# CHAPTER 9: Recurrent Neural Networks

## Overview

In this chapter, you will learn how to handle real sequential data. You will extend your knowledge of **artificial neural network** (**ANN**) models and **recurrent neural network** (**RNN**) architecture for training sequential data. You will also learn how to build an RNN model with an LSTM layer for natural language processing.

By the end of this chapter, you will have gained hands-on experience of applying multiple LSTM layers to build RNNs for stock price predictions.

## Introduction

Sequential data refers to datasets in which each data point is dependent on the previous ones. Think of it like a sentence, which is composed of a sequence of words that are related to each other. A verb will be linked to a subject and an adverb will be related to a verb. Another example is a stock price, where the price on a particular day is related to the price of the previous days. Traditional neural networks are not fit for processing this kind of data. There is a specific type of architecture that can ingest sequences of data. This chapter will introduce you to such models—known as **recurrent neural networks** (**RNNs**).

An RNN model is a specific type of deep learning architecture in which the output of the model feeds back into the input. Models of this kind have their own challenges (known as vanishing and exploding gradients) that will be addressed later in the chapter.

In many ways, an RNN is a representation of how a brain might work. RNNs use memory to help them learn. But how can they do this if information only flows in one direction? To understand this, you'll need to first review sequential data. This is a type of data that requires a working memory to process data effectively. Until now, you have only explored non-sequential models, such as a perceptron or CNN. In this chapter, you will look at sequential models such as RNN, LSTM, or GRU.

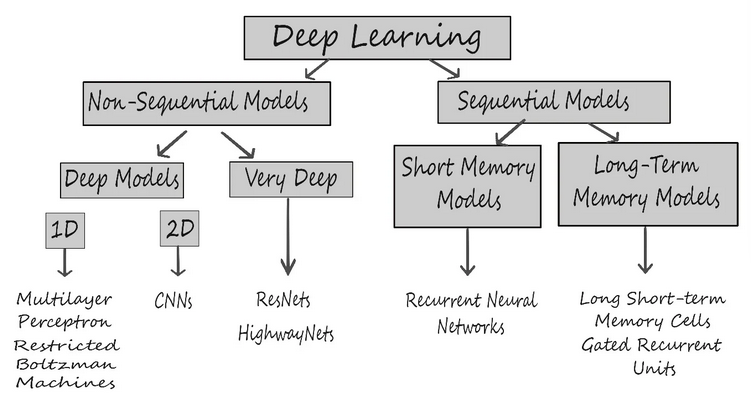


Figure 9.1: Sequential versus non-sequential models

## Sequential Data

Sequential data is information that happens in a sequence and is related to past and future data. An example of sequential data is time series data; as you perceive it, time only travels in one direction.

Suppose you have a ball (as in *Figure 9.2*), and you want to predict where this ball will travel next. If you have no prior information about the direction from which the ball was thrown, you will simply have to guess. However, if in addition to the ball's current location, you also had information about its previous location, the problem would be much simpler. To be able to predict the ball's next location, you need the previous location information in a sequential (or ordered) form to make a prediction about future events.



Figure 9.2: Direction of the ball

RNNs function in a way that allows the sequence of the information to retain value with the help of internal memory.

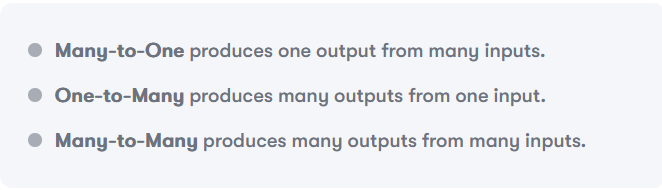
You'll look at some examples of sequential data in the following section.

## Examples of Sequential Data

Sequential data is a specific type of data where the order of each piece of information is important, and they all depend on each other.

One example of sequential data is financial data, such as stock prices. If you want to predict future data values for a given stock, you need to use previous values in time. In fact, you will work on stock prediction in *Exercise 9.01*, *Training an ANN for Sequential Data – Nvidia Stock Prediction*.

Audio and text can also be considered sequential data. Audio can be split up into a sequence of sound waves, and text can be split up into sequences of either characters or words. The sound waves or sequences of characters or words should be processed in order to convey the desired result. Beyond these two examples that you encounter every day, there are many more examples in which sequential processing may be useful, from analyzing medical signals such as EEGs, projecting stock prices, and inferring and understanding genomic sequences. There are three categories of sequential data:



Diagram

Description automatically generated

Figure 9.3: Categories of sequential data

Consider another example. Suppose you have a language model with a sentence, or a phrase and you are trying to predict the word that comes next, as in the following figure:

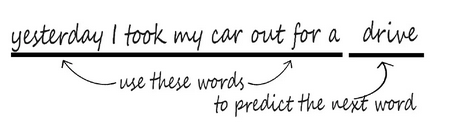


Figure 9.4: Sentence example

Say you're given the words yesterday I took my car out for a…, and you want to try to predict the next word, drive. One way you could do this is by building a deep neural network such as a feed-forward neural network. However, you would immediately run into a problem. A feed-forward network can only take a fixed-length input vector as its input; you have to specify the size of that input right from the start.

Because of this, your model needs a way to be able to handle variable-length inputs. One way you can do this is by using a fixed window. That means that you force your input vector to be just a certain length. For example, you can split the sentence into groups of two consecutive words (also called a bi-gram) and predict the next one. This means that no matter where you're trying to make that next prediction, your model will only be taking in the previous two words as its input. You need to consider how you can numerically represent this data. One way you can do this is by taking a fixed-length vector and allocating some space in that vector for the first word and some space in that vector for the second word. In those spaces, encode the identity of each word. However, this is problematic.

Why? Because you're using only a portion of the information available (that is, two consecutive words only). You have access to a limited window of data that doesn't give enough context to accurately predict what will be the next word. That means you cannot effectively model long-term dependencies. This is important in sentences like the one in Figure 9.5 where you clearly need information from much earlier in the sentence to be able to accurately predict the next word.

Figure 9.5: Sentence example


Figure 9.5: Sentence example

If you were only looking at the past two or three words, you wouldn't be able to make this next prediction, which you know is Italian. So, this means that you really need a way to integrate the information in the sentence from start to finish.

To do this, you could use a set of counts as a fixed-length vector and use the entire sentence. This method is known as bag of words.

You have a fixed-length vector regardless of the identity of the sentence, but what differs is adding the counts over this vocabulary. You can feed this into your model as an input to generate a prediction.

However, there's another big problem with this. Using just the counts means that you lose all sequential information and all information about the prior history.

Consider Figure 9.6. So, these two sentences, which have completely opposite semantic meanings would have the exact same representations in this bag of words format. This is because they have the exact same list of words, just in a different order. So, obviously, this isn't going to work. Another idea could be simply to extend the fixed window.

Figure 9.6: Bag of words example


Figure 9.6: Bag of words example

Now, consider Figure 9.7. You can represent your sentence in this way, feed the sentence into your model, and generate your prediction. The problem is that if you were to feed this vector into a feed-forward neural network, each of these inputs, yesterday I took my car, would have a separate weight connecting it to the network. So, if you were to repeatedly see the word yesterday at the beginning of the sentence, the network may be able to learn that yesterday represents a time or a setting. However, if yesterday were to suddenly appear later in that fixed-length vector, at the end of a sentence, the network may have difficulty understanding the meaning of yesterday. This is because the parameters that are at the end of a vector may never have seen the term yesterday before, and the parameters from the beginning of the sentence weren't shared across the entire sequence.

Figure 9.7: Sentence example


Figure 9.7: Sentence example

So, you need to be able to handle variable-length input and long-term dependencies, track sequential order, and have parameters that can be shared across the entirety of your sequence. Specifically, you need to develop models that can do the following:

* Handle variable-length input sequences.
* Track long-term dependencies in the data.
* Maintain information about the sequence's order.
* Share parameters across the entirety of the sequence.

How can you do this with a model where information only flows in one direction? You need a different kind of neural network. You need a recursive model. You will practice processing sequential data in the following exercise.

## Example 9.01: Training and ANN for Sequential Data – Apple Stock Prediction

In this exercise, you will build a simple ANN model to predict the Apple stock price. But unlike examples from previous chapters, this time the input data is sequential. So, you need to manually do some processing to create a dataset that will contain the price of the stock for a given day as the target variable and the price for the previous 60 days as features. You are required to split the data into training and testing sets before and after the date

First, instantiate a pre-trained MobileNetV2 model without the top layer:

*from tensorflow.keras.applications import MobileNetV2*

*base\_model = MobileNetV2(input\_shape=(224, 224, 3), \*

*weights='imagenet', include\_top=False)*

1. Next, iterate through the first layers and freeze them by setting them as non-trainable. In the following example, you will freeze only the first 100 layers:

*for layer in base\_model.layers[:100]:*

*layer.trainable = False*

1. Now you need to add your custom top layer to your base model. In the following example, you will be predicting 20 different classes, so you need to add a fully connected layer of 20 units with the softmax activation function:

*prediction\_layer = tf.keras.layers.Dense(20, activation='softmax')*

*model = tf.keras.Sequential([base\_model, prediction\_layer])*

1. Finally, you will compile and then train this model:

*model.compile(loss='sparse\_categorical\_crossentropy', \*

*optimizer = tf.keras.optimizers.Adam(0.001))*

*model.fit(features\_train, label\_train, epochs=5)*

This will display a number of logs, as seen in the following screenshot:



Figure 8.5: Fine-tuning results on a pre-trained MobileNetV2 model

That’s it.You have just performed fine-tuning on a pre-trained MobileNetV2 model. You have used the first 100 pre-trained weights from ImageNet and only updated the weights from layer 100 onward according to your dataset.

In the next activity, you will put into practice what you have just learned and apply fine-tuning to a pre-trained model.

## Activity 8.01: Food Classification with Fine-Tuning

The “**10\_food\_classes\_all\_data”** dataset

<https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip>

This data set consists of 10 food categories, with 10,000 images. For each class, 250 manually reviewed test images are provided as well as 750 training images. On purpose, the training images were not cleaned, and thus still contain some amount of noise. This comes mostly in the form of intense colors and sometimes wrong labels. All images were rescaled to have a maximum side length of 512 pixels. Our aim is to train a deep learning model which can successfully classify food images.

In this activity, you are tasked with training a NASNetMobile model to recognize images of different varieties of foods (classification into 10 different classes). You will use fine-tuning to train the final layers of this model.

Follow these steps will help you to complete this activity:

1. Import the dataset and unzip the file using TensorFlow.
2. Create a data generator with the following data augmentation:

*Rescale = 1./255,*

*rotation\_range = 40,*

*width\_shift\_range = 0.1,*

*height\_shift\_range = 0.1,*

*shear\_range = 0.2,*

*zoom\_range = 0.2,*

*horizontal\_flip = True,*

*fill\_mode = 'nearest*

1. Load a pre-trained NASNETMobile model from TensorFlow.
2. Freeze the first 600 layers of the model.
3. Add two fully connected layers on top of NASNETMobile

* A fully connected layer with Dense(1000, activation=relu)
* A fully connected layer with Dense(10, activation=’softmax’)

1. Specify an Adam Optimizer with a learning rate 0.001
2. Train the model.
3. Evaluate the model on the test set.

The expected output is as follows:

A picture containing text

Description automatically generated

Figure 8.6: Expected output of the activity

Note

The solution to this activity can be found via this [link](https://github.com/fenago/tf/blob/main/Chapter8-Pre-Traine_Networks/Activity%208.01-%20Food%20Classification%20with%20Fine-Tuning.ipynb)

Now that you know how to use pre-trained models from TensorFlow, you will learn how models can be accessed from TensorFlow Hub in the following section.

## TensorFlow Hub

TensorFlow Hub is a repository of TensorFlow modules shared by publishers such as Google, NVIDIA, and Kaggle. TensorFlow modules are self-contained models built on TensorFlow that can be reused for different tasks. Put simply, it is an external collection of published TensorFlow modules for transfer learning and fine-tuning. With TensorFlow Hub, you can access different deep learning models or weights than the ones provided directly from TensorFlow's core API.

Note

You can find more information about TensorFlow Hub here: <https://tfhub.dev/>.

In order to use it, you first need to install it:

*Pip install tensorflow-hub*

Once it's installed, you can load available classification models with the load() method by specifying the link to a module:

*import tensorflow\_hub as hub*

*MODULE\_HANDLE = 'https://tfhub.dev/tensorflow/efficientnet'\*

*'/b0/classification/1'*

*module = hub.load(MODULE\_HANDLE)*

In the preceding example, you have loaded the **EfficientNet B0** model, which was trained on ImageNet. You can find more details on this at the TensorFlow Hub page: <https://tfhub.dev/tensorflow/efficientnet/b0/classification/1>.

Note

TensorFlow Hub provides a search engine to find a specific module: <https://tfhub.dev/s?subtype=module,placeholder>.

By default, modules loaded from TensorFlow Hub contain the final layer of a model without an activation function. For classification purposes, you need to add an activation layer of your choice. To do so, you can use the Sequential API from Keras. You just need to convert your model into a Keras layer with the KerasLayer class:

*import tensorflow as tf*

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE,input\_shape=(224, 224, 3)),*

*tf.keras.layers.Activation('softmax')*

*])*

Then, you can use your final model to perform predictions:

*model.predict(data)*

You just performed transfer learning with a model from TensorFlow Hub. This is very similar to what you learned previously using the Keras API, where you loaded an entire model with include\_top=True. With TensorFlow Hub, you can access a library of pre-trained models for object detection or image segmentation.

In the next section, you will learn how to extract features from TensorFlow Hub pre-trained modules.

## Feature Extraction

TensorFlow Hub provides the option of downloading a model without the final layer. In this case, you will be using a TensorFlow module as a feature extractor; you can design your custom final layers on top of it. In TensorFlow Hub, a module used for feature extraction is known as a feature vector:

*import tensorflow\_hub as hub*

*MODULE\_HANDLE = 'https://tfhub.dev/google/efficientnet/b0'\*

*'/feature-vector/1'*

*module = hub.load(MODULE\_HANDLE)*

Note

To find all the available feature vectors on TensorFlow Hub, you can use its search engine: <https://tfhub.dev/s?module-type=image-feature-vector&tf-version=tf2>.

Once loaded, you can add your own final layer to the feature vector with the Sequential API:

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE, input\_shape=(224, 224, 3)),*

*tf.keras.layers.Dense(20, activation='softmax')*

*])*

In the preceding example, you added a fully connected layer of 20 units with the softmax activation function. Next, you need to compile and train your model:

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE, input\_shape=(224, 224, 3)),*

*tf.keras.layers.Dense(20, activation='softmax')*

*])*

And with that, you just used a feature vector from TensorFlow Hub and added your custom final layer to train the final model on your dataset.

Now, test the knowledge you have gained so far in the next activity.

## Activity 8.02: Transfer Learning with TensorFlow Hub

In this activity, you are required to correctly classify images of pizzas and steaks using transfer learning. Rather than training a model from scratch, you will benefit from the **EfficientNet B0** feature vector from TensorFlow Hub, which contains pre-computed weights that can recognize different types of objects.

You can find the dataset here

<https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip>

The following steps will help you to complete this activity:

1. Import the dataset and unzip the file using TensorFlow.
2. Create a data generator that will perform rescaling.
3. Load a pre-trained **EfficientNet B0** feature vector from TensorFlow Hub.
4. Add two fully connected layers on top of the feature vector:

– A fully connected layer with Dense(500, activation=relu)

– A fully connected layer with Dense(1, activation='sigmoid')

1. Specify an Adam optimizer with a learning rate of 0.001.
2. Train the model.
3. Evaluate the model on the test set.

The expected output is as follows:

A picture containing table

Description automatically generated

Figure 8.7: Model training output

The expected accuracy scores should be around 1.0 for the training and validation sets.

Note

The solution to this activity can be found via [this link](https://github.com/fenago/tf/blob/main/Chapter8-Pre-Traine_Networks/Activity%208.02-%20Transfer%20Learning%20with%20TensorFlow%20Hub.ipynb).

## Summary

In this chapter, you learned two very important concepts: transfer learning and fine-tuning. Both help deep learning practitioners to leverage existing pre-trained models and adapt them to their own projects and datasets.

Transfer learning is the re-use of models that have been trained on large datasets such as ImageNet (which contains more than 14 million images). TensorFlow provides a list of such pre-trained models in its core API. You can also access other models from renowned publishers such as Google and NVIDIA through TensorFlow Hub.

Finally, you got some hands-on practice fine-tuning a pre-trained model. You learned how to freeze the early layers of a model and only train the last layers according to the specificities of the input dataset.

These two techniques were a major breakthrough for the community as they facilitated access to state-of-the-art models for anyone interested in applying deep learning models.

In the next chapter, you will look at another type of model architecture, **recurrent neural networks** (**RNNs**). This type of architecture is well suited for sequential data such as time series or text.