

Deep Learning

Week 6: Recurrent neural networks

Contents

[1. Introduction](#)

[2. Recurrent neural networks \(*\)](#)

[3. Long Short Term Memory \(LSTM\) \(*\)](#)

[4. Preprocessing and Embedding layers \(*\)](#)

[References](#)

Introduction

In the last week of the module we studied a very important neural network architecture that is the convolutional neural network. You learned the operations that are carried out by convolutional layers and pooling layers, as well as the hyperparameter choices within those layers and the effect they can have on the layer outputs. We also covered transposed convolutions, which can be thought of as a reverse analog to the regular convolutional layers.

One of the main motivations for developing convolutional neural networks was to design a model that captures important structural properties that we know are contained in the data. CNNs have an equivariance property that means they're adapted well for image data, because we know that in images, we want to be able to detect the same features in different regions of the input.

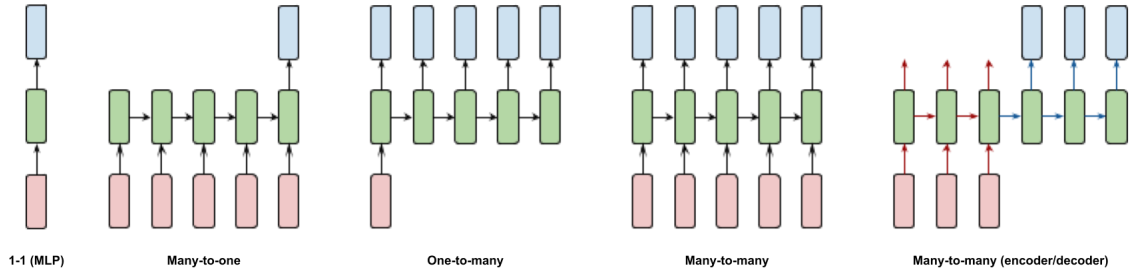
In this week of the course, we will look at another very important and widespread model architecture, which is the recurrent neural network (RNN). This is another network type where we deliberately build structure into the network itself in order to capture certain aspects of the data. In the case of recurrent neural networks, these are intended for sequence data.

We will examine the different possible types of sequence modelling tasks, and see how RNNs are very flexible models that can be used in many different configurations. You'll learn the basic RNN computation, and more sophisticated architectures such as stacked RNNs, bidirectional layers, and the long short term memory architecture (LSTM).

You will also see how to implement all of these models and layer types using the Keras RNN API, as well as learning about some of the preprocessing and embedding layers available in Keras.

Recurrent neural networks

A particular challenge with sequential data and modelling tasks is that the sequence lengths can vary from one dataset example to the next. This makes the use of a fixed input size architecture such as the MLP unsuitable. In addition, there can be many different types of sequential modelling tasks that we might want to consider, each of which could have different architectural requirements, as illustrated in the following diagram.



Different architectures for recurrent neural networks

Typical sequence modelling tasks could include:

- Text sentiment analysis (many-to-one)
- Image captioning (one-to-many)
- Language translation (many-to-many)
- Part-of-speech tagging (many-to-many)

Recurrent neural networks (Rumelhart et al 1986b) are designed to handle this variability of data lengths and diversity of problem tasks.

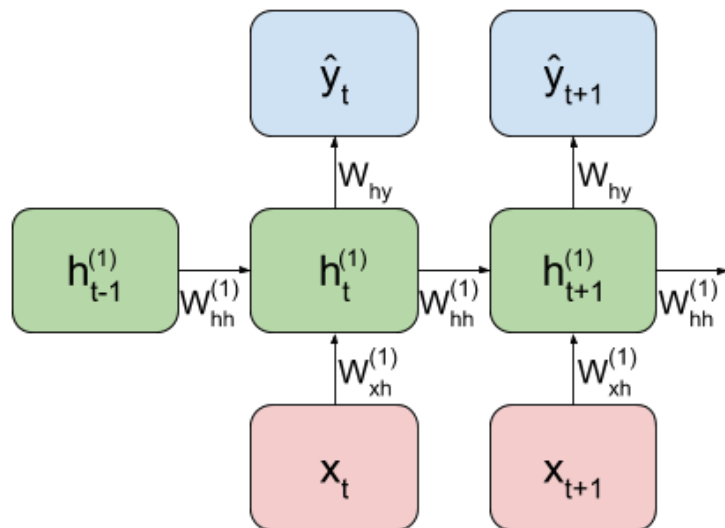
Basic RNN computation

Let $\{\mathbf{x}_t\}_{t=1}^T$ be an example sequence input, with each $\mathbf{x}_t \in \mathbb{R}^D$. Suppose that we are in the many-to-many setting, and there is a corresponding sequence of labels $\{y_t\}_{t=1}^T$, with $y_t \in Y$, where Y could be $\{0, 1\}$ for a binary classification task for example.

The basic RNN computation is given as follows:

$$\left. \begin{aligned} \mathbf{h}_t^{(1)} &= \sigma \left(\mathbf{W}_{hh}^{(1)} \mathbf{h}_{t-1}^{(1)} + \mathbf{W}_{xh}^{(1)} \mathbf{x}_t + \mathbf{b}_h^{(1)} \right), \\ \hat{y}_t &= \sigma_{out} \left(\mathbf{W}_{hy} \mathbf{h}_t^{(1)} + \mathbf{b}_y \right), \end{aligned} \right\} \quad (1)$$

for $t = 1, \dots, T$, where $\mathbf{h}^{(1)} \in \mathbb{R}^{n_1}$, $\mathbf{W}_{hh}^{(1)} \in \mathbb{R}^{n_1 \times n_1}$, $\mathbf{W}_{xh}^{(1)} \in \mathbb{R}^{n_1 \times D}$, $\mathbf{b}_h^{(1)} \in \mathbb{R}^{n_1}$, $\mathbf{W}_{hy} \in \mathbb{R}^{n_y \times n_1}$, $\mathbf{b}_y \in \mathbb{R}^{n_y}$, σ and σ_{out} are activation functions, n_1 is the number of units in the hidden layer, and n_y is the dimension of the output space Y .



Basic computation for recurrent neural networks

Recurrent neural networks make use of weight sharing, similar to convolutional neural networks, but this time the weights are shared across time. This allows the RNN to be 'unrolled' for as many time

steps as there are in the data input \mathbf{x} .

The RNN also has a **persistent state**, in the form of the hidden layer $\mathbf{h}^{(1)}$. This hidden state can carry information over an arbitrary number of time steps, and so predictions at a given time step t can depend on events that occurred at any point in the past, at least in principle. As with MLPs, the hidden state stores **distributed representations** of information, which allows them to store a lot of information, in contrast to hidden Markov models.

Note that the computation (1) requires an **initial hidden state** $\mathbf{h}_0^{(1)}$ to be defined. In practice, this is often just set to the zero vector, although it can also be learned as additional parameters.

In Keras, the RNN is available as the layer `SimpleRNN` in the `keras.layers` module (see [the docs](#)). It can be included in the list of layers passed to the `Sequential` constructor, or using the functional API.

```
In [2]: import keras
        from keras import ops
```

```
In [4]: # Demonstrate the SimpleRNN layer

        from keras.models import Sequential
        from keras.layers import Input, SimpleRNN

        rnn_model = Sequential([
            Input(shape=(10, 2)),
            SimpleRNN(32, activation='tanh') # 'tanh' is the default activation
        ])
```

The Tensor shape expected by a recurrent neural network layer is of the form `(batch_size, sequence_length, num_features)`. In the above, the `input_shape` specifies that the sequence length is 10 and there are 2 features.

```
In [5]: # Print the model summary

        rnn_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
simple_rnn_1 (SimpleRNN)	(None, 32)	1,120

Total params: 1,120 (4.38 KB)

Trainable params: 1,120 (4.38 KB)

Non-trainable params: 0 (0.00 B)

By default, the RNN only returns the final hidden state output.

```
In [6]: # Call the RNN on a dummy input

        inputs = keras.random.normal((1, 10, 2))
        rnn_model(inputs)
```

```
Out[6]: <tf.Tensor: shape=(1, 32), dtype=float32, numpy=
array([[ 0.12140578, -0.2350057 , -0.8039604 ,  0.02706611,  0.20736054,
         0.7405195 ,  0.3358254 ,  0.20457354, -0.54497945,  0.04887557,
         0.48803544, -0.10218947, -0.33647674, -0.41242748,  0.8253199 ,
        -0.36177987, -0.8023034 , -0.40736666, -0.36871386,  0.10368116,
        -0.1555262 ,  0.08075365,  0.42901447, -0.04500461, -0.532167 ,
         0.11766547,  0.05987708,  0.04540725,  0.15244623,  0.44724563,
         0.33395344, -0.5335401 ]], dtype=float32)>
```

The default initial hidden state is zeros, but it can be explicitly set in the layer's `call` method:

```
In [7]: # Set the initial hidden state of a SimpleRNN layer

rnn_layer = SimpleRNN(3)
dummy_inputs = keras.random.normal((16, 5, 2))
layer_output = rnn_layer(dummy_inputs, initial_state=ops.ones((16, 3)))
layer_output.shape
```

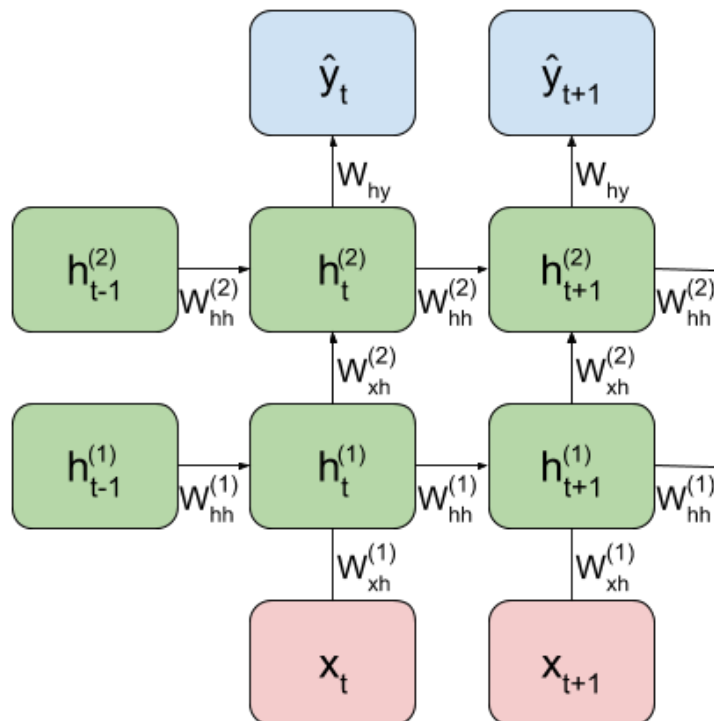
```
Out[7]: TensorShape([16, 3])
```

Stacked RNNs

RNNs can also be made more powerful by stacking recurrent layers on top of each other:

$$\left. \begin{aligned} \mathbf{h}_t^{(k)} &= \sigma \left(\mathbf{W}_{hh}^{(k)} \mathbf{h}_{t-1}^{(k)} + \mathbf{W}_{xh}^{(k)} \mathbf{h}_t^{(k-1)} + \mathbf{b}_h^{(k)} \right), \quad k = 1, \dots, L, \\ \hat{\mathbf{y}}_t &= \sigma_{out} \left(\mathbf{W}_{hy} \mathbf{h}_t^{(L)} + \mathbf{b}_y \right), \end{aligned} \right\} \quad (2)$$

where $\mathbf{h}^{(k)} \in \mathbb{R}^{n_k}$, $\mathbf{W}_{hh}^{(k)} \in \mathbb{R}^{n_k \times n_k}$, $\mathbf{W}_{xh}^{(k)} \in \mathbb{R}^{n_k \times n_{k-1}}$, $\mathbf{b}_h^{(k)} \in \mathbb{R}^{n_k}$, $\mathbf{W}_{hy} \in \mathbb{R}^{n_y \times n_L}$, $\mathbf{b}_y \in \mathbb{R}^{n_y}$, and we have set $n_{L+1} = n_y$, $n_0 = D$, and $\mathbf{h}^{(0)} = \mathbf{x}_t$.



Stacked recurrent neural network

To create a stacked RNN in Keras, we need to obtain the full sequence of hidden states in the lower layer. This can be done using the `return_sequences` keyword argument in the layer constructor.

```
In [8]: # Create a SimpleRNN layer that returns sequences
```

```
rnn_layer_1 = SimpleRNN(16, return_sequences=True)
```

```
In [9]: # Create the second SimpleRNN layer, this only returns the final state
```

```
rnn_layer_2 = SimpleRNN(8)
```

```
In [10]: # Build the stacked RNN model using the functional API
```

```
from keras.models import Model
from keras.layers import Input

inputs = Input(shape=(32, 5))
h = rnn_layer_1(inputs)
outputs = rnn_layer_2(h)
stacked_rnn_model = Model(inputs=inputs, outputs=outputs)
```

```
In [11]: # Print the model summary
```

```
stacked_rnn_model.summary()
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 32, 5)	0
simple_rnn_3 (SimpleRNN)	(None, 32, 16)	352
simple_rnn_4 (SimpleRNN)	(None, 8)	200

Total params: 552 (2.16 KB)

Trainable params: 552 (2.16 KB)

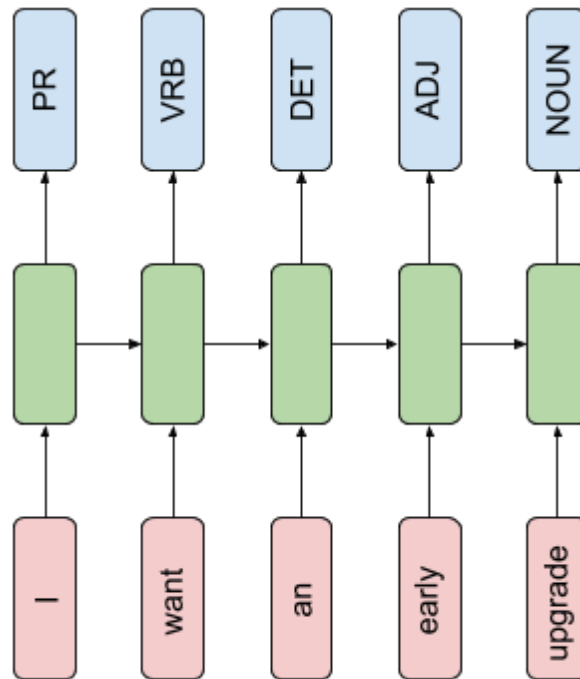
Non-trainable params: 0 (0.00 B)

Note the output shapes in the above summary. The first RNN layer returns a sequence (length 32) of hidden states (of size 16), and the second RNN layer only returns the final hidden state.

Bidirectional RNNs

Standard recurrent neural networks are uni-directional; that is, they only take past context into account. In some applications (where the full input sequence is available to make predictions) it is possible and desirable for the network to take both past and future context into account.

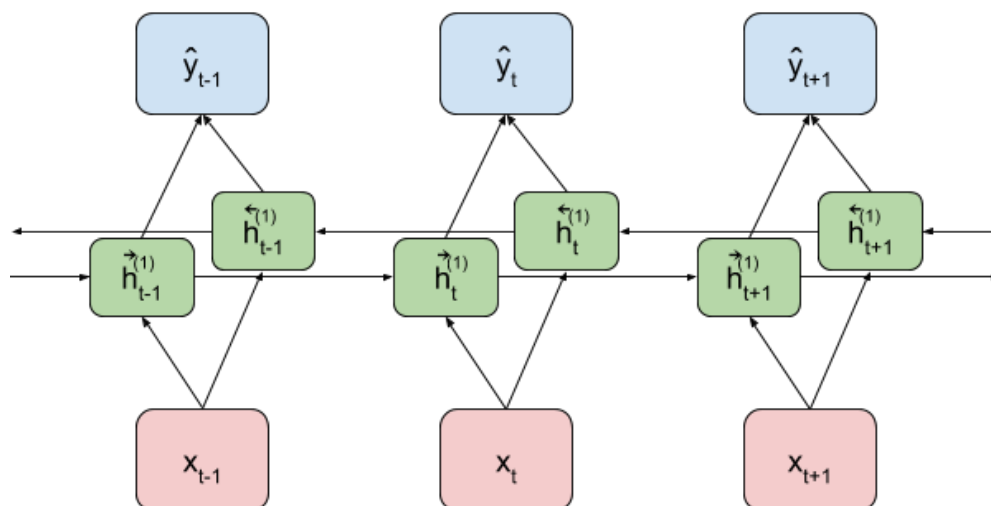
For example, consider a part-of-speech (POS) tagging problem, where the task is to label each word in a sentence according to its particular part of speech, e.g. noun, adjective, verb etc.



Part-of-speech (POS) tagging example

In some cases the correct label can be ambiguous given only the past context, for example the word **light** in the sentence "There's a light ..." could be a noun or a verb depending on how the sentence continues (e.g. "There's a light on upstairs" or "There's a light breeze").

Bidirectional RNNs (Schuster & Paliwal 1997) are designed to look at both future and past context. They consist of two RNNs running forward and backwards in time, whose states are combined in sum way (e.g. adding or concatenating) to produce the final hidden state of the layer.



Bidirectional recurrent neural network

Bidirectional recurrent neural networks (BRNNs) are implemented in Keras using the `Bidirectional` wrapper (see [the docs](#)):

```
In [12]: # Build a bidirectional recurrent neural network
from keras.layers import Bidirectional
```

```
brnn_model = Sequential([
    Input(shape=(64, 7)),
    Bidirectional(SimpleRNN(16, return_sequences=True), merge_mode='concat')
])
```

The `Bidirectional` wrapper constructs two RNNs running in different time directions. The `merge_mode='concat'` setting is the default for the `Bidirectional` constructor, and means that the bidirectional layer concatenates the hidden states from the forward and backward RNNs. This means that the number of units per time step in the output of the layer is $2 \times 16 = 32$:

```
In [13]: # Print the model summary

brnn_model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 64, 32)	768

Total params: 768 (3.00 KB)

Trainable params: 768 (3.00 KB)

Non-trainable params: 0 (0.00 B)

The `Bidirectional` wrapper can also operate on RNN layers with `return_sequences=False`, in which case it combines the final hidden states of the forward and backward RNNs.

Training RNNs

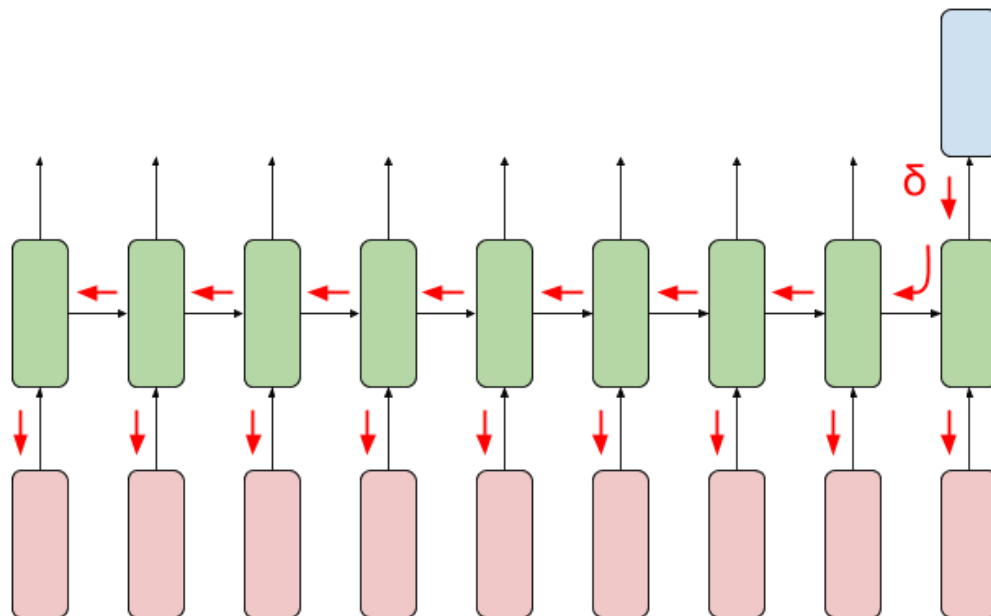
RNNs are trained in the same way as multilayer perceptrons and convolutional neural networks. A loss function $L(\mathbf{y}_1, \dots, \mathbf{y}_T, \hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_T)$ is defined according to the problem task and learning principle, and the network is trained using the backpropagation algorithm and a selected network optimiser. In the many-to-one case (e.g. sentiment analysis), the loss function may be defined as $L(\mathbf{y}_T, \hat{\mathbf{y}}_T)$.

Recall the equation describing the backpropagation of errors in the MLP case:

$$\delta^{(k)} = \sigma'(\mathbf{a}^{(k)}) (\mathbf{W}^{(k)})^T \delta^{(k+1)}, \quad k = 1, \dots, L \quad (3)$$

where k indexes the hidden layers. In the case of recurrent neural networks, the errors primarily backpropagate along the time direction, and we obtain the following propagation of errors in the hidden states:

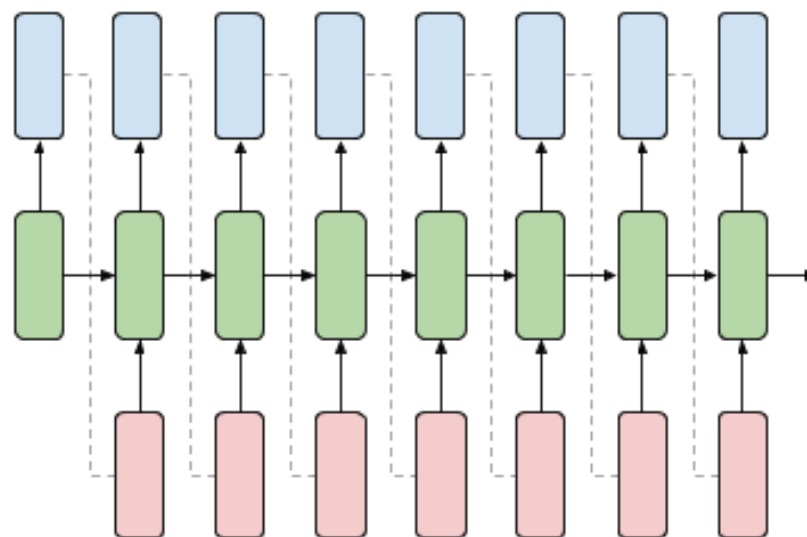
$$\delta_{t-1}^{(k)} = \sigma'(\mathbf{a}_{t-1}^{(k)}) (\mathbf{W}_{hh}^{(k)})^T \delta_t^{(k)}, \quad t = T, \dots, 1 \quad (4)$$



When training RNNs, the errors backpropagate along the time axis

For this reason, the backpropagation algorithm for RNNs is referred to as **backpropagation through time** (BPTT).

Recurrent neural networks can also be trained as generative models for unlabelled sequence data, by re-wiring the network to send the output back as the input to the next step:



Generative RNN model, with the outputs fed back at inputs at the next time step

This is an example of **self-supervised learning**, which is where we use an unlabelled dataset to frame a supervised learning problem. This can be used to train language models, or generative music models for example. In practical we treat this case the same as a supervised learning problem, where the outputs are the same as the inputs but shifted by one time step. This particular technique is also sometimes referred to as **teacher forcing**.

Long Short Term Memory (LSTM)

As mentioned previously, recurrent neural networks can in principle use information from events that occurred many time steps earlier to make predictions at the current time step. However, in

practice RNNs struggle to make use of long-term dependencies in the data.

Recall the equation describing the backpropagation of errors in an MLP:

$$\delta^{(k)} = \sigma'(\mathbf{a}^{(k)}) (\mathbf{W}^{(k)})^T \delta^{(k+1)}, \quad k = 1, \dots, L$$

where k indexes the hidden layers, and the corresponding equation for the backpropagation through time (BPTT) algorithm:

$$\delta_{t-1}^{(k)} = \sigma'(\mathbf{a}_{t-1}^{(k)}) (\mathbf{W}_{hh}^{(k)})^T \delta_t^{(k)}, \quad t = 1, \dots, T$$

where k now indexes the stacked recurrent layers and t indexes the time steps. The above equations indicate a fundamental problem of training neural networks: the **vanishing gradients problem**.

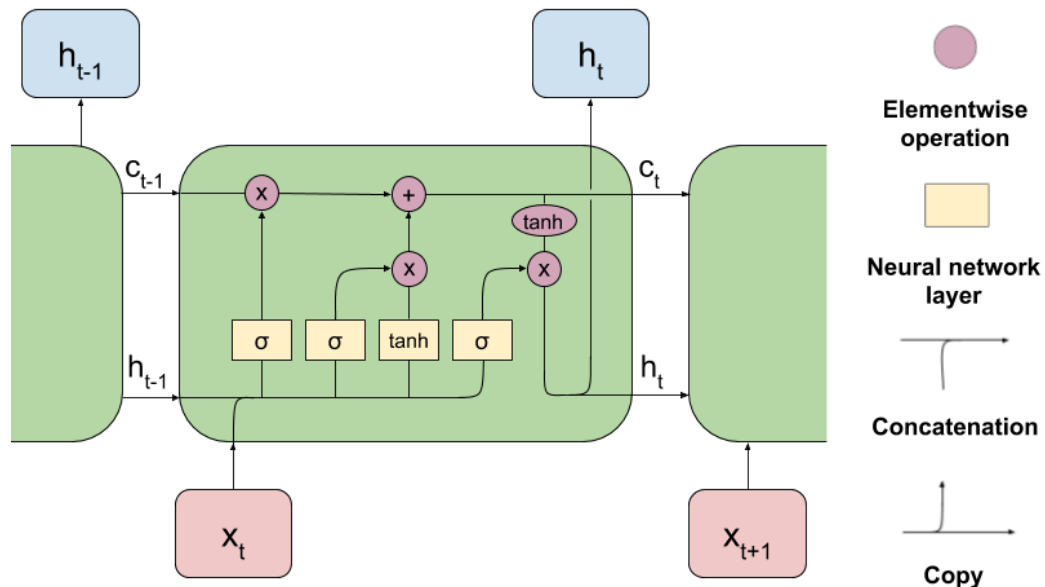
Gradients can explode or vanish with a large number of layers, or a large number of time steps. This problem was pointed out by [Hochreiter](#), and is particularly bad in the case of RNNs, where the length of sequences can be long (e.g. 100 time steps).

The Long Short Term Memory (LSTM) network was introduced by [Hochreiter and Schmidhuber](#) (and later updated by [Gers](#)) to mitigate the effect of vanishing gradients and allow the recurrent neural network to remember things for a long time.

The LSTM has inputs $\mathbf{x}_t \in \mathbb{R}^{n_{k-1}}$ and $\mathbf{h}_{t-1} \in \mathbb{R}^{n_k}$ just as regular RNNs. However, it also includes an internal **cell state** $\mathbf{c}_t \in \mathbb{R}^{n_k}$ that allows the unit to store and retain information (we drop the superscript (k) in this section to ease notation).

The LSTM cell works with a gating mechanism, consisting of logistic and linear units with multiplicative interactions. Information is allowed into the cell state when the 'write' gate is on, it can choose to erase information in the cell state when the 'forget' gate is on, and can read information from the cell state when the 'read' gate is on.

The following schematic diagram outlines the gating system of the LSTM unit.



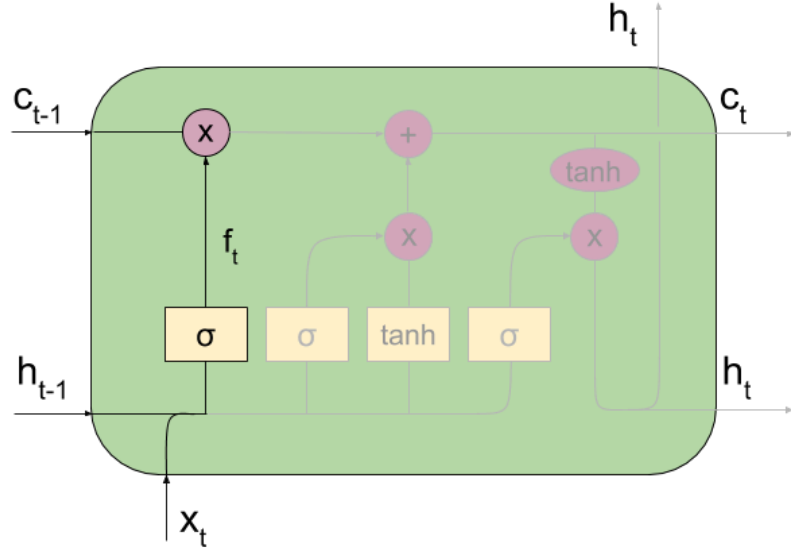
The Long Short Term Memory (LSTM) gating system

First of all, note that there is no neural network layer that operates directly on the cell state. This means that information is more freely able to travel across time steps in the cell state. The role of the hidden state is to manage the information flow in and out of the cell state, according to the signals provided in the inputs \mathbf{h}_{t-1} and \mathbf{x}_t .

The first of these operations is the *forget gate*.

The forget gate

The forget gate determines what information should be erased from the cell state.



The Long Short Term Memory (LSTM) forget gate

The information is controlled by signals in the inputs \mathbf{h}_{t-1} and \mathbf{x}_t according to the following equation:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_f),$$

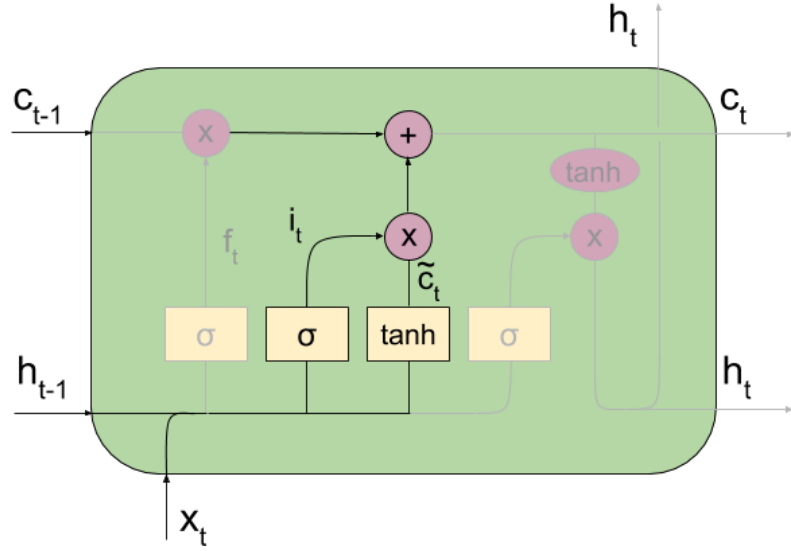
where $[\mathbf{x}_t, \mathbf{h}_{t-1}] \in \mathbb{R}^{n_k + n_{k-1}}$ is the concatenation of \mathbf{x}_t and \mathbf{h}_{t-1} , $\mathbf{W}_f \in \mathbb{R}^{n_k \times (n_k + n_{k-1})}$, $\mathbf{b}_f \in \mathbb{R}^{n_k}$ and σ is the sigmoid activation function. Note that entries of \mathbf{f}_t will be close to one for large positive pre-activation values, and close to zero for large negative pre-activation values. The cell state is then updated

$$\mathbf{c}_t \leftarrow \mathbf{f}_t \odot \mathbf{c}_{t-1}$$

where \odot is the Hadamard (element-wise) product, so that selected entries of the cell state \mathbf{c}_{t-1} are erased, while others are retained.

The input and content gates

The input gate determines when information should be written into the cell state. The content gate contains the information to be written.



The Long Short Term Memory (LSTM) input and content gates

The input and content gates are a combination of sigmoid and tanh activation gates:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_i) \quad (1)$$

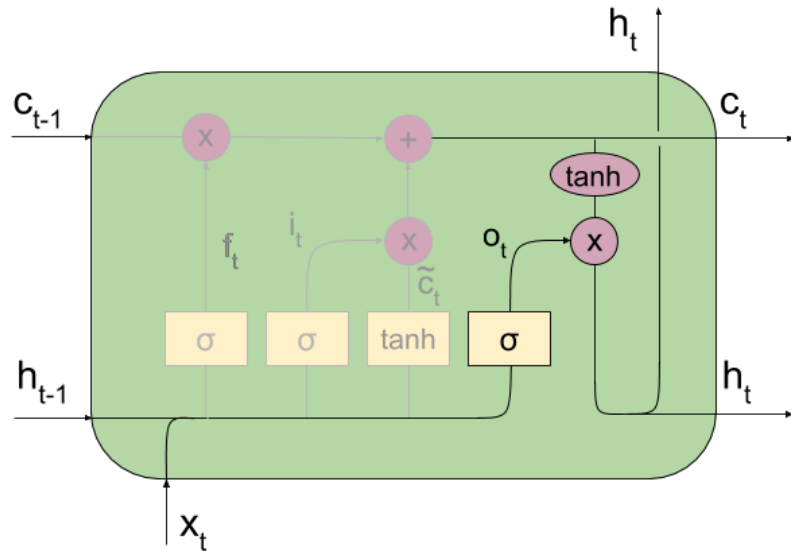
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_c), \quad (2)$$

where $\mathbf{W}_i, \mathbf{W}_c \in \mathbb{R}^{n_k \times (n_k + n_{k-1})}$ and $\mathbf{b}_i, \mathbf{b}_c \in \mathbb{R}^{n_k}$. In a similar way to the forget gate, the input gate \mathbf{i}_t is used to 'zero out' selected entries in the content signal $\tilde{\mathbf{c}}_t$. The content entries that are allowed through the gate are then added into the cell state:

$$\mathbf{c}_t \leftarrow \mathbf{c}_t + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

The output gate

Finally, the output gate decides which cell state values should be output in the hidden state.



The Long Short Term Memory (LSTM) output gate

The output gate is another sigmoid gate that releases information from the cell state after passing through a tanh activation:

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o) \quad (3)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (4)$$

The LSTM network has been immensely successful in sequence modelling tasks, including handwriting recognition (Graves et al 2009), speech recognition (Graves et al 2013), machine translation (Wu et al 2016) and reinforcement learning for video games (Vinyals et al 2019).

Another type of gated recurrent cell that should be mentioned is the Gated Recurrent Unit (GRU), proposed in Cho et al 2014, which simplifies the architecture by combining the forget and input gates into a single 'update' gate, and also merges the cell state and hidden state. We will not go into detail of this cell architecture, for more details refer to the paper.

In Keras, the LSTM is implemented as another layer in the `keras.layers` module:

```
In [14]: import keras
```

```
In [15]: # Build an LSTM model

from keras.models import Sequential
from keras.layers import Input, LSTM

lstm = Sequential([
    Input(shape=(None, 12)),
    LSTM(16, return_sequences=True),
    LSTM(16),
])
```

```
In [16]: # Print the model summary

lstm.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, None, 16)	1,856
lstm_1 (LSTM)	(None, 16)	2,112

Total params: 3,968 (15.50 KB)

Trainable params: 3,968 (15.50 KB)

Non-trainable params: 0 (0.00 B)

RNN cells also have the optional keyword argument `return_state`, which defaults to `False`. When `True`, the layer returns the final internal state, in addition to its output. In the case of LSTM, this internal state is the hidden state \mathbf{h}_t and cell state \mathbf{c}_t . So the `LSTM` layer would return `(outputs, hidden_state, cell_state)` when `return_state=True`.

```
In [17]: # Build an LSTM model that returns its final internal state

from keras.models import Model

inputs = Input(shape=(8, 4))
outputs = LSTM(6, return_state=True, return_sequences=True)(inputs)
lstm2 = Model(inputs=inputs, outputs=outputs)
```

```
In [18]: # Print the model summary

lstm2.summary()
```

Model: "functional_5"

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 8, 4)	0
lstm_2 (LSTM)	[(None, 8, 6), (None, 6), (None, 6)]	264

Total params: 264 (1.03 KB)

Trainable params: 264 (1.03 KB)

Non-trainable params: 0 (0.00 B)

```
In [19]: # View the model outputs
```

```
lstm2.outputs
```

```
Out[19]: [<KerasTensor shape=(None, 8, 6), dtype=float32, sparse=False, name=keras_tensor_13>,
<KerasTensor shape=(None, 6), dtype=float32, sparse=False, name=keras_tensor_14>,
<KerasTensor shape=(None, 6), dtype=float32, sparse=False, name=keras_tensor_15>]
```

```
In [21]: # Test the model on a dummy input
```

```
lstm2(keras.random.normal((1, 8, 4)))
```

```
Out[21]: (<tf.Tensor: shape=(1, 8, 6), dtype=float32, numpy=
array([[[ -0.09650248,  0.01431146,  0.07847904,  0.2846786 ,
          -0.01331123,  0.3721904 ],
        [-0.16631337,  0.15607817, -0.15621708,  0.26059946,
          0.14725848,  0.2761241 ],
        [ 0.02903903, -0.05416757,  0.03177598,  0.14389275,
          -0.01958566,  0.0867793 ],
        [ 0.09210492, -0.08639564, -0.15918073,  0.06285801,
          0.06761988, -0.03277335],
        [ 0.06509916,  0.03874862, -0.21841785, -0.03050784,
          0.07375132, -0.0411678 ],
        [-0.0590728 ,  0.23395304,  0.05345579,  0.20870732,
          -0.09139413,  0.15809146],
        [-0.237485 ,  0.24087806,  0.09957374,  0.22472683,
          -0.04042064, -0.01976617],
        [-0.1740079 ,  0.23214899,  0.16788602,  0.05304179,
          -0.11674525, -0.10827446]]], dtype=float32)>,
<tf.Tensor: shape=(1, 6), dtype=float32, numpy=
array([[ -0.1740079 ,  0.23214899,  0.16788602,  0.05304179, -0.11674525,
          -0.10827446]], dtype=float32)>,
<tf.Tensor: shape=(1, 6), dtype=float32, numpy=
array([[ -0.4907873 ,  0.36711833,  0.46778953,  0.11227942, -0.27226335,
          -0.23527455]], dtype=float32)>)
```

The `LSTM` can be also be called using the `initial_state` argument; in this case, a list of `[hidden_state, cell_state]` should be passed to this argument.

The GRU is also available as the `GRU` layer in `keras.layers`, and has a similar API.

Preprocessing and Embedding layers

In this final section of the week we will look at layers that are particularly useful when working with text data. [Preprocessing layers](#) can be used to convert text data into a numerical representation that can be used by neural networks. Embedding layers take data that has been tokenized into integer sequences, and act as a look-up table to map each integer token to its own embedding vector in \mathbb{R}^D .

```
In [2]: import os
os.environ['KERAS_BACKEND'] = 'torch'

import keras
from keras import ops
import torch
```

For this tutorial we will use the [Twitter airline sentiment dataset](#) from Kaggle, which consists of 14,640 tweets labelled as having positive, negative or neutral sentiment.

Loading and preparing the data

```
In [3]: # Load the data

import pandas as pd
from pathlib import Path

df = pd.read_csv(Path('./data/tweets.csv'))
print(df.shape)
df.head()
```

(14640, 15)

```
Out[3]:
```

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativer
--	----------	-------------------	------------------------------	----------------	-----------

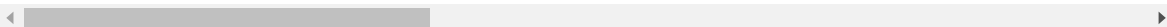
0	570306133677760513	neutral	1.0000	NaN	
---	--------------------	---------	--------	-----	--

1	570301130888122368	positive	0.3486	NaN	
---	--------------------	----------	--------	-----	--

2	570301083672813571	neutral	0.6837	NaN	
---	--------------------	---------	--------	-----	--

3	570301031407624196	negative	1.0000	Bad Flight	
---	--------------------	----------	--------	------------	--

4	570300817074462722	negative	1.0000	Can't Tell	
---	--------------------	----------	--------	------------	--



```
In [4]: # Extract the relevant columns

df = df[['text', 'airline_sentiment', 'airline_sentiment_confidence']]
```

```
In [5]: # View a sample tweet and its label

df.sample(1).values
```

```
Out[5]: array([[ '@AmericanAir we did for the last 8 hours. Your customer service is horre
ndous. I should have flown Delta',
                'negative', 1.0]], dtype=object)
```

```
In [6]: # Split the data into training, validation and test sets

from sklearn.model_selection import train_test_split

train_df, val_df = train_test_split(df, test_size=0.4)
val_df, test_df = train_test_split(val_df, test_size=0.5)
```

When working with text data, it is useful to know that in TensorFlow, Tensors of string type are allowed but in PyTorch they are not. Therefore, in order to remain backend independent, we will avoid creating string Tensors.

We will choose to work with PyTorch DataLoaders, as the dataset preparation is slightly more involved and will make a useful example for how to make custom Datasets in PyTorch.

NB: If using TensorFlow Datasets, we could save the above DataFrames to CSV files and use the `CsvDataset` class to load the CSVs directly into Dataset objects.

Preprocessing layers

We will need to convert the string data into a numeric representation for the models to process it. We will do this using [preprocessing layers](#).

Our custom Datasets will need to tokenise the input text as well as convert the output text labels to a numeric representation. We will use the `TextVectorization` and `StringLookup` Keras layers to help with these respective tasks.

```
In [8]: # Create a TextVectorization layer

from keras.layers import TextVectorization

textvectorization = TextVectorization(max_tokens=1000)
```

```
In [9]: # Configure the layer to the dataset

textvectorization.adapt(train_df['text'])
```

```
In [10]: # Test the TextVectorization layer

input_text = train_df['text'].sample(1)
print(input_text.values)
textvectorization(input_text)
```

```
['@SouthwestAir thanks, but they told him he couldn't a few hours ago, then my plane
left after I tried to do it. Stuck waiting for hours now']
```

```
Out[10]: tensor([[ 17,  36,  32,  62, 146, 516, 253,   1,   5, 505,  57, 282, 109,  12,
                   63, 233,  90,   4, 267,   2,  45,  21, 221, 107,   8,  57,  37]],
                device='cuda:0')
```

`TextVectorization` layers have a `get_vocabulary()` method which can be used to obtain the (ordered) list of words in the vocabulary. Note that the token index 0 is used for zero padding, and the token index 1 is the OOV (`[UNK]`) token.

```
In [11]: # Get the word vocabulary

vocabulary = textvectorization.get_vocabulary()
inx2word = {i: word for i, word in enumerate(vocabulary)}
```

```
In [12]: # Create a StringLookup layer

from keras.layers import StringLookup

output_labels = ['positive', 'negative', 'neutral']
stringlookup = StringLookup(vocabulary=output_labels, num_oov_indices=0)
```

```
In [13]: # Test the StringLookup layer

sample_labels = train_df['airline_sentiment'].sample(12)
```

```
print(sample_labels.values)
stringlookup(sample_labels)
```

```
['negative' 'negative' 'positive' 'positive' 'negative' 'negative'
 'negative' 'negative' 'negative' 'neutral' 'negative' 'negative']
```

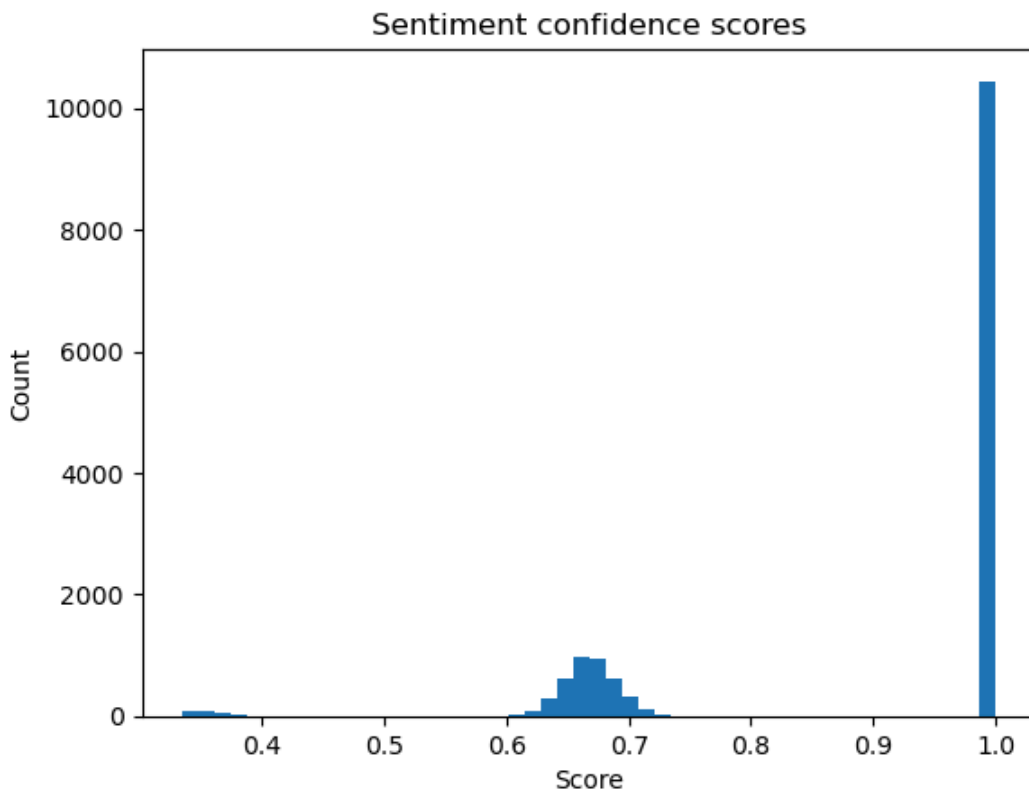
```
Out[13]: tensor([1, 1, 0, 0, 1, 1, 1, 1, 1, 2, 1, 1], device='cuda:0')
```

As an extra preprocessing step, we will also filter out examples where the confidence score is too low.

```
In [14]: # Plot a histogram of confidence scores
```

```
import matplotlib.pyplot as plt

plt.hist(df['airline_sentiment_confidence'], bins=50)
plt.title("Sentiment confidence scores")
plt.xlabel("Score")
plt.ylabel("Count")
plt.show()
```



We will choose 0.5 as a cutoff threshold for the confidence score.

Now we are ready to create our custom Dataset. Custom Datasets in PyTorch should subclass from the `torch.utils.data.Dataset` class, and should implement the `__init__`, `__len__`, and `__getitem__` methods. See [this tutorial](#) for more information and examples.

```
In [15]: # Custom Dataset to read and preprocess the CSV data
```

```
class TweetDataset(torch.utils.data.Dataset):

    def __init__(self, df, textvectorization_layer, stringlookup_layer):
        # NB: base class doesn't define an __init__,
        # so no need to call super().__init__()

        # Filter out low confidence labels
        self.df = df[df['airline_sentiment_confidence'] > 0.5]
```



```

        self.textvectorization = textvectorization_layer
        self.stringlookup = stringlookup_layer

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index):
        elem = self.df.iloc[index]
        tokenized_text = self.textvectorization(elem["text"])
        label = self.stringlookup(elem["airline_sentiment"])
        return tokenized_text, label

```

In [16]: *# Create the custom Dataset objects*

```

train_dataset = TweetDataset(train_df, textvectorization, stringlookup)
val_dataset = TweetDataset(val_df, textvectorization, stringlookup)
test_dataset = TweetDataset(test_df, textvectorization, stringlookup)

```

In [17]: *# Test the training Dataset*

```

import numpy as np

print(len(train_dataset))
sample_inx = np.random.choice(len(train_dataset))
train_dataset[sample_inx]

```

8642

Out[17]: (tensor([14, 12, 187, 15, 159, 12, 9, 40, 83, 10, 4, 31, 29, 1, 27, 129, 2, 144, 100, 28, 380, 76, 292, 277], device='cuda:0'), tensor(1, device='cuda:0'))

The Datasets are now ready to be loaded into DataLoaders as normal. However, these Datasets have the property that the input sentence tokens can have different lengths, making batching difficult. We will solve this problem by adding zero tokens to each example in a batch up to the length of the longest token sequence in the batch (this is what the zero token is for). We could have chosen to do add zero padding in the custom Dataset class above, but we will instead use the following function to pass to the `collate_fn` argument in the DataLoader initializer.

In [18]: *# Define batching function*

```

def padded_batch(batch):
    inputs, outputs = zip(*batch)

    # The pad_sequence fn expects torch Tensors.
    # The following conversion is only necessary for TF backend
    inputs = [torch.tensor(ops.convert_to_numpy(t)) for t in inputs]
    outputs = [torch.tensor(ops.convert_to_numpy(l)) for l in outputs]

    inputs = torch.nn.utils.rnn.pad_sequence(
        inputs, batch_first=True, padding_value=0)
    outputs = torch.tensor(outputs)
    return inputs, outputs

```

NB: If working with TensorFlow Datasets, using `.padded_batch` instead of `.batch` pads the examples as above.

In [19]: *# Create the DataLoaders*

```

train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=16, shuffle=True, collate_fn=padded_batch)
val_dataloader = torch.utils.data.DataLoader(
    val_dataset, batch_size=16, shuffle=False, collate_fn=padded_batch)

```

```
test_dataloader = torch.utils.data.DataLoader(
    test_dataset, batch_size=16, shuffle=False, collate_fn=padded_batch)
```

In [20]: *# Test the training DataLoader*

```
t, l = next(iter(train_dataloader))
print(t)
print(l)
```

```
tensor([[ 7, 43,  1, 60, 35,  6, 118,  2,  2, 290, 912, 693,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [17,  1, 67,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [18, 91, 822, 12, 155,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [ 7,  1, 139, 562,  1, 29,  1, 10, 27,  5,  1, 16,  3,  1,
        78, 476,  1,  1,  1,  1, 20,  2, 529, 20,  8,  1,  1, 262],
        [14, 12,  9,  1, 34, 622, 371,  2, 244, 24, 40, 83, 119, 46,
        11, 66,  1, 10,  4, 74, 23, 27,  1,  2, 296, 584, 47,  0],
        [ 7,  1, 899, 11, 213, 115, 37, 32, 681, 334,  1,  3,  1,  8,
         6,  1, 75,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [ 7, 29, 12,  1, 110, 27,  1,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [18, 15, 25,  5,  1, 191, 161, 84, 25, 133, 232, 60,  4, 23,
         2,  1,  8,  5, 142,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [79, 36, 106,  1, 836, 105,  3,  1,  1,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [ 7, 211, 55, 108, 60,  6, 228,  1, 155, 36, 679,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [ 7,  9,  1, 418, 11, 410, 16, 434, 29, 917, 298, 63, 165, 145,
         1, 10,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [13, 50, 111,  2, 31, 12,  1,  1, 10, 29, 76, 15, 770, 28,
         1, 23, 267,  2, 80, 82,  1, 220,  0,  0,  0,  0,  0,  0,  0],
        [17, 261, 134, 72, 299,  5, 257, 44,  4, 38, 168, 12, 286,  2,
        652, 52, 54,  4, 92, 364,  2, 45,  0,  0,  0,  0,  0,  0],
        [17, 29, 861, 39, 86, 502,  1,  4, 24, 43,  1, 217, 19, 786,
         2,  6, 996, 26,  3, 139,  0,  0,  0,  0,  0,  0,  0,  0],
        [ 7,  9,  1, 775, 30, 327, 41,  1, 884,  1,  4, 335, 540, 586,
        163,  1, 16, 747,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
        [14, 38,  4, 31,  5, 292, 158,  2,  1,  8, 179,  1,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
tensor([1, 0, 1, 1, 1, 1, 1, 2, 0, 1, 1, 1, 1, 0, 1, 2])
```

Embedding layer

We are now able to process the text data into numerical form. However, the input integer tokens should be further processed to transform them into a representation that is more useful for the network. This is where the **Embedding** layer can be used - it creates a lookup table of vectors in \mathbb{R}^D such that each integer token in the vocabulary has its own D -dimensional embedding vector.

In [21]: *# Create an Embedding layer*

```
from keras.layers import Embedding

embedding = Embedding(1000, 2, mask_zero=True)
```

In [22]: *# View the output of the Embedding layer*

```
t, l = next(iter(train_dataloader))
print(embedding(t).shape)
embedding(t)._keras_mask
```

```
torch.Size([16, 25, 2])
```

```
Out[22]: tensor([[ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True, False, False, False, False, False, False,
                  False, False, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True, False, False, False, False, False,
                  False, False, False, False, False, False, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True, False, False, False, False,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True],
 [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
                  False, False, False, False, False]], device='cuda:0')
```

```
In [23]: # Build the classifier model
```

```
from keras.models import Sequential
from keras.layers import Input, Bidirectional, LSTM, Dense

lstm_classifier = Sequential([
    Input(shape=[None]),
    embedding,
    Bidirectional(LSTM(8)),
    Dense(3, activation='softmax')
])
lstm_classifier.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 2)	2,000
bidirectional (Bidirectional)	(None, 16)	704
dense (Dense)	(None, 3)	51

Total params: 2,755 (10.76 KB)


Trainable params: 2,755 (10.76 KB)

Non-trainable params: 0 (0.00 B)


In []: *# Compile and train the model*

```
lstm_classifier.compile(
    optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
history = lstm_classifier.fit(
    train_dataloader, validation_data=val_dataloader, epochs=5)
```


Epoch 1/5

541/541  **132s** 243ms/step - accuracy: 0.6429 - loss: 0.8794 - val_accuracy: 0.6995 - val_loss: 0.6793


Epoch 2/5

541/541  **131s** 242ms/step - accuracy: 0.7317 - loss: 0.6093 - val_accuracy: 0.7273 - val_loss: 0.6095


Epoch 3/5

541/541  **131s** 242ms/step - accuracy: 0.7558 - loss: 0.5679 - val_accuracy: 0.7696 - val_loss: 0.5932

Epoch 4/5

541/541  **131s** 243ms/step - accuracy: 0.7845 - loss: 0.5335 - val_accuracy: 0.7745 - val_loss: 0.5865

Epoch 5/5

541/541  **132s** 243ms/step - accuracy: 0.7968 - loss: 0.5119 - val_accuracy: 0.7856 - val_loss: 0.5603

In [25]: *# Evaluate the model on the test Dataset*

```
lstm_classifier.evaluate(test_dataloader)
```

180/180  **29s** 163ms/step - accuracy: 0.7727 - loss: 0.5856

Out[25]: [0.5860937237739563, 0.7704861164093018]

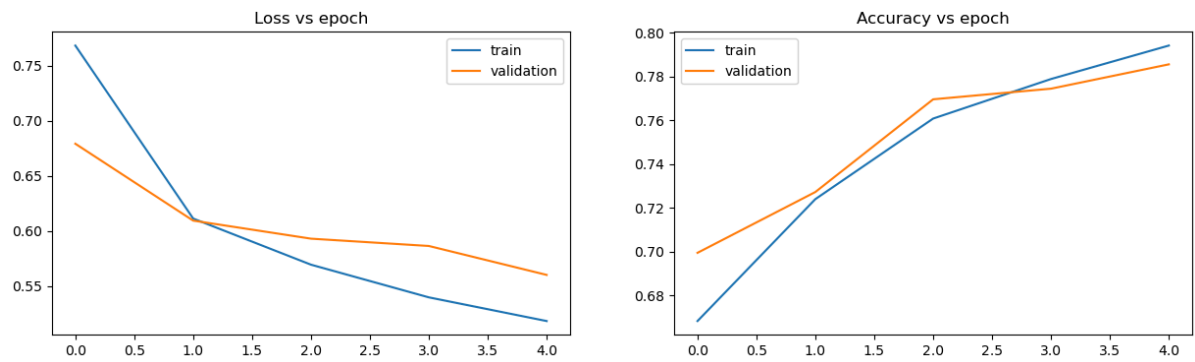
In [26]: *# Plot the learning curves*

```
fig = plt.figure(figsize=(15, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='validation')
plt.legend()
plt.title("Loss vs epoch")

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='validation')
plt.legend()
plt.title("Accuracy vs epoch")

plt.show()
```



In [27]: *# View some model predictions on the test set*

```
text, label = next(iter(test_dataloader))

ground_truth = np.array(output_labels)[label]
predicted_label_ints = np.argmax(
    ops.convert_to_numpy(lstm_classifier(text)), axis=1)
predicted_labels = np.array(output_labels)[predicted_label_ints]

for t, l, g in zip(text, predicted_labels, ground_truth):
    print(' '.join([inx2word[i] for i in ops.convert_to_numpy(t)]))
    print("True label: {}\nPredicted label: {}".format(g, l))
```

usairways amp there are seats together they just have fees seems if im taking a [UNK] flight the least you can do is [UNK] the 10
True label: negative
Predicted label: negative

southwestair for my [UNK] [UNK] [UNK] she would love a [UNK] [UNK] from your flight crew she was a [UNK] for [UNK] airlines
True label: neutral
Predicted label: neutral

jetblue flight [UNK] you had us stuck on the plane for [UNK] 4 hrs set to take off only for flight control to [UNK] us the worst service
True label: negative
Predicted label: negative

wanted to get my bag [UNK] but instead get 25 [UNK] on all three tickets when [UNK] a card [UNK] [UNK] is only option united
True label: negative
Predicted label: negative

americanair can you please help youve now rebooked me [UNK] after flight was cancelled flightled please fix my flt to tuesday
True label: negative
Predicted label: negative

jetblue i would like to receive [UNK]
True label: neutral
Predicted label: neutral

[UNK] youll [UNK] this "jetblue our fleets on fleek [UNK]
True label: neutral
Predicted label: neutral

americanair your staff at jfk airport are [UNK] the way we were left stranded
True label: negative
Predicted label: negative

united follow me back please and ill [UNK] dm you the link because id rather not share my travel plans [UNK]
True label: neutral
Predicted label: negative

usairways [UNK] he still has 1 more flight but so happy to hear theyre in the air just about [UNK] must be the [UNK] [UNK] thx
True label: positive
Predicted label: neutral

usairways why are your customer service [UNK] so [UNK] on [UNK] my [UNK] and [UNK] [UNK] usairways
True label: negative
Predicted label: negative

jetblue hey jetblue you stranded an entire plane that was supposed to go to jfk and we are getting [UNK] need better communication
True label: negative
Predicted label: negative

southwestair used to love you but you keep [UNK] my flights [UNK]
True label: negative
Predicted label: positive

virginamerica thank you see yall soon im excited to see the [UNK] of [UNK] [UNK] those [UNK]
True label: positive
Predicted label: positive

southwestair [UNK] on hold for 2 hours

True label: negative
Predicted label: negative

southwestair its not fun having a delay from nashville to las vegas but the crew at the gate [UNK] desk has been awesome [UNK] [UNK]
True label: positive
Predicted label: negative

Exercise 1. Train a new LSTM classifier model using `tf.data.Dataset` objects, loading the training/validation/test splits with `CsvDataset`, and carrying out all preprocessing using the `map` and `filter` methods.

Exercise 2. Test the trained model as above, but pull the examples directly from the test DataFrame.

References

- Cho, K., van Merriënboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014), "Learning phrase representations using rnn encoder–decoder for statistical machine translation", in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734.
- Gers, F.A. (1999), "Learning to forget: Continual prediction with LSTM", *9th International Conference on Artificial Neural Networks: ICANN '99*, 850–855.
- Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., & Schmidhuber, J. (2009), "A Novel Connectionist System for Unconstrained Handwriting Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **31** (5), 855–868.
- Graves, A., Mohamed, A.-R., Hinton, G. (2013), "Speech Recognition with Deep Recurrent Neural Networks", arXiv preprint, abs/1303.5778.
- Hochreiter, S. (1991), "Untersuchungen zu dynamischen neuronalen Netzen", Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.
- Hochreiter, S. and Schmidhuber, J. (1997), "Long short-term memory", *Neural Computation*, **9** (8), 1735–1780.
- Rumelhart, D. E., Hinton, G., and Williams, R. (1986b), "Learning representations by back-propagating errors", *Nature*, **323**, 533-536.
- Schuster, M. & Paliwal, K. K. (1997), "Bidirectional Recurrent Neural Networks", *IEEE Transactions on Signal Processing*, **45** (11), 2673-2681.
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., & Silver, D. (2019) "Grandmaster level in StarCraft II using multi-agent reinforcement learning", *Nature*, **575** (7782), 350-354.
- Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., & Gao, Q. (2016), "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", arXiv preprint, abs/1609.08144.