bidirectional=True`, *h_n* will contain a concatenation of the final forward and reverse hidden states, respectively.
- **c_n**: tensor of shape $(D * \text{num\_layers}, H_{cell})$ for unbatched input or $(D * \text{num\_layers}, N, H_{cell})$ containing the final cell state for each element in the sequence. When `bidirectional=True`, *c_n* will contain a concatenation of the final forward and reverse cell states, respectively.

```python
# Manually initialize h0 and c0
inputs = torch.randn([32, 10, 8])

lstm = torch.nn.LSTM(input_size=8, hidden_size=20, num_layers=1, batch_first=True)
h0 = torch.randn(1, 32, 20)
c0 = torch.randn(1, 32, 20)

output, (h_n, c_n) = lstm(inputs, (h0, c0))
```

```python
print(output.shape)
print(h_n.shape)
print(c_n.shape)
```

```
torch.Size([32, 10, 20])
torch.Size([1, 32, 20])
torch.Size([1, 32, 20])
```

GRU cells

GRU

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]

Apply a multi-layer gated recurrent unit (GRU) RNN to an input sequence. For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) \odot n_t + z_t \odot h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $\odot$ is the Hadamard product.

Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

Inputs: input, h_0

- **input**: tensor of shape (L, H_{in}) for unbatched input, (L, N, H_{in}) when `batch_first=False` or (N, L, H_{in}) when `batch_first=True` containing the features of the input sequence. The input can also be a packed variable length sequence. See `torch.nn.utils.rnn.pack_padded_sequence()` or `torch.nn.utils.rnn.pack_sequence()` for details.
- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for the input sequence. Defaults to zeros if not provided.

where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{out} &= \text{hidden\_size}
\end{aligned}
$$

Outputs: output, h_n

- **output**: tensor of shape $(L, D * H_{out})$ for unbatched input, $(L, N, D * H_{out})$ when `batch_first=False` or $(N, L, D * H_{out})$ when `batch_first=True` containing the output features (*h_t*) from the last layer of the GRU, for each *t*. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for the input sequence.

```
# Manually initialize h0
inputs = torch.randn([32, 10, 8])

lstm = torch.nn.GRU(input_size=8, hidden_size=20, num_layers=1, batch_first=True)
h0 = torch.randn(1, 32, 20)

output, h_n = lstm(inputs, h0)
```

```
print(output.shape)
print(h_n.shape)
```

```
torch.Size([32, 10, 20])
torch.Size([1, 32, 20])
```

RNN for sentiment analysis

Download the dataset

```
import os
import requests
from torchvision.datasets.utils import download_and_extract_archive

def download_and_read(url):
    # download and extract the file
    download_and_extract_archive(url, download_root='datasets')

    local_file = url.split('/')[-1]
    local_file = local_file.replace("%20", " ")
    local_folder = os.path.join("datasets", local_file.split('.')[0])

    labeled_sentences = []
    for labeled_filename in os.listdir(local_folder):
        if labeled_filename.endswith("_labelled.txt"):
            with open(os.path.join(local_folder, labeled_filename), "r") as f:
                for line in f:
                    sentence, label = line.strip().split('\t')
                    labeled_sentences.append((sentence, label))
    return labeled_sentences
```

- The function download_and_read(url) supports us to download the dataset at the specific url and store to the folder **datasets** inside the current directory.
- Return a **list of all sentences** and **labels** of this dataset.

```
sentences = [s for (s, l) in labeled_sentences]
labels = [int(l) for (s, l) in labeled_sentences]
```

```
for (s,l) in zip(sentences[0:5], labels[0:5]):
    print("Sentence: {}".format(s))
    print("Label: {}".format(l))
    print("\n")
```

```
Sentence: So there is no way for me to plug it in here in the US unless I go by a converter.
Label: 0


Sentence: Good case, Excellent value.
Label: 1


Sentence: Great for the jawbone.
Label: 1


Sentence: Tied to charger for conversations lasting more than 45 minutes.MAJOR PROBLEMS!!
Label: 0


Sentence: The mic is great.
Label: 1
```

Process data and create dictionaries

```python
from collections import Counter
from nltk.tokenize import word_tokenize

# create a custom dataset class
class TextDataset():
    def __init__(self, sentences):
        self.sentences = sentences
        self.word2idx = {}
        self.idx2word = {}

        # create vocabulary
        word_freq = Counter()
        for sent in self.sentences:
            #tokens consists of words in a sentence, e.g., ['think', 'of', 'the', 'film', 'being', 'like', 'a', 'dream', '.']
            tokens = word_tokenize(sent.lower()) # convert to lowercase and tokenize
            word_freq.update(tokens)

        # vocabulary sorted by frequency (most common words first)
        vocabulary = [word for word, freq in word_freq.most_common()]

        self.word2idx = {word: idx for idx, word in enumerate(vocabulary)}
        self.idx2word = {idx: word for word, idx in self.word2idx.items()}
```

```python
# create an instance of the dataset class
dataset = TextDataset(sentences)

word2idx = dataset.word2idx
idx2word = dataset.idx2word
```

```python
print(list(word2idx.items())[0:10])
```

```
[('.', 0), ('the', 1), (',', 2), ('and', 3), ('i', 4), ('a', 5), ('it', 6), ('is', 7), ('to', 8), ('this', 9)]
```

```python
print(list(idx2word.items())[0:10])
```

```
[(0, '.'), (1, 'the'), (2, ','), (3, 'and'), (4, 'i'), (5, 'a'), (6, 'it'), (7, 'is'), (8, 'to'), (9, 'this')]
```

Build up vocabulary

```
from collections import Counter

# tokenizer function using split
def simple_tokenizer(text):
    return word_tokenize(text.lower())

# generator function to yield tokens
def yield_tokens(data_iter):
    for text in data_iter:
        yield simple_tokenizer(text)

# build vocabulary using Counter and add special tokens
def build_vocab_from_iterator(iterator, specials=None):
    word_freq = Counter()
    for tokens in iterator:
        word_freq.update(tokens)

    # sort words by frequency (most common first) and include special tokens
    sorted_vocab = specials + [word for word, freq in word_freq.most_common()]
    vocab = {word: idx for idx, word in enumerate(sorted_vocab)}

    return vocab

# build the vocabulary
vocab = build_vocab_from_iterator(yield_tokens(sentences), specials=["<pad>"])

# print the vocabulary and default index
print("Vocabulary:", vocab)
```

```
Vocabulary: {'<pad>': 0, '.': 1, 'the': 2, ',': 3, 'and': 4, 'i': 5, 'a': 6, 'it': 7, 'is': 8, 'to': 9,
```

```
len(vocab)
```

5354

Create train/valid/test batch datasets

```python
import torch
from torch.utils.data import TensorDataset, DataLoader

# assume max_sequence_length is a predefined constant
max_sequence_length = 32

# tokenize sentences using split() function
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]

# transform sentences to sequences of indices (list of lists of indices)
sentences_as_ints = [[vocab[token] for token in tokens] for tokens in tokenized_sentences]

# pad and truncate sequences
padded_sequences = []
for seq in sentences_as_ints:
    if len(seq) > max_sequence_length:
        seq = seq[:max_sequence_length]

    # pad sequence to max_sequence_length
    padding_length = max_sequence_length - len(seq)
    if padding_length > 0:
        seq.extend([0] * padding_length)  # extend with padding values
    padded_sequences.append(seq)

# convert list of lists to tensor
truncated_padded_sequences = torch.tensor(padded_sequences, dtype=torch.long)

# convert labels to tensor
labels_as_ints = torch.tensor(labels)

# create dataset
dataset = TensorDataset(truncated_padded_sequences, labels_as_ints)
```

```python
from torch.utils.data import random_split

# split the dataset into training, validation, and testing sets
train_size = int(0.8 * len(dataset))
val_size = int(0.1 * len(dataset))
test_size = len(dataset) - train_size - val_size

print(train_size)
print(val_size)
print(test_size)

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])

batch_size = 32

# create the dataloader for each set
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

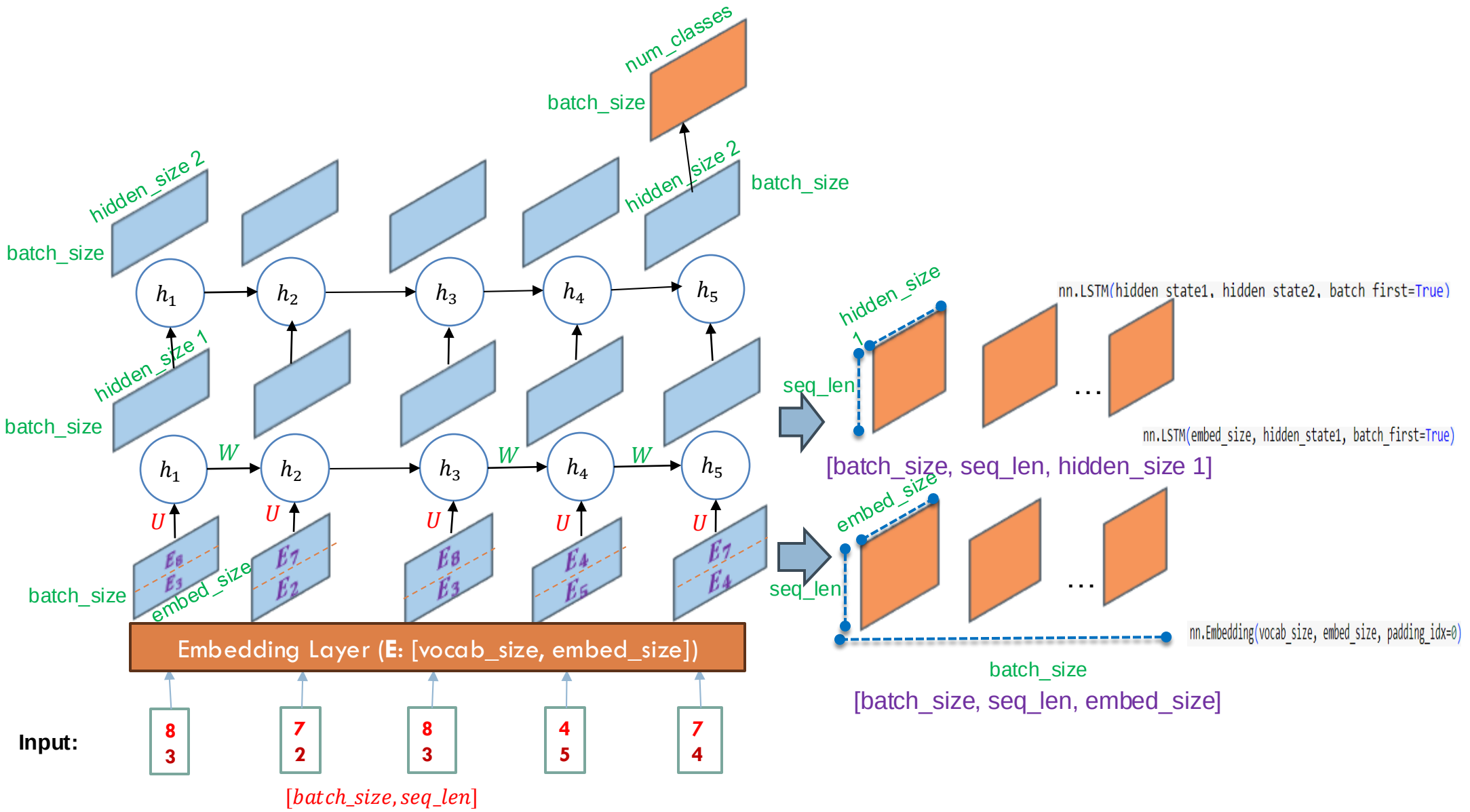
```
2400
300
300
```

```python
for (ts,l,si,tps,li) in zip(tokenized_sentences[0:3], labels[0:3], sentences_as_ints[0:3], truncated_padded_sequences[0:3], labels_as_ints[0:3]):
    print("Sentence: {}".format(ts))
    print("Label: {}".format(l))
    print("si: {}".format(si))
    print("tps: {}".format(tps))
    print("li: {}".format(li))
    print("\n")
```

```
Sentence: ['so', 'there', 'is', 'no', 'way', 'for', 'me', 'to', 'plug', 'it', 'in', 'here', 'in', 'the', 'us', 'unless', 'i', 'go', 'by', 'a', 'converter', '.']
Label: 0
si: [32, 45, 8, 64, 124, 15, 81, 9, 383, 7, 14, 70, 14, 2, 193, 654, 5, 89, 67, 6, 2235, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
tps: tensor([  32,   45,    8,   64,  124,   15,   81,    9,  383,    7,   14,   70,
          14,    2,  193,  654,    5,   89,   67,    6, 2235,    1,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0])
li: 0

Sentence: ['good', 'case', ',', 'excellent', 'value', '.']
Label: 1
si: [23, 163, 3, 104, 527, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
tps: tensor([ 23, 163,   3, 104, 527,   1,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0])
li: 1
```

Shape Transformation in RNNs



Build up/ train/ evaluate a RNN model

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.init as init

embed_size = 128

class Model(nn.Module):
    def __init__(self, vocab_size):
        super(Model, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size, padding_idx=0)

        self.lstm1 = nn.LSTM(embed_size, 128, batch_first=True)
        self.lstm2 = nn.LSTM(128, 128, batch_first=True)

        self.fc = nn.Linear(128, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x,(_,_) = self.lstm1(x)
        _,(x,_) = self.lstm2(x)
        x = self.fc(x.view(-1,128))
        return x.squeeze(1)

lstm_rnn = Model(len(vocab)).to(device)

criterion = nn.BCELoss()
optimizer = optim.Adam(lstm_rnn.parameters(), lr=0.001)
```

```
class BaseTrainer:
    def __init__(self, model, criterion, optimizer, train_loader, val_loader):
        self.model = model
        self.criterion = criterion  #the loss function
        self.optimizer = optimizer  #the optimizer
        self.train_loader = train_loader  #the train loader
        self.val_loader = val_loader  #the valid loader


    #the function to train the model in many epochs
    def fit(self, num_epochs):
        self.num_batches = len(self.train_loader)

        for epoch in range(num_epochs):
            print(f'Epoch {epoch + 1}/{num_epochs}')

            train_loss, train_accuracy = self.train_one_epoch()
            val_loss, val_accuracy = self.validate_one_epoch()
            print(
                f'{self.num_batches}/{self.num_batches} - loss: {train_loss:.4f} - accuracy: {train_accuracy:.4f} - val_loss: {val_loss:.4f} - val_accuracy: {val_accuracy:.4f}'
            )

    #train in one epoch
    def train_one_epoch(self):
        self.model.train()
        running_loss, correct, total = 0.0, 0, 0

        for i, (inputs, labels) in enumerate(self.train_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            self.optimizer.zero_grad()
            outputs = self.model(inputs)
            outputs = torch.sigmoid(outputs)
            loss = self.criterion(outputs.view(-1), labels.to(torch.float32))
            loss.backward()
            self.optimizer.step()

            running_loss += loss.item()
            predictions = torch.round(outputs)
            # Update the number of correct predictions and total samples
            correct += (predictions == labels).sum().item()
            total += labels.size(0)
        train_accuracy = correct / total
        train_loss = running_loss / self.num_batches
        return train_loss, train_accuracy

    #return the train_acc, train_loss, val_acc, val_loss
    #be called at the end of each epoch
    def validate_one_epoch(self):
        self.model.eval()
        val_loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
            for inputs, labels in self.val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = self.model(inputs)
                outputs = torch.sigmoid(outputs)
                loss = self.criterion(outputs.view(-1), labels.to(torch.float32))
                val_loss += loss.item()
                predictions = torch.round(outputs)
               # Update the number of correct predictions and total samples
                correct += (predictions == labels).sum().item()
                total += labels.size(0)
        val_accuracy = correct / total
        val_loss = val_loss / len(self.val_loader)
        return val_loss, val_accuracy
```

Build up/ train/ evaluate a RNN model

```
trainer = BaseTrainer(lstm_rnn, criterion, optimizer, train_loader, val_loader)
trainer.fit(num_epochs = 10)
```

```
Epoch 1/10
75/75 - loss: 0.0102 - accuracy: 0.9988 - val_loss: 1.6869 - val_accuracy: 0.7267
Epoch 2/10
75/75 - loss: 0.0099 - accuracy: 0.9988 - val_loss: 1.6668 - val_accuracy: 0.7467
Epoch 3/10
75/75 - loss: 0.0071 - accuracy: 0.9992 - val_loss: 1.6876 - val_accuracy: 0.7433
Epoch 4/10
75/75 - loss: 0.0071 - accuracy: 0.9992 - val_loss: 1.7502 - val_accuracy: 0.7467
Epoch 5/10
75/75 - loss: 0.0070 - accuracy: 0.9992 - val_loss: 1.7102 - val_accuracy: 0.7467
Epoch 6/10
75/75 - loss: 0.0070 - accuracy: 0.9992 - val_loss: 1.7054 - val_accuracy: 0.7467
Epoch 7/10
75/75 - loss: 0.0070 - accuracy: 0.9992 - val_loss: 1.7077 - val_accuracy: 0.7467
Epoch 8/10
75/75 - loss: 0.0070 - accuracy: 0.9992 - val_loss: 1.7666 - val_accuracy: 0.7467
Epoch 9/10
75/75 - loss: 0.0070 - accuracy: 0.9992 - val_loss: 1.7440 - val_accuracy: 0.7467
Epoch 10/10
75/75 - loss: 0.0071 - accuracy: 0.9992 - val_loss: 1.7359 - val_accuracy: 0.7467
```

```
correct, total = 0, 0
lstm_rnn.eval()
with torch.no_grad():
    for test_x, test_y in test_loader:
        test_x, test_y = test_x.to(device), test_y.to(device)
        test_outputs = lstm_rnn(test_x)
        test_outputs = torch.sigmoid(test_outputs)
        predictions = torch.round(test_outputs)
        # Update the number of correct predictions and total samples
        correct += (predictions == test_y).sum().item()
        total += test_y.size(0)
test_accuracy = correct / total

print(f'Test_acc: {test_accuracy:.4f}')
```

```
Test acc: 0.7767
```

Thanks for your attention!