1 Lecture 5 - Recurrent Neural Networks for Modeling Sequential Data

Outline

[1]:
```
from IPython.display import Image
%matplotlib inline
```

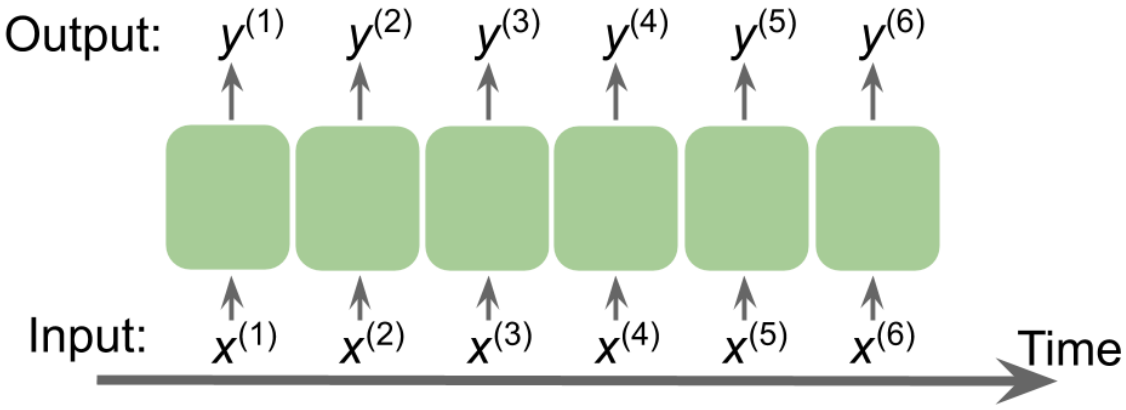
2 Introducing sequential data

2.1 Modeling sequential data–order matters

2.2 Representing sequences

[2]:
```
Image(filename='figures/15_01.png', width=500)
```

[2]:



The standard NN model covered so far, such as multilayer perceptrons (MLPs) assumes that the training examples are independent of each other and thus do not incorporate ordering information.

Recurrent Neural Networks (RNNs), by contrast, are designed for modeling sequences and are capable of remembering past information and processing new events accordingly, which is a clear advantage when working with sequence data.

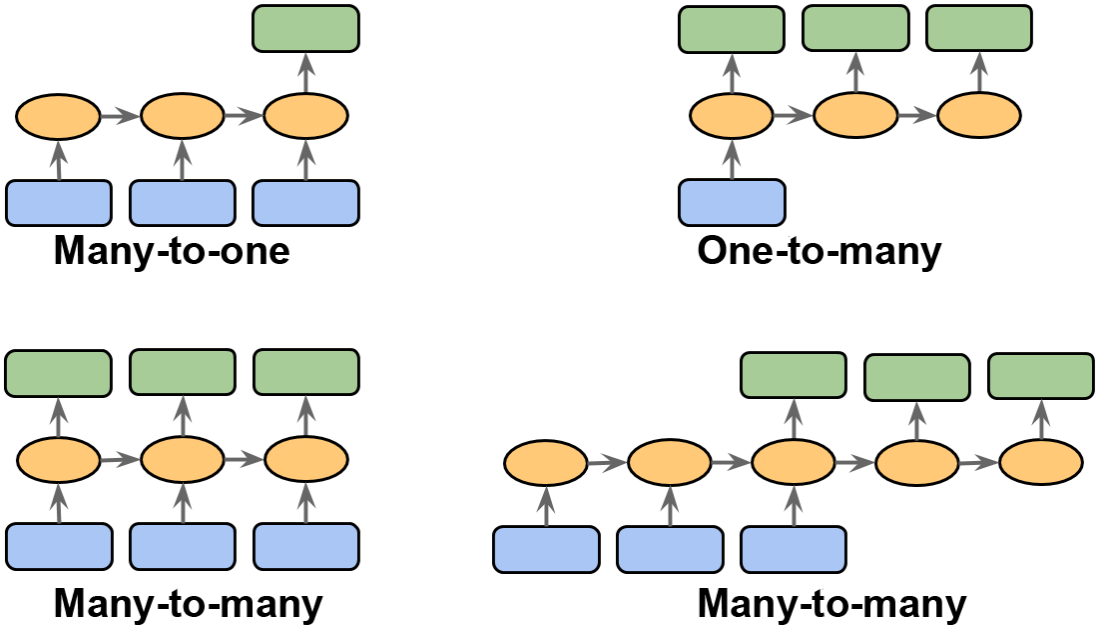
In finance, for examples, RNNs can be used to model a limit order book where the new trade orders are dependent on past trade orders.

2.3 The different categories of sequence modeling

If neither the input nor output data represent sequences, then we are dealing with standard data, and we could simply use a multilayer perceptron to model such data. However, if either the input or output is a sequence, the modeling task likely falls into one of these categories.

[3]:
```
Image(filename='figures/15_02.png', width=500)
```

[3]:



Notice that many-to-many can be further categorised into synchronous and asynchronous. Again, modeling a limit order book can fall into any of the above categories.

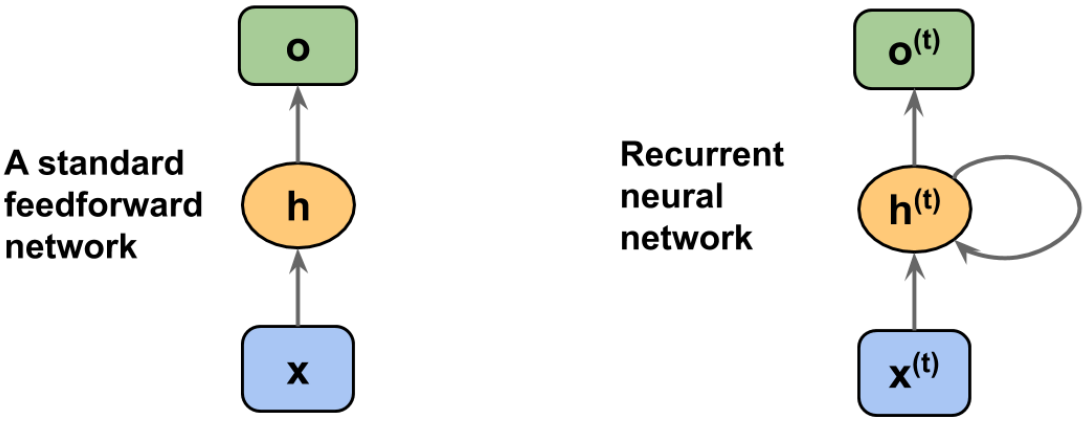
3 RNNs for modeling sequences

The typical structure of an RNN includes a recursive component to model sequence data. The two popular RNNs are long short term memory (LSTM) and gated recurrent units (GRUs).

3.1 Understanding the RNN looping mechanism

[4]:
```
Image(filename='figures/15_03.png', width=500)
```

[4]:



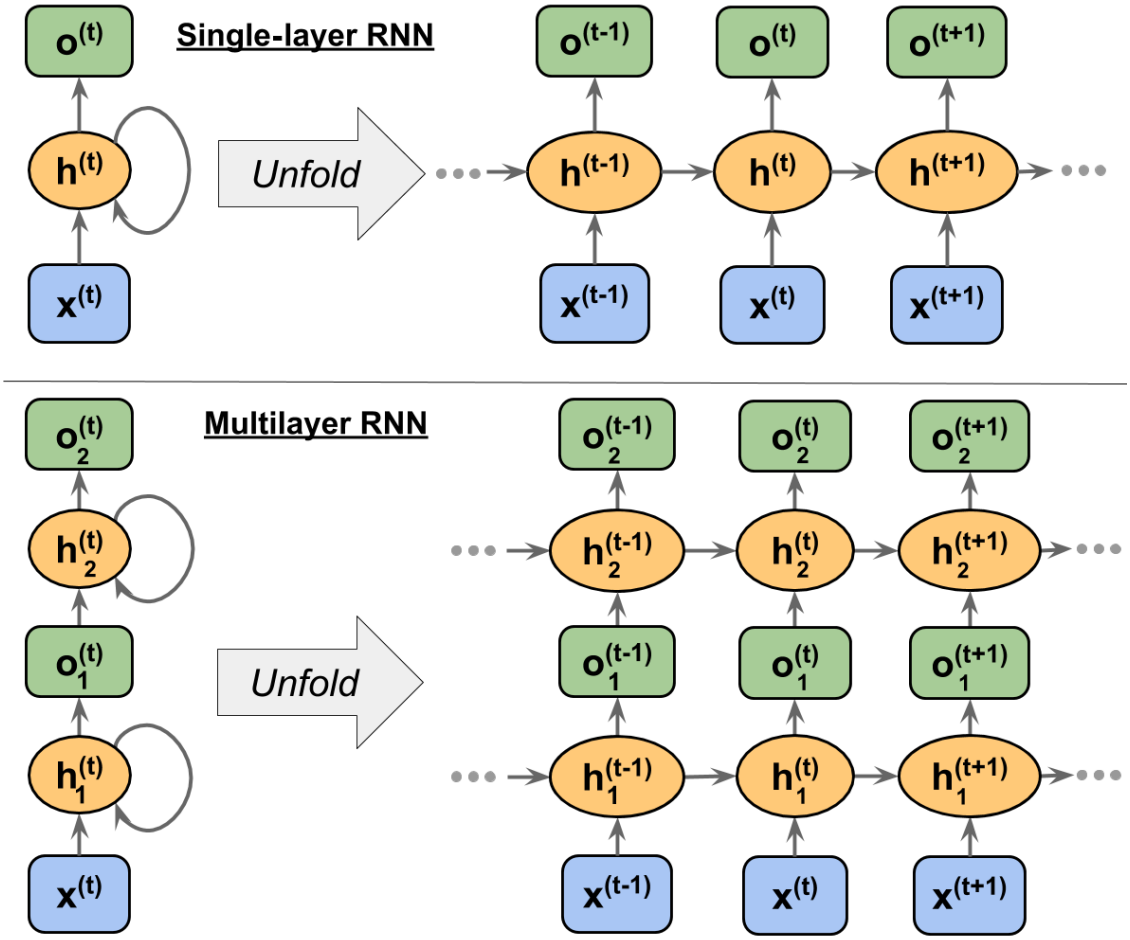
Above shows the dataflow in a standard feedforward NN and in an RNN side by side for comparison. Both of these networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer ($\mathbf{x}$), hidden layer ($\mathbf{h}$), and output layer ($\mathbf{o}$) are vectors that contain many units.

In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step.

The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general RNN architecture got its name.

[5]:
```
Image(filename='figures/15_04.png', width=500)
```

[5]:



Similar to multilayer perceptrons, RNNs can consist of multiple hidden layers. Note that it's a common convention to refer to RNNs with one hidden layer as a single-layer RNN.

Each hidden unit in a standard NN receives only one input—the net preactivation associated with the input layer. In contrast, each hidden unit in an RNN receives two distinct sets of input—the preactivation from the input layer and the activation of the same hidden layer from the previous time step, $t-1$.

At the first time step, $t=0$, the hidden units are initialized to zeros or small random values. Then, at a time step where $t>0$, the hidden units receive their input from the data point at the current time, $\mathbf{x}^{(t)}$, and the previous values of hidden units at $t-1$, indicated as $\mathbf{h}^{(t-1)}$.

In the case of a multilayer RNN, we can summarize the information flow as follows: - *layer=1*: Here, the the hidden layer is represented as $\mathbf{h}_1^{(t)}$ and it receives its input from the data point, $\mathbf{X}^{(t)}$, and the hidden values in the same layer, but at the previous time step, $\mathbf{h}_1^{(t-1)}$. - *layer=2*: The second hidden layer, $\mathbf{h}_2^{(t)}$, receives its inputs from the outputs of the layer below at the current time step ($\mathbf{o}_1^{(t)}$) and its own hidden values from the previous time step $\mathbf{h}_2^{(t-1)}$.

Since, in the case of multilayer RNNs, each recurrent layer must receive a sequence as input, all the

recurrent layers except the last one must return a sequence as output (that is, we will later have to set `return_sequences=True`). The behaviour of the last recurrent layer depends on the type of problem.

3.2 Computing activations in an RNN

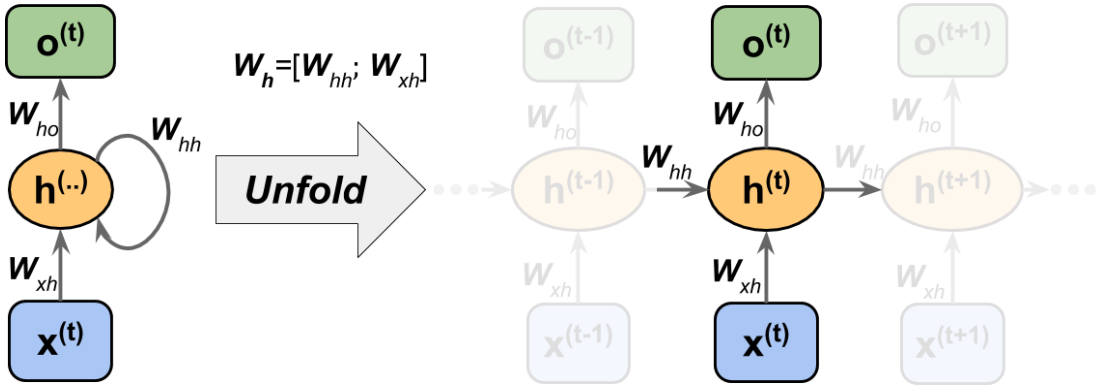
For simplicity, we consider just a single hidden layer; however, the same concept applies to multilayer RNNs.

Each directed edge in the representation of an RNN that we just looked at is associated with a weight matrix. Those weights do not depend on time, t; therefore, they are shared across the time axis. The different weight matrices in a single-layer RNN are as follows: - W_{xh} : The weight matrix between the input, $x^{(t)}$, and the hidden layer, $\mathbf{h}$ - W_{hh} : The weight matrix associated with the recurrent edge - W_{ho} : The weight matrix between the hidden layer and output layer

In certain implementations, the weight matrices, W_{xh} and W_{hh}, are concatenated to a combined matrix, W_h=[W_{xh};W_{hh}].

[6]:
```
Image(filename='figures/15_05.png', width=500)
```

[6]:

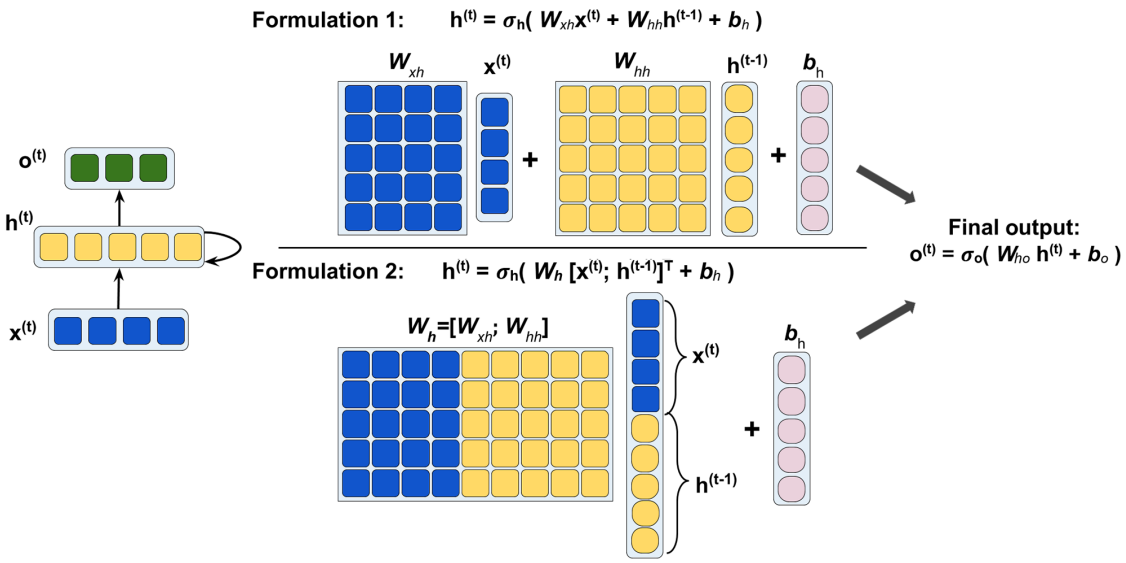


Computing the activations is very similar to standard multilayer perceptrons and other types of feedforward NNs.

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h. \tag{1}$$

[7]:
```
Image(filename='figures/15_06.png', width=700)
```

[7]:



For training RNNs, we use backpropagation through time (BPTT) algorithm. The derivation of the gradients might be a bit complicated, but the basic idea is that the overall loss, L, is the sum of all the loss functions at times $t = 1$ to $t = T$:

$$L = \sum_{t=1}^{T} L^{(t)}. \tag{2}$$

Since the loss at time t is dependent on the hidden units at all previous time steps 1 to t, the gradient will be computed as follows:

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \mathbf{o}^{(t)}} \times \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \times \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right). \tag{3}$$

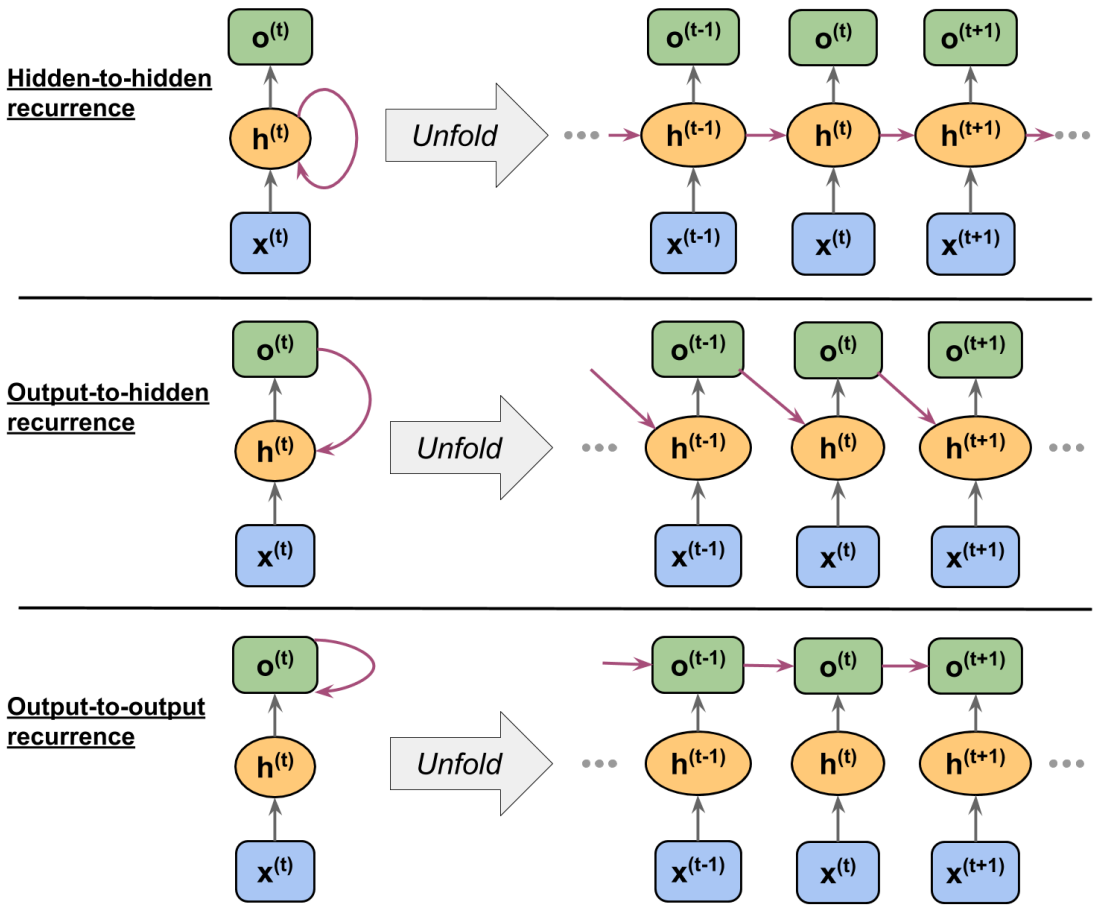
In the above, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$ is computed as a multiplication of adjacent time steps:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}. \tag{4}$$

3.3 Hidden recurrence versus output recurrence

[8]:
```
Image(filename='figures/15_07.png', width=500)
```

[8]:



Using the `torch.nn` module, a recurrent layer can be defined via RNN, which is similar to the hidden-to-hidden recurrence. In the following code, we will create a recurrent layer from RNN and perform a forward pass on an input sequence of length 3 to compute the output. We will also manually compute the forward pass and compare the results with those of RNN.

[9]:
```
import torch
import torch.nn as nn

torch.manual_seed(1)

# create a layer
rnn_layer = nn.RNN(input_size=5, hidden_size=2, num_layers=1, batch_first=True)

w_xh = rnn_layer.weight_ih_l0
w_hh = rnn_layer.weight_hh_l0
b_xh = rnn_layer.bias_ih_l0
b_hh = rnn_layer.bias_hh_l0
```

```
print('W_xh shape:', w_xh.shape)
print('W_hh shape:', w_hh.shape)
print('b_xh shape:', b_xh.shape)
print('b_hh shape:', b_hh.shape)
```

```
W_xh shape: torch.Size([2, 5])
W_hh shape: torch.Size([2, 2])
b_xh shape: torch.Size([2])
b_hh shape: torch.Size([2])
```

The input shape for this layer is `(batch_size, sequence_length, 5)`, where the first dimension is the batch dimension (as we set `batch_first=True`), the second dimension corresponds to the sequence, and the last dimension corresponds to the features.

Notice that we will output a sequence, which, for an input sequence of length 3, will result in the output sequence $\langle \mathbf{o}^{(0)}, \mathbf{o}^{(1)}, \mathbf{o}^{(2)} \rangle$.

Also, RNN uses one layer by default, and we can set `num_layers` to stack multiple RNN layers together to form a stacked RNN. Next, we call the forward pass on the `rnn_layer` and manually compute the outputs at each time step and compare them.

[10]:
```
x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()

## output of the simple RNN:
output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))

## manually computing the output:
out_man = []
for t in range(3):
    xt = torch.reshape(x_seq[t], (1, 5))
    print(f'Time step {t} =>')
    print('   Input           :', xt.numpy())

    ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
    print('   Hidden          :', ht.detach().numpy())

    if t>0:
        prev_h = out_man[t-1]
    else:
        prev_h = torch.zeros((ht.shape))

    ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) + b_hh
    ot = torch.tanh(ot)
    out_man.append(ot)
    print('   Output (manual) :', ot.detach().numpy())
    print('   RNN output      :', output[:, t].detach().numpy())
    print()
```

```
Time step 0 =>
   Input           : [[1. 1. 1. 1. 1.]]
```

```
   Hidden           : [[-0.4701929  0.5863904]]
   Output (manual)  : [[-0.3519801   0.52525216]]
   RNN output       : [[-0.3519801   0.52525216]]

Time step 1 =>
   Input            : [[2. 2. 2. 2. 2.]]
   Hidden           : [[-0.88883156  1.2364397 ]]
   Output (manual)  : [[-0.68424344  0.76074266]]
   RNN output       : [[-0.68424344  0.76074266]]

Time step 2 =>
   Input            : [[3. 3. 3. 3. 3.]]
   Hidden           : [[-1.3074701  1.886489 ]]
   Output (manual)  : [[-0.8649416   0.90466356]]
   RNN output       : [[-0.8649416   0.90466356]]
```

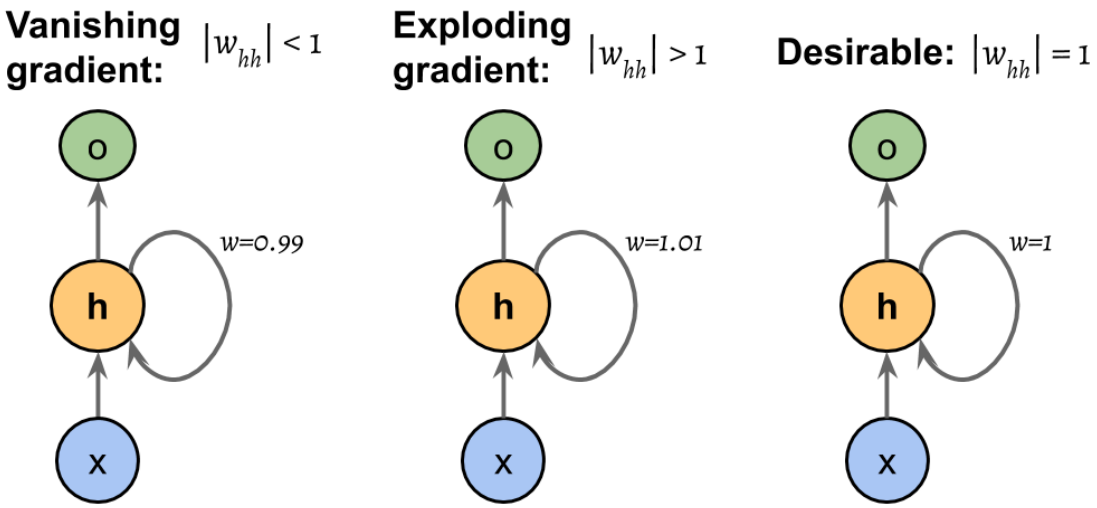
The hyperbolic tangent (`tanh`) activation function is the default activation used in `RNN`.

3.4 The challenges of learning long-range interactions

Because of the multiplicative factor, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$, in computing the gradients of a loss function, the so-called **vanishing** and **exploding** gradient problems arise.

[11]:
```
Image(filename='figures/15_08.png', width=500)
```

[11]:



As $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$ has $t \check{} k$ multiplications; therefore, multiplying the weight, w, by itself $t \check{} k$ times results in a factor, $w^{t \check{} k}$. As a result, if $|w| < 1$, this factor becomes very small when $t \check{} k$ is large. On the other hand, if the weight of the recurrent edge is $|w| > 1$, then $w^{t \check{} k}$ becomes very large when $t \check{} k$ is large.

Note that a large $t \check{} k$ refers to long-range dependencies. A naive solution to avoid vanishing or exploding gradients can be reached by ensuring $|w| = 1$.

In practice, there are at least three solutions to this problem: - Gradient clipping - Truncated backpropagation through time (TBPTT) - Long short-term memory cells (LSTM)

While both gradient clipping and TBPTT can solve the exploding gradient problem, the truncation limits the number of steps that the gradient can effectively flow back and properly update the weights.

On the other hand, LSTM, is more successful in vanishing and exploding gradient problems while modeling long-range dependencies through the use of memory cells.

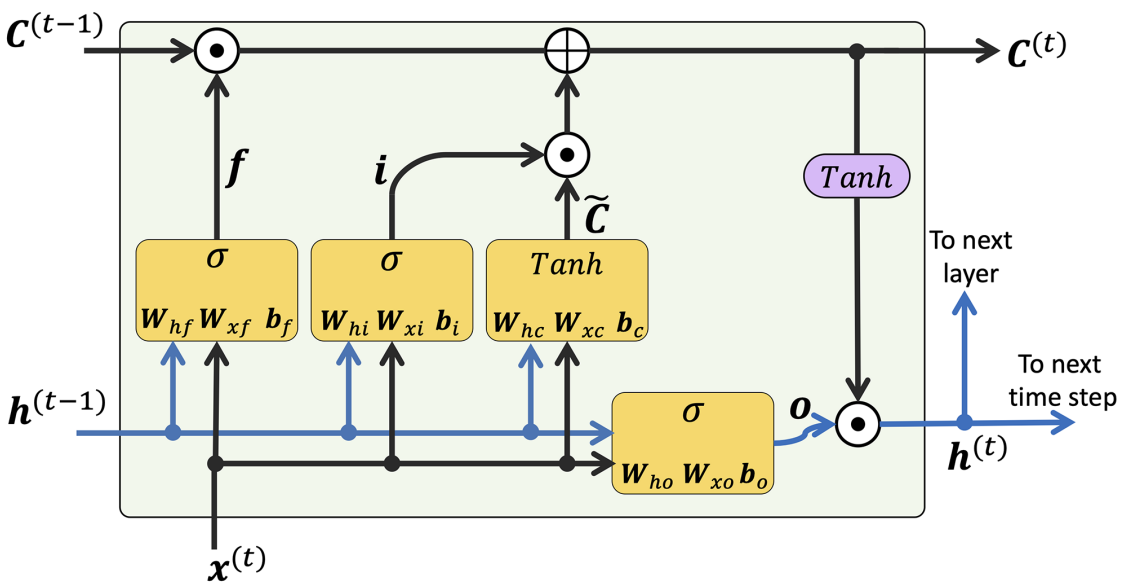
3.5 Long Short-Term Memory cells

The building block of an LSTM is a **memory cell**, which essentially represents or replaces the hidden layer of standard RNNs.

In each memory cell, there is a recurrent edge that has the desirable weight, $w = 1$, to overcome the vanishing and exploding gradient problems. The values associated with this recurrent edge are collectively called the **cell state**. The unfolded structure of a modern LSTM cell is as below.

[12]:
```
Image(filename='figures/15_09.png', width=500)
```

[12]:



The cell state from the previous time step, $C^{(t \check{} 1)}$, is modified to get the cell state at the current time step, $C^{(t)}$, without being multiplied directly by any weight factor. The flow of information in this memory cell is controlled by several computation units, called *gates*.

In the above figure, $\odot$ refers to the element-wise product (element-wise multiplication) and $\oplus$ means element-wise summation (element-wise addition). Furthermore, $x^{(t)}$ refers to the input data at time t, and $h^{(t \check{} 1)}$ indicates the hidden units at time $t \check{} 1$.

Four boxes are indicated with an activation function, either the sigmoid function (σ) or tanh, and a set of weights. These boxes apply a linear combination by performing matrix-vector multiplications on their inputs (which are $h^{(t\check{}1)}$ and $x^{(t)}$). These units of computation with sigmoid activation functions, whose output units are passed through $\odot$, are called gates.

In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate, and the output gate:

The **forget gate** ($\mathbf{f}_t$) allows the memory cell to reset the cell state without growing indefinitely. Basically, the forget gate decides which information is allowed to go through and which information to suppress. It is computed as follows

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_{xf}\mathbf{x}^t + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f\right). \tag{5}$$

The **input gate** ($\mathbf{i}_t$) and **candidate value** ($\widetilde{\mathbf{C}}_t$) are responsible for updating the cell state. They are computed as follows:

$$\mathbf{i}_t = \sigma\left(\mathbf{W}_{xi}\mathbf{x}^t + \mathbf{W}_{hi}\mathbf{h}^{(t-1)} + \mathbf{b}_i\right) \tag{6}$$

$$\widetilde{\mathbf{C}}_t = \tanh\left(\mathbf{W}_{xc}\mathbf{x}^t + \mathbf{W}_{hc}\mathbf{h}^{(t-1)} + \mathbf{b}_c\right). \tag{7}$$

The cell state at time t is computed as follows:

$$\mathbf{C}^{(t)} = \left(\mathbf{C}^{(t-1)} \odot \mathbf{f}_t\right) \oplus \left(\mathbf{i}_t \odot \widetilde{\mathbf{C}}_t\right). \tag{8}$$

The **output gate** ($\mathbf{o}_t$) decides how to update the values of hidden units:

$$\mathbf{o}_t = \sigma\left(\mathbf{W}_{xo}\mathbf{x}^t + \mathbf{W}_{ho}\mathbf{h}^{(t-1)} + \mathbf{b}_o\right). \tag{9}$$

Given this, the hidden units at the current time step are computed as follows:

$$\mathbf{h}^{(t)} = \mathbf{o}_t \odot \tanh(\mathbf{C}^{(t)}). \tag{10}$$

Next we apply RNNs and LSTMs to real-world datasets.

4 Implementing RNNs for sequence modeling in PyTorch

4.1 Predicting the sentiment of IMDb movie reviews

The sentiment analysis (language modeling) is concerned with analyzing the expressed opinion of a sentence or a text document. This can be very useful in creating a sentiment score for an exchange-listed firm or even a financial index by creating a sentiment score based on news articles and analyst reports. This is typically an approach which is used to automate the economicy policy sentiment indices such as the one developed by Baker and Bloom: https://www.policyuncertainty.com/

As any sentiment score would be based on a long sequence of text inputs, we implement a many-to-many RNN.

4.1.1 Preparing the movie review data

[1]:
```
import torch
import torch.nn as nn
from IPython.display import Image
%matplotlib inline
```

Each set has 25,000 samples. And each sample of the datasets consists of two elements, the sentiment label representing the target label we want to predict (`neg` refers to negative sentiment and `pos` refers to positive sentiment), and the movie review text (the input features). The text component of these movie reviews is sequences of words, and the RNN model classifies each sequence as a positive (`1`) or negative (`0`) review.

Before we feed the data into an RNN model, we need to apply several preprocessing steps: 1. Split the training dataset into separate training and validation partitions. 2. Identify the unique words in the training dataset 3. Map each unique word to a unique integer and encode the review text into encoded integers (an index of each unique word) 4. Divide the dataset into mini-batches as input to the model

First, we import the necessary modules and read the data from `torchtext` (which we install via `pip install torchtext`). To import data, we need to install `torchdata` as well.

[2]:
```
import torchdata
from torchtext.datasets import IMDB
from torch.utils.data.dataset import random_split

# Step 1: load and create the datasets

train_dataset = IMDB(split='train')
test_dataset = IMDB(split='test')

torch.manual_seed(1)
train_dataset, valid_dataset = random_split(list(train_dataset), [20000, 5000])
```

The original training dataset contains 25,000 examples. 20,000 examples are randomly chosen for training, and 5,000 for validation.

To prepare the data for input to an NN, we need to encode it into numeric values. To do this, we first find the unique words (tokens) in the training dataset. It is more efficient to use the `Counter` class from the `collections` package, which is part of Python's standard library.

In the following code, we instantiate a new `Counter` object (`token_counts`) that collects the unique word frequencies. Note that in this particular application (and in contrast to the `bag-of-words` model), we are only interested in the set of unique words and won't require the word counts, which are created as a side product. In the `bag-of-words` model, we would have required to keep track of word count as well.

The `tokenizer` function removes HTML markups as well as punctuation and other non-letter characters.

[4]:
```
## Step 2: find unique tokens (words)
import re
from collections import Counter, OrderedDict
```

```
token_counts = Counter()

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) +\
        ' '.join(emoticons).replace('-', '')
    tokenized = text.split()
    return tokenized


for label, line in train_dataset:
    tokens = tokenizer(line)
    token_counts.update(tokens)


print('Vocab-size:', len(token_counts))
```

```
Vocab-size: 69023
```

We map each unique word to a unique integer. This can be done manually using a Python dictionary, where the keys are the unique tokens (words) and the value associated with each key is a unique integer. However, the `torchtext` package already provides a class, `Vocab`, which we use to create such a mapping and encode the entire dataset.

First, we create a `vocab` object by passing the ordered dictionary mapping tokens to their corresponding occurrence frequencies (the ordered dictionary is the sorted `token_counts`). Second, we prepend two special tokens to the vocabulary – the padding and the unknown token.

To demonstrate how to use the `vocab` object, we convert an example input text into a list of integer values.

[5]:
```
## Step 3: encoding each unique token into integers
from torchtext.vocab import vocab

sorted_by_freq_tuples = sorted(token_counts.items(), key=lambda x: x[1],␣
 ↪reverse=True)
ordered_dict = OrderedDict(sorted_by_freq_tuples)

vocab = vocab(ordered_dict)

vocab.insert_token("<pad>", 0)
vocab.insert_token("<unk>", 1)
vocab.set_default_index(1)

print([vocab[token] for token in ['bad', 'movie', 'terrible', 'watch']])
```

```
[78, 18, 389, 104]
```

Note that there might be some tokens in the validation or testing data that are not present in the training data and are thus not included in the mapping. If we have q tokens (that is, the size of `token_counts` passed to `Vocab`, which in this case is 69,023), then all tokens that haven't been seen before, and are thus not included in `token_counts`, will be assigned the integer `1` (a placeholder for the unknown token). In other words, the index `1` is reserved for unknown words. Another reserved value is the integer `0`, which serves as a placeholder, a so-called padding token, for adjusting the sequence length. This is handy when we are building an RNN model in PyTorch.

Next we define the `text_pipeline` function to transform each text in the dataset accordingly and the `label_pipeline` function to convert each label to 1 or 0.

[6]:
```
## Step 3-A: define the functions for transformation

# device = torch.device("cuda:0") # use this when GPU on the computing device
device = 'cpu'

text_pipeline = lambda x: [vocab[token] for token in tokenizer(x)]
label_pipeline = lambda x: 1. if x == 'pos' else 0.
```

We generate batches of samples using `DataLoader` and pass the data processing pipelines declared previously to the argument `collate_fn`. We wrap the text encoding and label transformation function into the `collate_batch` function.

[7]:
```
## Step 3-B: wrap the encode and transformation function
def collate_batch(batch):
    label_list, text_list, lengths = [], [], []
    for _label, _text in batch:
        label_list.append(label_pipeline(_label))
        processed_text = torch.tensor(text_pipeline(_text),
                                      dtype=torch.int64)
        text_list.append(processed_text)
        lengths.append(processed_text.size(0))
    label_list = torch.tensor(label_list)
    lengths = torch.tensor(lengths)
    padded_text_list = nn.utils.rnn.pad_sequence(text_list, batch_first=True)
    return padded_text_list.to(device), label_list.to(device), lengths.to(device)
```

Above we convert sequences of words into sequences of integers, and labels of `pos` or `neg` into 1 or 0. However, the sequences currently have different lengths (as shown in the result of executing the following code for four examples). Although, in general, RNNs can handle sequences with different lengths, we still need to make sure that all the sequences in a mini-batch have the same length to store them efficiently in a tensor.

PyTorch provides an efficient method, `pad_sequence()`, which automatically pads the consecutive elements that are to be combined into a batch with placeholder values (0s) so that all sequences within a batch will have the same shape.

In the previous code, we already created a data loader of a small batch size from the training dataset and applied the `collate_batch` function, which itself included a `pad_sequence()` call.

[19]:
```
## Take a small batch

from torch.utils.data import DataLoader
dataloader = DataLoader(train_dataset, batch_size=4, shuffle=False,␣
 ↪collate_fn=collate_batch)
text_batch, label_batch, length_batch = next(iter(dataloader))
print(text_batch)
print(label_batch)
print(length_batch)
print(text_batch.shape)
```

```
tensor([[   35,  1739,     7,   449,   721,     6,   301,     4,   787,     9,
             4,    18,    44,     2,  1705,  2460,   186,    25,     7,    24,
           100,  1874,  1739,    25,     7, 34415,  3568,  1103,  7517,   787,
             5,     2,  4991, 12401,    36,     7,   148,   111,   939,     6,
         11598,     2,   172,   135,    62,    25,  3199,  1602,     3,   928,
          1500,     9,     6,  4601,     2,   155,    36,    14,   274,     4,
         42945,     9,  4991,     3,    14, 10296,    34,  3568,     8,    51,
           148,    30,     2,    58,    16,    11,  1893,   125,     6,   420,
          1214,    27, 14542,   940,    11,     7,    29,   951,    18,    17,
         15994,   459,    34,  2480, 15211,  3713,     2,   840,  3200,     9,
          3568,    13,   107,     9,   175,    94,    25,    51, 10297,  1796,
            27,   712,    16,     2,   220,    17,     4,    54,   722,   238,
           395,     2,   787,    32,    27,  5236,     3,    32,    27,  7252,
          5118,  2461,  6390,     4,  2873,  1495,    15,     2,  1054,  2874,
           155,     3,  7015,     7,   409,     9,    41,   220,    17,    41,
           390,     3,  3925,   807,    37,    74,  2858,    15, 10297,   115,
            31,   189,  3506,   667,   163,     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,     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,     0],
        [  216,   175,   724,     5,    11,    18,    10,   226,   110,    14,
           182,    78,     8,    13,    24,   182,    78,     8,    13,   166,
           182,    50,   150,    24,    85,     2,  4031,  5935,   107,    96,
            28,  1867,   602,    19,    52,   162,    21,  1698,     8,     6,
          1181,   367,     2,   351,    10,   140,   419,     4,   333,     5,
          6022,  7136,  5055,  1209, 10892,    32,   219,     9,     2,   405,
          1413,    13,  4031,    13,  1099,     7,    85,    19,     2,    20,
          1018,     4,    85,   565,    34,    24,   807,    55,     5,    68,
           658,    10,   507,     8,     4,   668,     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,     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,     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,     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,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0],
[  10,   121,    24,    28,    98,    74,   589,     9,   149,     2,
 7372,  3030, 14543,  1012,   520,     2,   985,  2327,     5, 16847,
 5479,    19,    25,    67,    76,  3478,    38,     2,  7372,     3,
   25,    67,    76,  2951,    34,    35, 10893,   155,   449, 29495,
23725,    10,    67,     2,   554,    12, 14543,    67,    91,     4,
   50,    20,    19,     8,    67,    24,  4228,     2,  2142,    37,
   33,  3478,    87,     3,  2564,   160,   155,    11,   634,   126,
   24,   158,    72,   286,    13,   373,     2,  4804,    19,     2,
 7372,  6794,     6,    30,   128,    73,    48,    10,   886,     8,
   13,    24,     4,    85,    20,    19,     8,    13,    35,   218,
    3,   428,   710,     2,   107,   936,     7,    54,    72,   223,
    3,    10,    96,   122,     2,   103,    54,    72,    82,     2,
  658,   202,     2,   106,   293,   103,     7,  1193,     3,  3031,
  708,  5760,     3,  2918,  3991,   706,  3327,   349,   148,   286,
   13,   139,     6,     2,  1501,   750,    29,  1407,    62,    65,
 2612,    71,    40,    14,     4,   547,     9,    62,     8,  7943,
   71,    14,     2,  5687,     5,  4868,  3111,     6,   205,     2,
   18,    55,  2075,     3,   403,    12,  3111,   231,    45,     5,
  271,     3,    68,  1400,     7,  9774,   932,    10,   102,     2,
   20,   143,    28,    76,    55,  3810,     9,  2723,     5,    12,
   10,   379,     2,  7372,    15,     4,    50,   710,     8,    13,
   24,   887,    32,    31,    19,     8,    13,   428],
[18923,     7,     4,  4753,  1669,    12,  3019,     6,     4, 13906,
  502,    40,    25,    77,  1588,     9,   115,     6, 21713,     2,
   90,   305,   237,     9,   502,    33,    77,   376,     4, 16848,
  847,    62,    77,   131,     9,     2,  1580,   338,     5, 18923,
   32,     2,  1980,    49,   157,   306, 21713,    46,   981,     6,
10298,     2, 18924,   125,     9,   502,     3,   453,     4,  1852,
  630,   407,  3407,    34,   277,    29,   242,     2, 20200,     5,
18923,    77,    95,    41,  1833,     6,  2105,    56,     3,   495,
  214,   528,     2,  3479,     2,   112,     7,   181,  1813,     3,
  597,     5,     2,   156,   294,     4,   543,   173,     9,  1562,
  289, 10038,     5,     2,    20,    26,   841,  1392,    62,   130,
  111,    72,   832,    26,   181, 12402,    15,    69,   183,     6,
   66,    55,   936,     5,     2,    63,     8,     7,    43,     4,
   78, 23726, 15995,    13,    20,    17,   800,     5,   392,    59,
 3992,     3,   371,   103,  2596,     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,     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,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0]])
tensor([0., 0., 0., 0.])
tensor([165,  86, 218, 145])
torch.Size([4, 218])
```

As we can see above, after the padding has been applied, the batch has all 4 examples of length 218.

Finally, we divide all three datasets into data loaders with a batch size of 32.

[20]:
```
## Step 4: batching the datasets

batch_size = 32

train_dl = DataLoader(train_dataset, batch_size=batch_size,
                      shuffle=True, collate_fn=collate_batch)
valid_dl = DataLoader(valid_dataset, batch_size=batch_size,
                      shuffle=False, collate_fn=collate_batch)
test_dl = DataLoader(test_dataset, batch_size=batch_size,
                     shuffle=False, collate_fn=collate_batch)
```

Now, the data is in a suitable format for an RNN model.

We first discuss feature **embedding**, which is an optional but highly recommended preprocessing step that is used to reduce the dimensionality of the word vectors. Otherwise the input dimension could be unnecessarily large (think of dimension being the length of the input examples vocabulary), which can slow down the training or even result in a bad model fit.

4.1.2 Embedding layers for sentence encoding

During the data preparation in the previous step, we generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the indices of unique words. These word indices can be converted into input features in several different ways.

One naive way is to apply one-hot encoding to convert the indices into vectors of zeros and ones. Then, each word will be mapped to a vector whose size is the number of unique words in the entire dataset. Given that the number of unique words (the size of the vocabulary) can be in the order of 10^4 – 10^5, which will also be the number of our input features, a model trained on such features may suffer from the **curse of dimensionality**. Furthermore, these features are very sparse since all are zero except one.

A more elegant approach is to map each word to a vector of a fixed size with real-valued elements (not necessarily integers). In contrast to the one-hot encoded vectors, we can use finite-sized vectors to represent an infinite number of real numbers. (In theory, we can extract infinite real numbers from a given interval, for example [–1, 1].)

This is the idea behind embedding, which is a feature-learning technique that we utilize here to automatically learn the salient features to represent the words in our dataset.

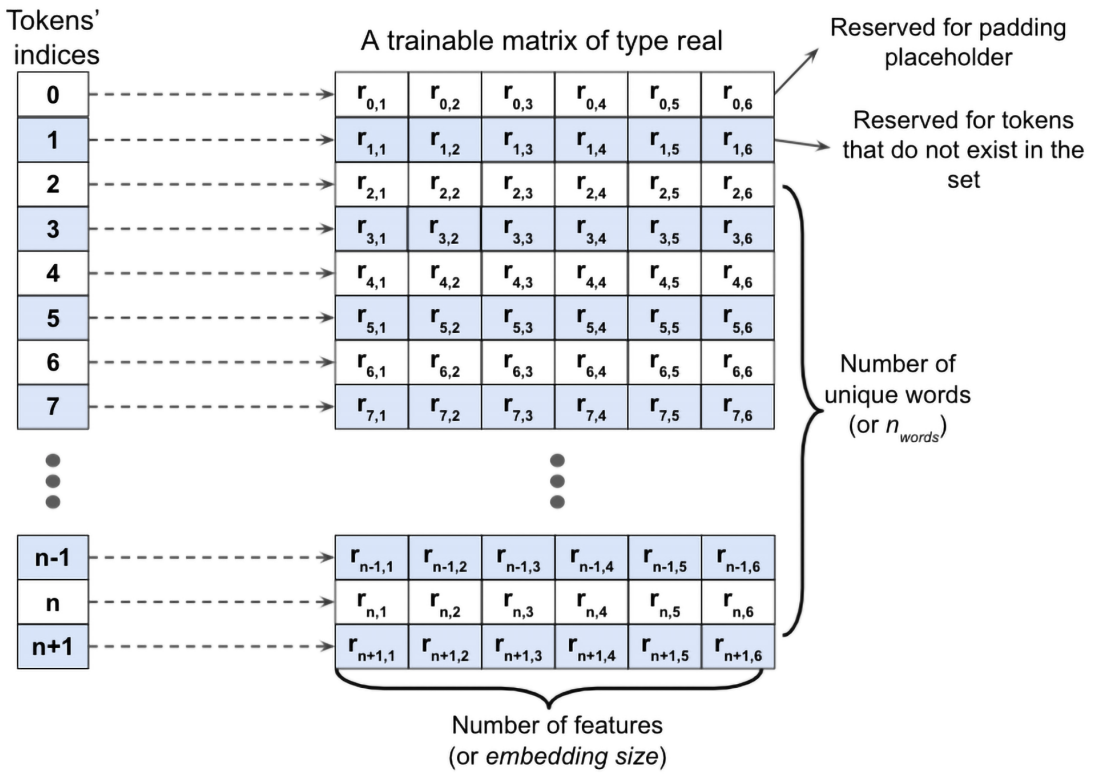
Given the number of unique words, n_{words}, we can select the size of the embedding vectors (a.k.a.,

embedding dimension) to be much smaller than the number of unique words ($embedding\_dim << n_{words}$) to represent the entire vocabulary as input features.

The advantages of embedding over one-hot encoding are as follows: - A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality - The extraction of salient features since the embedding layer in an NN can be optimized (or learned)

[4]:
```
Image(filename='figures/15_10.png', width=600)
```

[4]:



Given a set of tokens of size $n_{words}+2$ (n_{words}) is the size of the token set, plus index 0 is reserved for the padding placeholder, and 1 is for the words not present in the token set), an embedding matrix of size $(n_{words} + 2) \times embedding\_dim$ will be created where each row of this matrix represents numeric features associated with a token. Therefore, when an integer index, i, is given as input to the embedding, it will look up the corresponding row of the matrix at index i and return the numeric features.

The embedding matrix serves as the input layer to the NN models. In practice, creating an embedding layer can simply be done using `nn.Embedding`. Let's see an example where we create an embedding layer and apply it to a batch of two samples.

[13]:
```
embedding = nn.Embedding(num_embeddings=10,
                         embedding_dim=4,
                         padding_idx=0)
```

```
# a batch of 2 samples of 5 indices each
text_encoded_input = torch.LongTensor([[1,2,4,5,6],[4,3,2,6,0]])
print(embedding(text_encoded_input))
```

```
tensor([[[-1.6155,  0.7387,  0.5137, -0.2635],
         [-2.6622, -0.8662, -0.6131, -0.4082],
         [-1.1674, -1.3222, -0.3867,  2.0638],
         [-0.9681,  1.5224, -0.8617, -0.3623],
         [-0.6853, -0.3833, -1.3204, -1.0817]],

        [[-1.1674, -1.3222, -0.3867,  2.0638],
         [-1.8212, -0.9495, -1.0268, -1.0582],
         [-2.6622, -0.8662, -0.6131, -0.4082],
         [-0.6853, -0.3833, -1.3204, -1.0817],
         [ 0.0000,  0.0000,  0.0000,  0.0000]]], grad_fn=<EmbeddingBackward0>)
```

The input to this model (embedding layer) must have rank 2 with the dimensionality $batchsize \times input\_length$, where $input\_length$ is the length of sequences (here, 5).

For example, an input sequence in the mini-batch could be $< 1, 8, 5, 9, 2 >$, where each element of this sequence is the index of the unique words. The output will have the dimensionality $batchsize \times input\_length \times embedding\_dim$, where $embedding\_dim$ is the size of the embedding features (here, set to 4).

The other argument provided to the embedding layer, `num_embeddings`, corresponds to the unique integer values that the model will receive as input (for instance, $n + 2$, set here to 10). Therefore, the embedding matrix in this case has the size 10×4.

`padding_idx` indicates the token index for padding (here, 0), which, if specified, will not contribute to the gradient updates during training. In our example, the length of the original sequence of the second sample is 4, and we padded it with 1 more element 0. The embedding output of the padded element is [0, 0, 0, 0].

4.1.3 Building an RNN model

Now we are ready to build an RNN model. Using the `nn.Module` class, we can combine the embedding layer, the recurrent layers of the RNN, and the fully connected non-recurrent layers. For the recurrent layers, we can use any of the following implementations:

- **RNN layers:**
  - `nn.RNN(input_size, hidden_size, num_layers=1)`
  - `nn.LSTM(..)`
  - `nn.GRU(..)`
  - `nn.RNN(input_size, hidden_size, num_layers=1, bidirectional=True)`

To see how a multilayer RNN model can be built using one of these recurrent layers, in the following example, we create an RNN model with two recurrent layers of type RNN. Finally, we add a non-recurrent fully connected layer as the output layer, which returns a single output value as the prediction.

[15]:
```
## An example of building a RNN model
## with simple RNN layer

# Fully connected neural network with one hidden layer
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.rnn = nn.RNN(input_size,
                          hidden_size,
                          num_layers=2,
                          batch_first=True)
        #self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        #self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, 1) # output dimension is 1

    def forward(self, x):
        _, hidden = self.rnn(x)
        out = hidden[-1, :, :] # take the output of last recurrent hidden layer
        out = self.fc(out)
        return out

model = RNN(64, 32)

print(model)

model(torch.randn(10, 3, 64)) # Recall the input shape is (batch_size,␣
 ↪sequence_length, features)
```

```
RNN(
  (rnn): RNN(64, 32, num_layers=2, batch_first=True)
  (fc): Linear(in_features=32, out_features=1, bias=True)
)
```

[15]:
```
tensor([[ 0.2048],
        [-0.0966],
        [ 0.1329],
        [-0.2409],
        [ 0.0342],
        [-0.0525],
        [ 0.0878],
        [-0.6212],
        [-0.0543],
        [-0.0429]], grad_fn=<AddmmBackward0>)
```

4.1.4 Building an RNN model for the sentiment analysis task

Since we have very long sequences in the movie reviews, we use an LSTM layer to account for long-range effects. We create an RNN model for sentiment analysis, starting with an embedding layer producing word embeddings of feature size 20 (`embed_dim=20`).

Then, a recurrent layer of type LSTM will be added. Finally, we add a fully connected layer as a hidden layer and another fully connected layer as the output layer, which returns a single class-membership probability value via the logistic sigmoid activation as the prediction.

[24]:
```
class RNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size,
                                      embed_dim,
                                      padding_idx=0)
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
                           batch_first=True)  # checks for the first dimension␣
↪being the batch size
        self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(fc_hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, text, lengths):
        out = self.embedding(text)
        out = nn.utils.rnn.pack_padded_sequence(out, lengths.cpu().numpy(),␣
↪enforce_sorted=False, batch_first=True)
        out, (hidden, cell) = self.rnn(out) # note the cell output from LSTM
        out = hidden[-1, :, :]
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.sigmoid(out)
        return out

vocab_size = len(vocab)
embed_dim = 10
rnn_hidden_size = 32
fc_hidden_size = 32

torch.manual_seed(1886)
model = RNN(vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size)
model = model.to(device)
```

Next we develop the `train` function to train the model on the given dataset for one epoch and return the classification accuracy and loss.

[25]:
```
def train(dataloader):
    model.train()
    total_acc, total_loss = 0, 0
    for text_batch, label_batch, lengths in dataloader:
        optimizer.zero_grad()
        pred = model(text_batch, lengths)[:, 0]
        loss = loss_fn(pred, label_batch)
        loss.backward()
        optimizer.step()
        total_acc += ((pred>=0.5).float() == label_batch).float().sum().item()
        total_loss += loss.item()*label_batch.size(0)
    return total_acc/len(dataloader.dataset), total_loss/len(dataloader.dataset)

def evaluate(dataloader):
    model.eval()
    total_acc, total_loss = 0, 0
    with torch.no_grad():
        for text_batch, label_batch, lengths in dataloader:
            pred = model(text_batch, lengths)[:, 0]
            loss = loss_fn(pred, label_batch)
            total_acc += ((pred>=0.5).float() == label_batch).float().sum().
 ↪item()
            total_loss += loss.item()*label_batch.size(0)
    return total_acc/len(dataloader.dataset), total_loss/len(dataloader.dataset)
```

Similarly, we will develop the `evaluate` function to measure the model's performance on a given dataset. For a binary classification with a single class-membership probability output, we use the binary cross-entropy loss (`BCELoss`) as the loss function.

We train the model for 10 epochs and display the training and validation performances.

[ ]:
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10

torch.manual_seed(1886)

for epoch in range(num_epochs):
    acc_train, loss_train = train(train_dl)
    acc_valid, loss_valid = evaluate(valid_dl)
    print(f'Epoch {epoch} accuracy: {acc_train:.4f} val_accuracy: {acc_valid:.
 ↪4f}')
```

After training this model for 10 epochs, we evaluate it on the test data.

[ ]:
```
acc_test, _ = evaluate(test_dl)
print(f'test_accuracy: {acc_test:.4f}')
```

Note that the above model is not the best when compared to the state-of-the-art methods used on the IMDb dataset.

The bidirectional RNN We can set the bidirectional configuration of the LSTM to `True`, which will make the recurrent layer pass through the input sequences from both directions, start to end, as well as in the reverse direction.

[ ]:
```
class RNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size,
                                      embed_dim,
                                      padding_idx=0)
        self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
                           batch_first=True, bidirectional=True)
        self.fc1 = nn.Linear(rnn_hidden_size*2, fc_hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(fc_hidden_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, text, lengths):
        out = self.embedding(text)
        out = nn.utils.rnn.pack_padded_sequence(out, lengths.cpu().numpy(),␣
 ↪enforce_sorted=False, batch_first=True)
        _, (hidden, cell) = self.rnn(out)
        out = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.sigmoid(out)
        return out

torch.manual_seed(1)
model = RNN(vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size)
model = model.to(device)
```

The bidirectional RNN layer makes two passes over each input sequence: a forward pass and a reverse or backward pass (note that this is not to be confused with the forward and backward passes in the context of backpropagation). The resulting hidden states of these forward and backward passes are usually concatenated into a single hidden state. Other merge modes include summation, multiplication (multiplying the results of the two passes), and averaging (taking the average of the two).

[ ]:
```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)
```

```
num_epochs = 10

torch.manual_seed(1)

for epoch in range(num_epochs):
    acc_train, loss_train = train(train_dl)
    acc_valid, loss_valid = evaluate(valid_dl)
    print(f'Epoch {epoch} accuracy: {acc_train:.4f} val_accuracy: {acc_valid:.
 ↪4f}')
```

[ ]:
```
test_dataset = IMDB(split='test')
test_dl = DataLoader(test_dataset, batch_size=batch_size,
                     shuffle=False, collate_fn=collate_batch)
```

[ ]:
```
acc_test, _ = evaluate(test_dl)
print(f'test_accuracy: {acc_test:.4f}')
```

We can also try other types of recurrent layers, such as the regular RNN. However, as it turns out, a model built with regular recurrent layers won't be able to reach a good predictive performance (even on the training data). For example, if you try replacing the bidirectional LSTM layer in the previous code with a unidirectional `nn.RNN` (instead of `nn.LSTM`) layer and train the model on full-length sequences, you may observe that the loss will not even decrease during training. The reason is that the sequences in this dataset are too long, so a model with an RNN layer cannot learn the long-term dependencies and may suffer from vanishing or exploding gradient problems.