Deep Learning

Week 6: Recurrent neural networks

Contents

Introduction

In the last week of the module we studied a very important neural network 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.

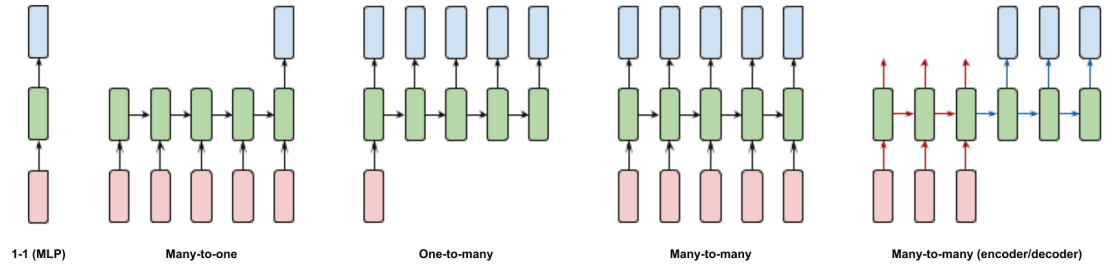
We will examine the different possible types of sequence modelling tasks, and see how RNNs are very flexible models that can be used in many different configurations. You'll learn the basic RNN computation, and more sophisticated architectures such as stacked RNNs, bidirectional layers, and the long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

You will also see how to implement all of these models and layer types using the TensorFlow RNN API, as well as learning about some of the preprocessing and

embedding layers available in TensorFlow.

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

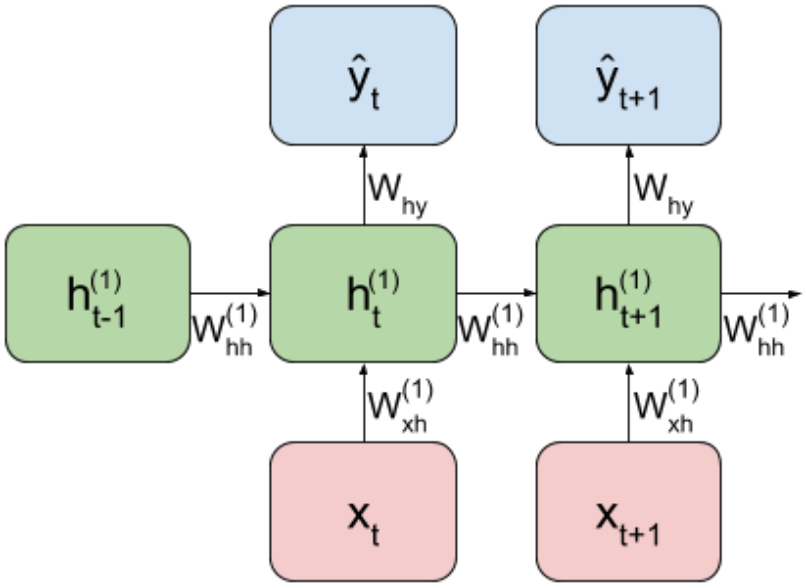
Let $\{\mathbf{x}_t\}_{t=1}^T$ be an example sequence input, with each $\mathbf{x}_t \in \mathbb{R}^D$. Suppose that we are in the many-to-many setting, and there is a corresponding sequence of labels $\{y_t\}_{t=1}^T$, with $y_t \in Y$, where Y could be $\{0, 1\}$ for a binary classification task for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

The basic RNN computation is given as follows:

$$\mathbf{h}_t^{(1)} = \sigma\left(\mathbf{W}_{hh}^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{W}_{xh}^{(1)}\mathbf{x}_t + \mathbf{b}_h^{(1)}\right), \tag{7}$$

$$\hat{\mathbf{y}}_t = \sigma_{out}\left(\mathbf{W}_{hy}\mathbf{h}^{(1)} + \mathbf{b}_y\right), \tag{8}$$

for $t = 1, \ldots, T$, 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 (7)-(8) 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 TensorFlow, the RNN is available as the layer `SimpleRNN` in the `tf.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 [1]:
```
import tensorflow as tf
```

```
2024-01-17 20:45:38.676590: I tensorflow/core/platform/cpu_feature_guard.c
c:182] This TensorFlow binary is optimized to use available CPU instructio
ns in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebui
ld TensorFlow with the appropriate compiler flags.
```

In [2]:
```
# Demonstrate the SimpleRNN layer

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN

rnn_model = Sequential([
    SimpleRNN(32, activation='tanh', input_shape=(10, 2))  # 'tanh' is th
])
```

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 [3]:
```
# Print the model summary

rnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 32)                1120

=================================================================
Total params: 1120 (4.38 KB)
Trainable params: 1120 (4.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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

In [4]:
```
# Call the RNN on a dummy input

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

Out[4]:
```
<tf.Tensor: shape=(1, 32), dtype=float32, numpy=
array([[-0.5844114 ,  0.24856322, -0.1135987 , -0.2496192 , -0.00964957,
        -0.5602601 , -0.69852173,  0.3390541 ,  0.56584084,  0.35058576,
         0.28950974,  0.09292008, -0.01746542, -0.6440658 , -0.5817331 ,
        -0.50168025,  0.3755319 , -0.4939724 ,  0.65413314, -0.8752905 ,
        -0.9538335 ,  0.71521693,  0.73917866, -0.85640454, -0.29861575,
         0.1204354 ,  0.3275132 ,  0.1288251 , -0.1385051 ,  0.08878877,
         0.3688124 , -0.01693271]], dtype=float32)>
```

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

In [5]:
```
# Set the initial hidden state of a SimpleRNN layer

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

Out[5]:
```
TensorShape([16, 3])
```

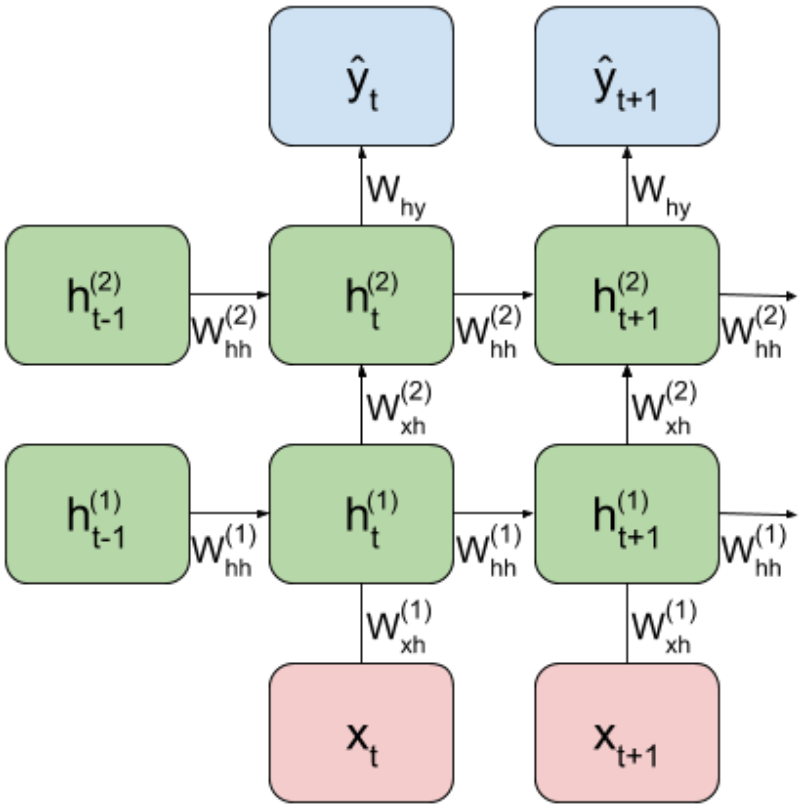
Stacked RNNs

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

$$\mathbf{h}_t^{(k)} = \sigma\left(\mathbf{W}_{hh}^{(k)}\mathbf{h}_{t-1}^{(k)} + \mathbf{W}_{xh}^{(k)}\mathbf{h}_t^{(k-1)} + \mathbf{b}_h^{(k)}\right), \quad k = 1, \ldots, L, \tag{9}$$

$$\hat{\mathbf{y}}_t = \sigma_{out}\left(\mathbf{W}_{hy}\mathbf{h}^{(L)} + \mathbf{b}_y\right), \tag{10}$$

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 TensorFlow, 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 [6]:
```
# Create a SimpleRNN layer that returns sequences

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

In [7]:
```
# Create the second SimpleRNN layer, this only returns the final state

rnn_layer_2 = SimpleRNN(8)
```

In [8]:
```
# Build the stacked RNN model using the functional API

from tensorflow.keras.models import Model
from tensorflow.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 [9]:
```
# Print the model summary

stacked_rnn_model.summary()
```

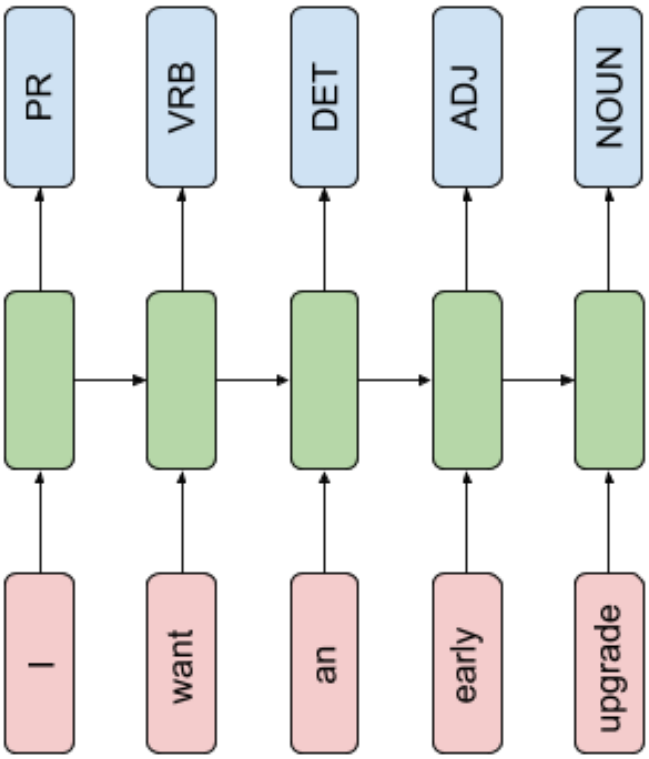
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 5)]           0

 simple_rnn_2 (SimpleRNN)    (None, 32, 16)            352

 simple_rnn_3 (SimpleRNN)    (None, 8)                 200

=================================================================
Total params: 552 (2.16 KB)
Trainable params: 552 (2.16 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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.

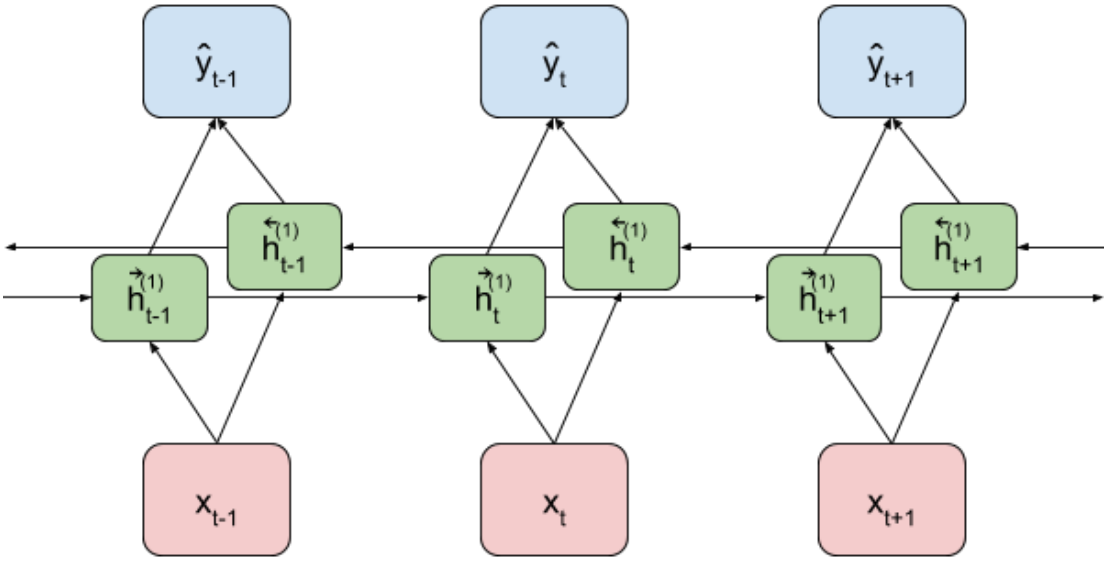
For example, consider a part-of-speech (POS) tagging problem, where the task is to label each word in a sentence according to its particular part of speech, e.g. noun, adjective, verb etc.



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 TensorFlow using the `Bidirectional` wrapper (see the docs):

In [10]:
```
# Build a bidirectional recurrent neural network

from tensorflow.keras.layers import Bidirectional

brnn_model = Sequential([
    Bidirectional(SimpleRNN(16, return_sequences=True), merge_mode='conca
])
```

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

brnn_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional (Bidirection  (None, 64, 32)            768
 al)

=================================================================
Total params: 768 (3.00 KB)
Trainable params: 768 (3.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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

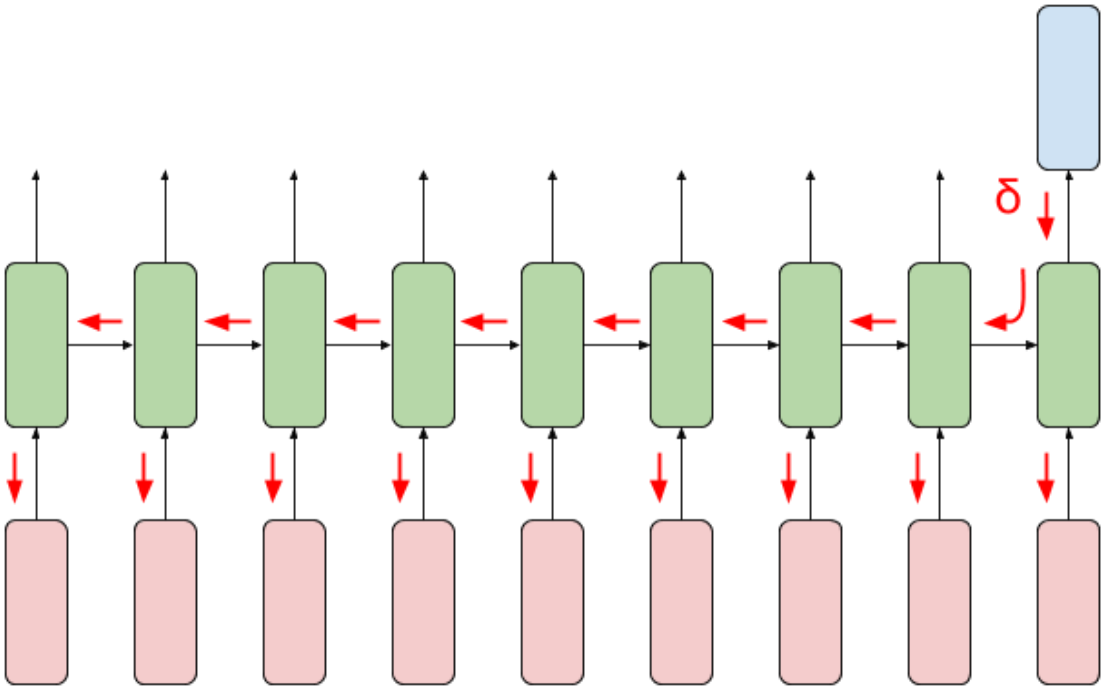
RNNs are trained in the same way as multilayer perceptrons and convolutional neural networks. A loss function $L(\mathbf{y}_1, \ldots, \mathbf{y}_T, \hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_T)$ is defined according to the problem task and learning principle, and the network is trained using the backpropagation 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:

$$\delta^{(k)} = \sigma'(\mathbf{a}^{(k)})(\mathbf{W}^{(k)})^T \delta^{(k+1)}, \qquad k = 1, \ldots, L \tag{11}$$

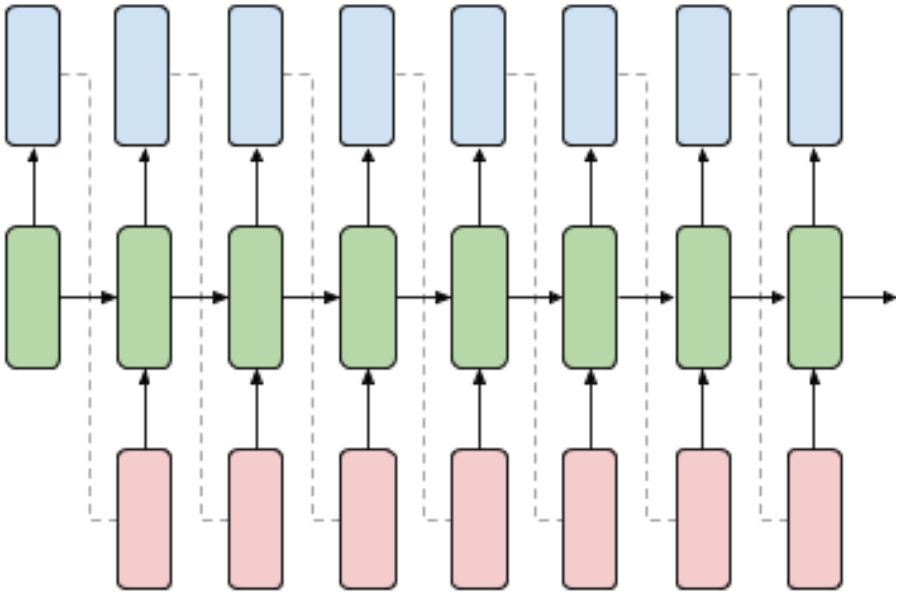
where k indexes the hidden layers. In the case of recurrent neural networks, the errors primarily backpropagate along the time direction, and we obtain the following propagation of errors in the hidden states:

$$\delta_{t-1}^{(k)} = \sigma'(\mathbf{a}_{t-1}^{(k)})(\mathbf{W}_{hh}^{(k)})^T \delta_t^{(k)}, \quad t = T, \ldots, 1 \tag{12}$$



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:

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

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

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

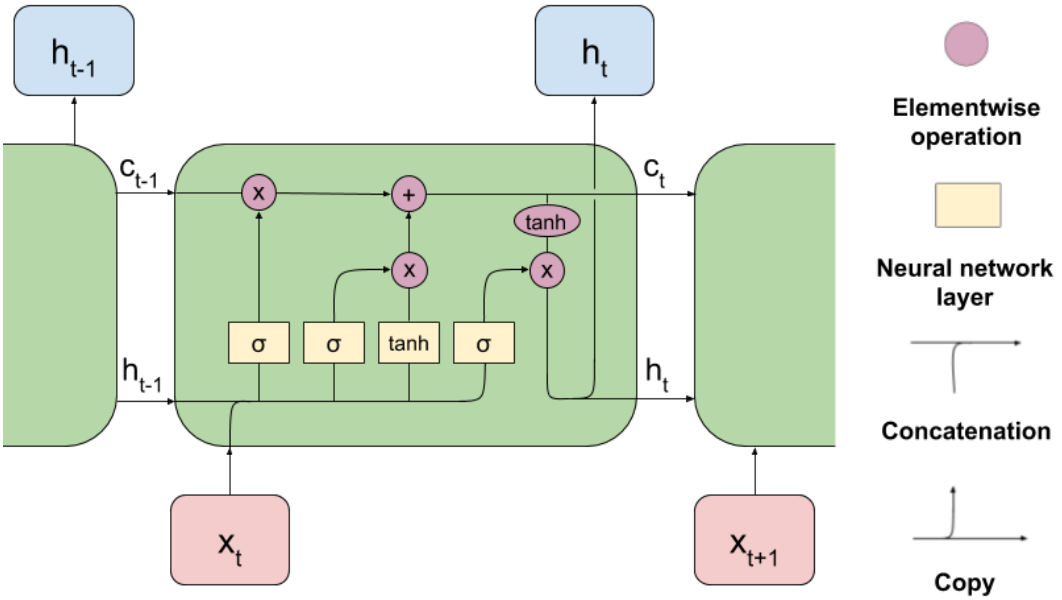
where k now indexes the stacked recurrent layers and t indexes the time steps. The above equations indicates a fundamental problem of training neural networks: the **vanishing gradients problem**. Gradients can explode or vanish with a large number of layers, or a large number of time steps. This problem 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.



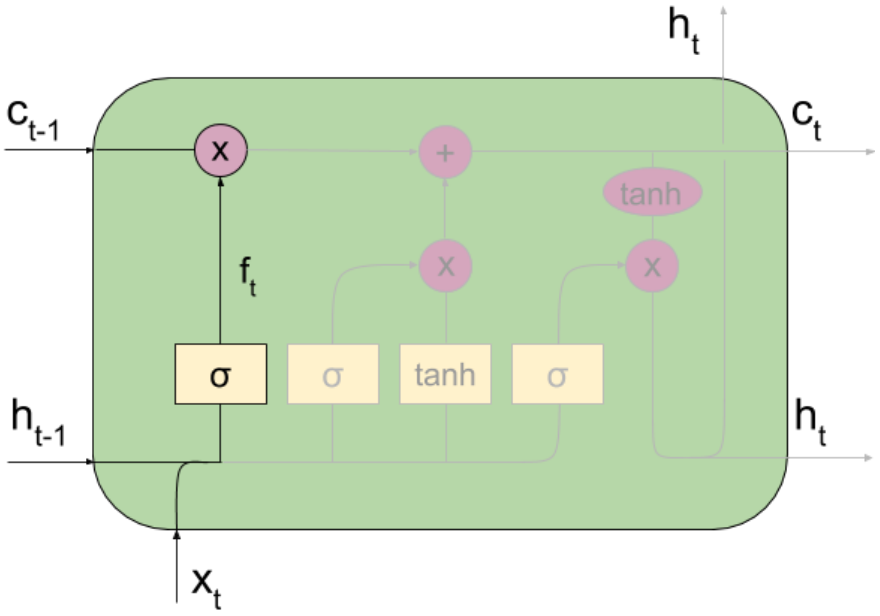
The Long Short Term Memory (LSTM) gating system

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:

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

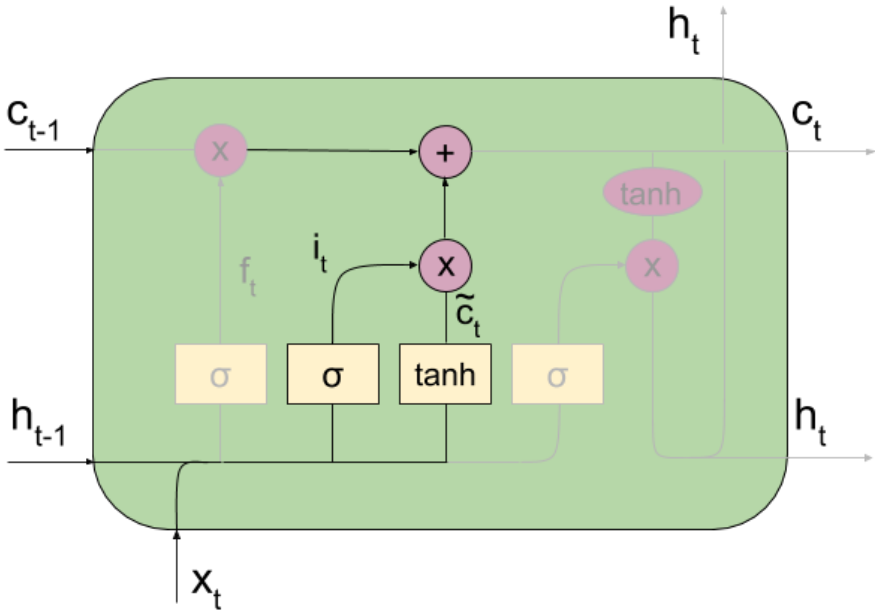
where $[\mathbf{x}_t, \mathbf{h}_{t-1}] \in \mathbb{R}^{n_k \times (n_k + n_{k-1})}$ is the concatenation of $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, $\mathbf{W}_f \in \mathbb{R}^{n_k \times (n_k + n_{k-1})}$, $\mathbf{b}_f \in \mathbb{R}^{n_k}$ and σ is the sigmoid activation function. Note that entries 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:

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

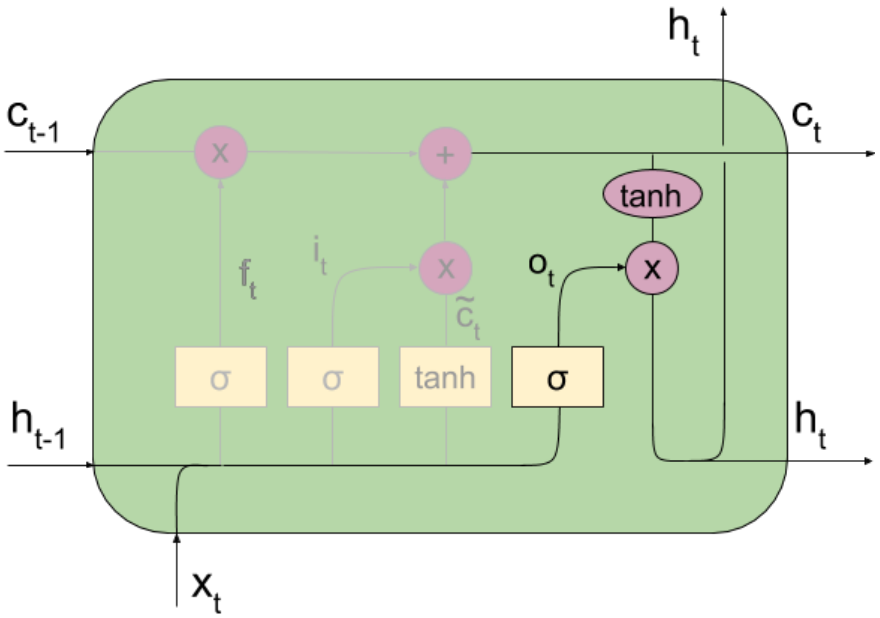
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_c), \quad (2)$$

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:

$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o \cdot [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o\right) \quad (3)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (4)$$

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 TensorFlow, the LSTM is implemented as another layer in the `tf.keras.layers` module:

In [12]:
```
import tensorflow as tf
```

In [13]:
```
# Build an LSTM model

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM

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

In [14]:
```
# Print the model summary

lstm.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, None, 16)          1856

 lstm_1 (LSTM)               (None, 16)                2112

=================================================================
Total params: 3968 (15.50 KB)
Trainable params: 3968 (15.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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 [15]:
```
# Build an LSTM model that returns its final internal state

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

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

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

lstm2.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 8, 4)]            0

 lstm_2 (LSTM)               [(None, 8, 6),            264
                              (None, 6),
                              (None, 6)]

=================================================================
Total params: 264 (1.03 KB)
Trainable params: 264 (1.03 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [17]:
```
# View the model outputs

lstm2.outputs
```

Out[17]:
```
[<KerasTensor: shape=(None, 8, 6) dtype=float32 (created by layer 'lstm_
2')>,
 <KerasTensor: shape=(None, 6) dtype=float32 (created by layer 'lstm_2')
>,
 <KerasTensor: shape=(None, 6) dtype=float32 (created by layer 'lstm_2')
>]
```

In [18]:
```
# Test the model on a dummy input

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

Out[18]:
```
[<tf.Tensor: shape=(1, 8, 6), dtype=float32, numpy=
 array([[[-0.03009245, -0.03507248, -0.0688527 , -0.01009785,
           0.05648322,  0.14905876],
         [-0.04633192, -0.09271716, -0.15814364, -0.05043248,
           0.13212055,  0.33866498],
         [-0.14151254, -0.04908141, -0.0789455 , -0.10347227,
           0.01492786,  0.10687232],
         [-0.26218152,  0.09769987,  0.02992789,  0.08786865,
           0.15505761,  0.10145675],
         [-0.38050023,  0.21848698, -0.02084113,  0.01585027,
           0.09084634,  0.0264043 ],
         [-0.1551278 ,  0.02773884, -0.09272097,  0.0908715 ,
           0.2090067 ,  0.07513203],
         [-0.03559721, -0.08252515, -0.23903099,  0.11785682,
           0.25037274,  0.20686252],
         [-0.16639878, -0.10387875, -0.13891618,  0.05744028,
           0.21010293,  0.12165201]]], dtype=float32)>,
 <tf.Tensor: shape=(1, 6), dtype=float32, numpy=
 array([[-0.16639878, -0.10387875, -0.13891618,  0.05744028,  0.2101029
3,
          0.12165201]], dtype=float32)>,
 <tf.Tensor: shape=(1, 6), dtype=float32, numpy=
 array([[-0.32546133, -0.24564439, -0.26976773,  0.15134288,  0.4865452
3,
          0.43546784]], 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 `tf.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 [19]:
```
import tensorflow as tf
```

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 [20]:
```
# Load the data

import pandas as pd

df = pd.read_csv('./data/tweets.csv')
```

```
print(df.shape)
df.head()
```

```
(14640, 15)
```

Out[20]:

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativere
0	570306133677760513	neutral	1.0000	
1	570301130888122368	positive	0.3486	
2	570301083672813571	neutral	0.6837	
3	570301031407624196	negative	1.0000	Bad
4	570300817074462722	negative	1.0000	Car

In [21]:
```
# Extract the relevant columns

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

In [22]:
```
# View a sample tweet and its label

# df.sample(1)
df.sample(1).values
```

Out[22]:
```
array([["@USAirways - I'm currently missing a basketball game due to you
r lost bag policy #terribleservice #wheresmyrefund",
        'negative', 1.0]], dtype=object)
```

In [23]:
```
# 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)
```

In [24]:
```
# Save the splits to CSV files

train_df.to_csv("./data/train.csv", index=False)
val_df.to_csv("./data/val.csv", index=False)
test_df.to_csv("./data/test.csv", index=False)
```

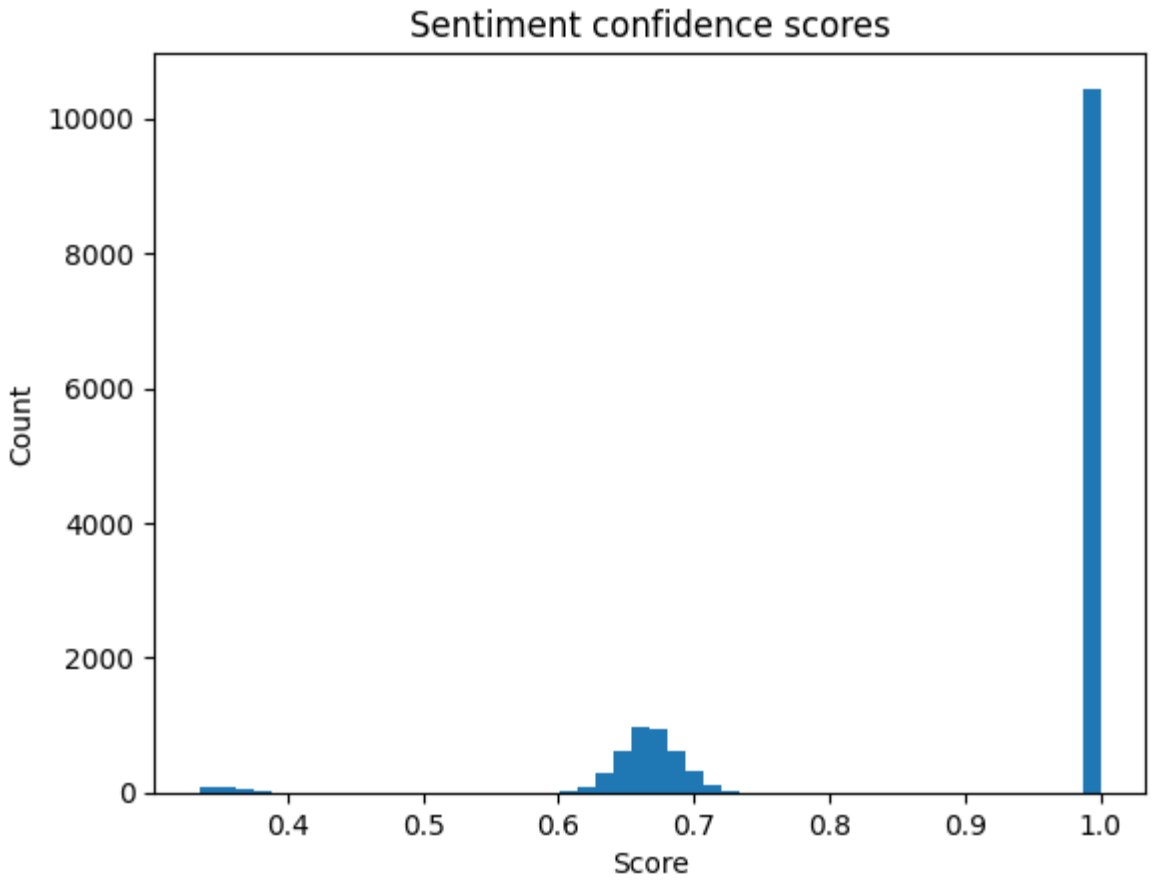
In [25]:
```
# Create Datasets from CSV files

train_dataset = tf.data.experimental.CsvDataset('./data/train.csv', [tf.s
                                                header=True)
```

```
val_dataset = tf.data.experimental.CsvDataset('./data/val.csv', [tf.strin
                                              header=True)
test_dataset = tf.data.experimental.CsvDataset('./data/test.csv', [tf.str
                                               header=True)
```

In [26]:
```
# View a sample from the training Dataset

for elem in train_dataset.take(1):
    print(elem)
```

```
(<tf.Tensor: shape=(), dtype=string, numpy=b'@united no, I have a pricey C
hase Mileage Plus CC.'>, <tf.Tensor: shape=(), dtype=string, numpy=b'neutr
al'>, <tf.Tensor: shape=(), dtype=float32, numpy=1.0>)
```

In [27]:
```
# 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()
```



In [28]:
```
# Filter out low confidence labels

def remove_low_confidence_labels(text, label, score):
    return tf.math.greater(score, 0.5)

train_dataset = train_dataset.filter(remove_low_confidence_labels)
val_dataset = val_dataset.filter(remove_low_confidence_labels)
test_dataset = test_dataset.filter(remove_low_confidence_labels)
```

In [29]:
```
# Create inputs and outputs

def text_label(text, label, score):
    return text, label

train_dataset = train_dataset.map(text_label)
val_dataset = val_dataset.map(text_label)
test_dataset = test_dataset.map(text_label)
```

In [30]:
```
# Shuffle the training Dataset

train_dataset = train_dataset.shuffle(500)
```

In [31]:
```
# View a sample from the training Dataset

for elem in train_dataset.take(1):
    print(elem)
```

```
(<tf.Tensor: shape=(), dtype=string, numpy=b'@united This must be a drone
\xe2\x80\x9c@united: @KeamBleam We understand your frustration. Our Bag te
am is working hard to get your bag(s) to you..."'>, <tf.Tensor: shape=(),
dtype=string, numpy=b'negative'>)
```

In [32]:
```
# Batch the Datasets

train_dataset = train_dataset.batch(16)
val_dataset = val_dataset.batch(16)
test_dataset = test_dataset.batch(16)
```

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.

First we will look at the output labels, and convert them to a one-hot encoding using the `StringLookup` and `CategoryEncoding` layers.

In [33]:
```
# Create a StringLookup layer

from tensorflow.keras.layers import StringLookup

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

In [34]:
```
# View the output of the StringLookup layer

for t, l in train_dataset.take(1):
    print(l)
    print(stringlookup(l))
```

```
tf.Tensor(
[b'negative' b'positive' b'negative' b'neutral' b'negative' b'negative'
 b'negative' b'negative' b'negative' b'positive' b'positive' b'negative'
 b'negative' b'negative' b'negative' b'negative'], shape=(16,), dtype=stri
ng)
tf.Tensor([1 0 1 2 1 1 1 1 1 0 0 1 1 1 1 1], shape=(16,), dtype=int64)
```

In [35]:
```
# Apply the label preprocessing to the Datasets

def convert_to_integer(text, labels):
    return text, stringlookup(labels)

train_dataset = train_dataset.map(convert_to_integer)
val_dataset = val_dataset.map(convert_to_integer)
test_dataset = test_dataset.map(convert_to_integer)

train_dataset = train_dataset.prefetch(tf.data.AUTOTUNE)
val_dataset = val_dataset.prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.prefetch(tf.data.AUTOTUNE)
```

Now we will process the input tweet text - this can be handled using the `TextVectorization` layer.

In [36]:
```
# Create a TextVectorization layer

from tensorflow.keras.layers import TextVectorization

textvectorization = TextVectorization(max_tokens=1000)
```

In [37]:
```
# Configure the layer to the dataset

textvectorization.adapt(train_dataset.map(lambda t, l: t))
```

In [38]:
```
# View the output of the TextVectorization layer

for t, l in train_dataset.take(1):
    print(t)
    print(textvectorization(t))
```

```
tf.Tensor(
[b'@VirginAmerica Why is it taking 12 years to fly home to Dallas? Get you
r shit together.'
 b'@united wanted to, but you Cancelled Flightled my tickets.... They were
too special i guess even for united.'
 b"@AmericanAir Flight's Cancelled Flightled. Website says to call phone n
umber.Phone says to check online.  How am I supposed to get some help?"
 b'@USAirways Is your refund system down?'
 b'@USAirways provided the best service for me today! Thank you so much
:)'
 b"\xe2\x80\x9c@AmericanAir: @TilleyMonsta George, that doesn't look good.
Please follow this link to start the refund process: http://t.co/4gr39s91D
l\xe2\x80\x9d\xf0\x9f\x98\x82"
 b'@AmericanAir this delayed bag was for my friend Lisa Pafe. She got her
bag after 3 days in Costa Rica. Issue no updates on your system.'
 b"@AmericanAir Not only was 5418 Late Flight, but we've been boarded and
waiting for over 30min. WTF?"
 b"@united For my Grandma Ella's 80th, she would love a birthday greeting
from your flight crew! She was a stewardess for Eastern Airlines."
 b'@USAirways your a miserable airline and your loss of revenue is a refle
ction of that.  Go extinct.  A merger only delays the inevitable'
 b"@united Male agnt in LAS threatens Canadian cust when cust takes pic of
him at gate after agents announce can't help rebook. #friendlyskies?"
 b"@AmericanAir we've been on hold for over 4 hrs, you Cancelled Flighted
flt 2222 phl-dfw. Need assistance!!!"
 b'@united you asked me to DM then ignore them. Please assist in changing
to another flight today- 21 Feb fra to mco avoiding IAD.'
 b'@united...lies lies lies....still sitting at the gate, have not moved a
n inch http://t.co/LulGnwEfFH'
 b'@JetBlue ah no the staff was perfect this morning at JFK, more sleep fo
r me!! :)'
 b'@USAirways Looking forward to some friends and family time in Arizon
a.'], shape=(16,), dtype=string)
tf.Tensor(
[[ 76  68  15  21 371 395 644   2 107 142   2 428  31  22 848 478   0   0
    0   0   0   0   0   0   0   0]
 [  6 631   2  32   7  35  81  12 294  54 103 164 921   4 508 134   8   6
    0   0   0   0   0   0   0   0]
 [ 14  60  35  81 240 227   2  89  99   1 227   2 158 195  70  98   4 423
    2  31 196  47   0   0   0   0]
 [ 13  15  22 279 241 276   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0]
 [ 13   1   3 220  42   8  20  96  65   7  46 178   0   0   0   0   0   0
    0   0   0   0   0   0   0   0]
 [  1   1   1  29 225 330 133  77 267  30 517   2 490   3 279 535   1   0
    0   0   0   0   0   0   0   0]
 [ 14  30  75  90  24   8  12 593   1   1 184  85 212  90  87 125 179  16
    1   1 245  28 646  10  22 241]
 [ 14  25 119  24   1  95   9  32 477  45 673  11 106   8 102   1 712   0
    0   0   0   0   0   0   0   0]
 [  6   8  12   1   1   1 184  73 177   5   1   1  33  22   9 180 184  24
    5   1   8   1 239   0   0   0]
 [ 13  22   5   1 101  11  22   1  19   1  15   5   1  19  29 113   1   5
  934 119 272   3   1   0   0   0]
 [  6   1   1  16 609   1   1 545  63 545 782   1  19 526  26  82  87 253
    1  72  47 274   1   0   0   0]
 [ 14 477  45  10  62   8 102 148 197   7  35 172 258   1   1  78 643   0
    0   0   0   0   0   0   0   0]
 [  6   7 628  20   2 153 109   1 147  77   1  16 690   2 138   9  96   1
  868   1   2 792   1 591   0   0]
```

```
[  1   1   1 194  26   3  82  23  25 681  43   1   1   0   0   0   0   0
   0   0   0   0   0   0   0   0]
[ 18   1  28   3 205  24   1  30 222  26 201 100 761   8  20   0   0   0
   0   0   0   0   0   0   0   0]
[ 13 291 528   2 196 704  11 440  48  16   1   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0]], shape=(16, 26), dtype=int64)
```

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 [39]:
```
# Create an Embedding layer

from tensorflow.keras.layers import Embedding

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

In [40]:
```
# View the output of the Embedding layer

for t, l in train_dataset.take(1):
    print(embedding(textvectorization(t)).shape)
```

```
(16, 31, 2)
```

In [41]:
```
# Build the classifier model

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, LSTM, Dense

lstm_classifier = Sequential([
    textvectorization,
    embedding,
    Bidirectional(LSTM(8)),
    Dense(3, activation='softmax')
])
```

In [42]:
```
# Compile and train the model

lstm_classifier.compile(optimizer='adam', loss='sparse_categorical_crosse
history = lstm_classifier.fit(train_dataset, validation_data=val_dataset,
```

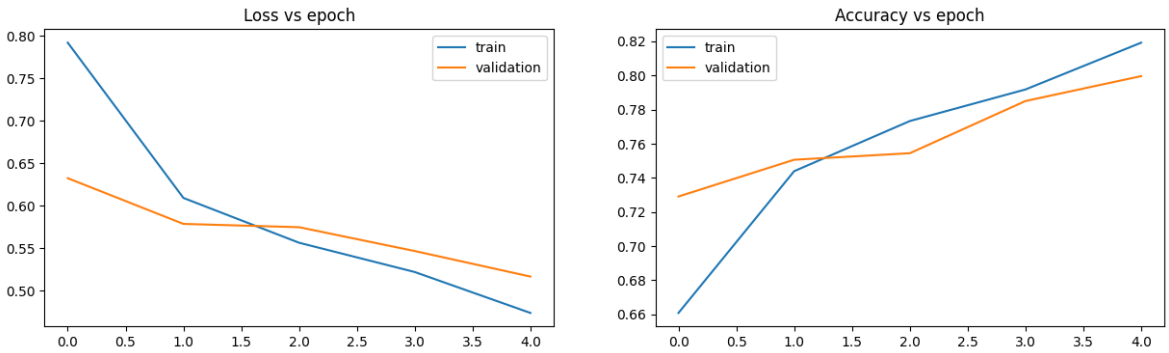
```
Epoch 1/5
541/541 [==============================] - 24s 24ms/step - loss: 0.7921 -
accuracy: 0.6609 - val_loss: 0.6326 - val_accuracy: 0.7291
Epoch 2/5
541/541 [==============================] - 10s 19ms/step - loss: 0.6093 -
accuracy: 0.7439 - val_loss: 0.5788 - val_accuracy: 0.7506
Epoch 3/5
541/541 [==============================] - 12s 21ms/step - loss: 0.5566 -
accuracy: 0.7733 - val_loss: 0.5749 - val_accuracy: 0.7544
Epoch 4/5
541/541 [==============================] - 11s 21ms/step - loss: 0.5221 -
accuracy: 0.7918 - val_loss: 0.5468 - val_accuracy: 0.7850
Epoch 5/5
541/541 [==============================] - 10s 18ms/step - loss: 0.4740 -
accuracy: 0.8192 - val_loss: 0.5168 - val_accuracy: 0.7996
```

In [43]:
```
# Evaluate the model on the test Dataset

lstm_classifier.evaluate(test_dataset)
```

```
180/180 [==============================] - 1s 8ms/step - loss: 0.5299 - ac
curacy: 0.7921
```

Out[43]: [0.5298656821250916, 0.7920722961425781]

In [44]:
```
# 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 [45]:
```
# View some model predictions on the test set

import numpy as np

for text, label in test_dataset.take(1):
    ground_truth = np.array(output_labels)[label]
```

```
    predicted_label_ints = np.argmax(lstm_classifier(text).numpy(), axis=
    predicted_labels = np.array(output_labels)[predicted_label_ints]

for t, l, g in zip(text, predicted_labels, ground_truth):
    print(t.numpy().decode('utf-8'))
    print("True label: {}\nPredicted label: {}\n".format(g, l))
```

```
@SouthwestAir I'm scheduled to come back tomorrow, finally but I don't kno
w SWA hasn't flown out of @Fly_Nashville yet but @Delta has
True label: neutral
Predicted label: negative

@JetBlue saving my sanity. Leaving it behind for sunshine. #escape #FL #bl
iss #travel #InDenial #WhatFrozenPipes http://t.co/6TtzEJV3hY
True label: neutral
Predicted label: neutral

@JetBlue Thx for the pointer, but Im good with the big monitor.  Any advic
e on who I should pick to come our of the Western Conf (NBA)?
True label: neutral
Predicted label: positive

@AmericanAir Orbitz has only shown "priority" seats since we booked &amp;
paid 3 weeks ago.
True label: negative
Predicted label: negative

@AmericanAir another generic response guys? Cmon. You're terrible. How abo
ut an actual helpful person? Not all the rude employees at LAX
True label: negative
Predicted label: negative

@AmericanAir I would love to say the same but it's not so much. 5hs fligh
t, no internet, no personal A/C, 20yr old plane. #NoFunAtAll
True label: negative
Predicted label: negative

@JetBlue I would love for you to fly my best friend home to PVD for a week
end. 😊 http://t.co/cH0NmjyMgh
True label: positive
Predicted label: positive

@USAirways I'm sure the people are working very hard, but the systems shou
ld assist with IVR prompting, offering call backs, hold wait times
True label: negative
Predicted label: negative

@SouthwestAir I'm on the 110 lol has anyone arrived yet??
True label: neutral
Predicted label: negative

@SouthwestAir my flight was Cancelled Flighted for tomorrow and hold times
are long can I Cancelled Flight a leg with you?
True label: negative
Predicted label: negative

@SouthwestAir okaaaaay UGH IM 8 MIN AWAY
True label: negative
Predicted label: neutral

@JetBlue we made it safe and sound. Thank you for the safe travels.
True label: positive
Predicted label: positive

@united do you teach your gate agents 2 lie? Or do they just learn on thei
r own? There was overhead space for my bag,  didn't have to check
True label: negative
```

```
Predicted label: negative

@AmericanAir we had to sit in the airport a long periolodicly time. the pl
anes were nasty
True label: negative
Predicted label: negative

@SouthwestAir just had a great flight #4223 with Damion! He was the best #
damionflight4223
True label: positive
Predicted label: positive

@united i think he actually did not like your screen  @campilley 😃😃😃
True label: negative
Predicted label: negative
```

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.