

Deep Learning



Deep Learning

Lecture: Introduction to Sequence Data

Ted Scully

Sequence Data

A challenging category of problem for Machine Learning is identifying patterns in **sequence data**, such as text, spoken words, genomic data, handwriting, times series data etc.

Common example problems include:

- **Document classification:** Identification of the topic of an article or the author of a book
- **Sequence-to-sequence learning:** For example, translating an English sentence into Spanish
- **Text summarization:** When provided with a text document your objective is to create another shorter document that summarizes the main points of the first document.
- **Time series forecasting:** For example, predicting the future weather at a certain location, given recent weather data.

Natural Language Processing

Natural language processing is a very common type of sequential problem.

Like all other neural networks, deep-learning models don't take as input raw text as they only work with numerical tensors. Therefore, we must vectorise the text.

Vectorizing text is the process of transforming text into numeric tensors, which typically involves:

- Segmenting text into **words** (often referred to as tokens). This process is referred to as **tokenization**.
- Transform each token into a **vector**. This process is referred to as **vectorization** and involved associating numeric vectors with the generated tokens.

Natural Language Processing

While there are multiple methods of vectorization the two most widely used are **one-hot-encoding** and **word embeddings**. Let's use the IMDB dataset (which is built into Keras as an example).

The IMDB dataset contains **50,000** movie reviews split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Below we load the IMDB dataset.

```
import tensorflow as tf

(train_data, train_labels), (test_data, test_labels) =
tf.keras.datasets.imdb.load_data(num_words=10000)
```

num_words: max number of words to include. Words are ranked by how often they occur (in the training set) and only the most frequent words are kept

Natural Language Processing

While there are multiple methods of vectorization the two most widely used are **one-hot-encoding** and **word embeddings**. Let's use the IMDB dataset (which is built into Keras as an example).

The IMDB dataset contains **50,000** movie reviews split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Below we load the IMDB dataset.

```
import tensorflow as tf

(train_data, train_labels), (test_data, test_labels) =
tf.keras.datasets.imdb.load_data(num_words=10000)

print (type(train_data))
# Num words in first two reviews
print (len(train_data[0]), len(train_data[1]))

# first ten words in first review
print (train_data[0][:10])
```

The training data is a numpy array that contains 25000 NumPy arrays, each contains integer values and represents the words in one document.

You will notice that all words have already been converted to integer values.

```
<class 'numpy.ndarray'>
218 189
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

One Hot Encoding for Vectorization

- With the neural networks we have looked at so far, in order to process a sequence or a temporal series of data points (such as each review shown previously), you have to show the **entire sequence** (each sequence corresponding to one movie review) **to the network at once**: turning it into a single data point.
- If we were classifying a movie review using **one hot encoding** we would have to transform an entire movie review into a **single large vector** (length of vector equals vocabulary size) and feed that to a neural network.
- In contrast when humans read a sentence we process it word by word keeping a memory of the words that have gone before and after.
- Notice the structure above means we are only indicating **presence of absence of words** (we lose the exact sequence of the words)

One Hot Encoding for Vectorization

- In the IMDB example, we were using a **vocabulary size** of 10000 words.
- Therefore, even though as we observed each movie review may have a different size we could translate each individual review into a **fixed size vector** with 10,000 elements.
- In the basic example below we have a vocabulary of 6 words. We can translate each of the reviews into a six element vector.

2, 5, 6, 1  1, 1, 0, 0, 1, 1

2, 4, 3  0, 1, 1, 1, 0, 0

Limitations of Traditional NN Architectures

```
import tensorflow as tf
import numpy as np

def vectorize_sequences(sequences, dimension):

    results = np.zeros((len(sequences), dimension))

    for i, sequence in enumerate(sequences):

        results[i, sequence] = 1.0
    return results

NUM_WORDS = 10000
(train_data, train_labels), (test_data, test_labels) = tf.keras.datasets.imdb.load_data(num_words=NUM_WORDS)

x_train = vectorize_sequences(train_data, NUM_WORDS)
x_test = vectorize_sequences(test_data, NUM_WORDS)

print (x_train.shape)
print (x_train[0])
```

In this code we perform the same transformation that was undertaken in the previous slide.

Notice the shape of our training is now 25000 reviews by 10000 vocab size.

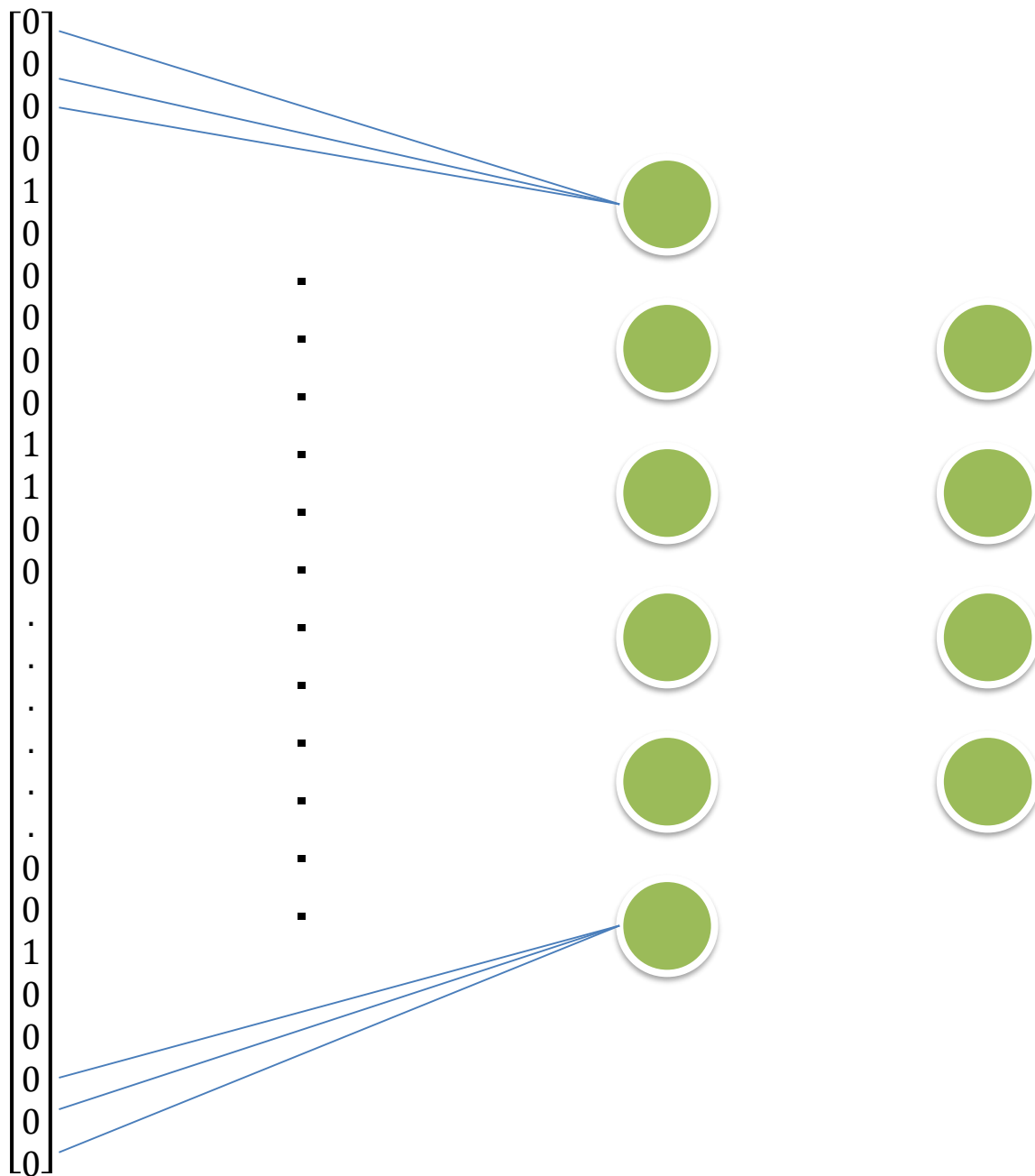
Each review is represented as a fixed size 10000 element vector.

```
(25000, 10000)
[0. 1. 1. ... 0. 0. 0.]
```


Here we illustrate a single review being fed into a standard densely connected network.

With such networks, in order to process a sequence or a temporal series of data points, you have to show the **entire sequence** to the network at once: turn it into a single data point.

Notice we now **lose all information related to the sequence of words**. For example, if “not good” was a sequence of words in the review this would just show that the word good and the word not occur somewhere within the review.



Vectorization – One Hot Encoding

One-hot encoding is a common and basic way of turning **individual tokens into a vector**.

As we saw with densely connected network we represented each document (movie review) as a single vector.

With **recurrent networks** we will be feeding each individual **word** into the recurrent network as a **sequence**. Therefore, we can encode each word as a one hot encoded vector. (The distinction here is that each one hot encoding vector just represents a single word, in the previous example, each one hot encoding vector represented a document.)

This just involves associating a unique integer index with every word and then turning this integer index i into a binary vector of size N (the size of the vocabulary); the vector is all zeros except for the i th entry, which is 1.

<surf waves car jeep>

<surf >

0	0	0	1
---	---	---	---

<waves>

0	0	1	0
---	---	---	---

<car>

0	1	0	0
---	---	---	---

<jeep>

1	0	0	0
---	---	---	---

Vectorization – One Hot Encoding

The problem with one-hot-encoding is it conveys little information about the relationship between each the values encoded.

In the example below clearly tennis and badminton would be as closely related than tennis and car. Likewise car and jeep would be more closely related than jeep and badminton.

However, each of the encoded values are equally distant from each other when placed in a one hot encoded format.

Likewise the **size of each vector can be very large** (the same size as the vocabulary)

<surf waves car jeep>

<tennis>	0	0	0	1
<badminton>	0	0	1	0
<car>	0	1	0	0
<jeep>	1	0	0	0

Deep Learning



Deep Learning

Lecture: Word Embeddings

Ted Scully

Vectorization – Word Embeddings

Another powerful method of vectorization is to learn a feature-based representation of each of the tokens and represent them as dense word vectors, called word embeddings.

- You will have noticed one-hot encoding vectors are **binary and sparse** and depending on the number of words you can end up with a very high-dimensional vector. For example, if your vocabulary is 20,000 words then each word is encoded as a vector of 20000 values.
- In contrast word embeddings are lower dimensional floating-point vectors that encode a much **richer** feature encoding of the words and their semantic meaning.
- Word embeddings are learned from the data.

0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0.1	-0.7	0.3	1.0	0.8	-0.1
-----	------	-----	-----	-----	------

Vectorization – One Hot Encoding

- A sample one hot encoding is shown below.
- Notice the representation of each word is binary and sparse.
- The presentation is of course because each word is equally similar to every other word.

Surf	Wave	Car	Jeep
0	1	0	0
1	0	0	0
0	0	1	0
0	0	0	1

Word Embeddings

- Word embeddings aim to map semantic meaning into a geometric space. This is done by defining each word in our vocabulary as a fixed sized vector, such that the distance between any two vectors captures part of the semantic relationship between the two associated words.
- Take the example below where we learn to represent each word in the vocabulary using word embeddings..

	Surf	Wave	Car	Jeep
Feature 1	-0.9	-0.8	0.9	0.95
Feature 2	0.8	0.7	-0.5	-0.7
Feature 3	0.8	0.9	0.2	0.4

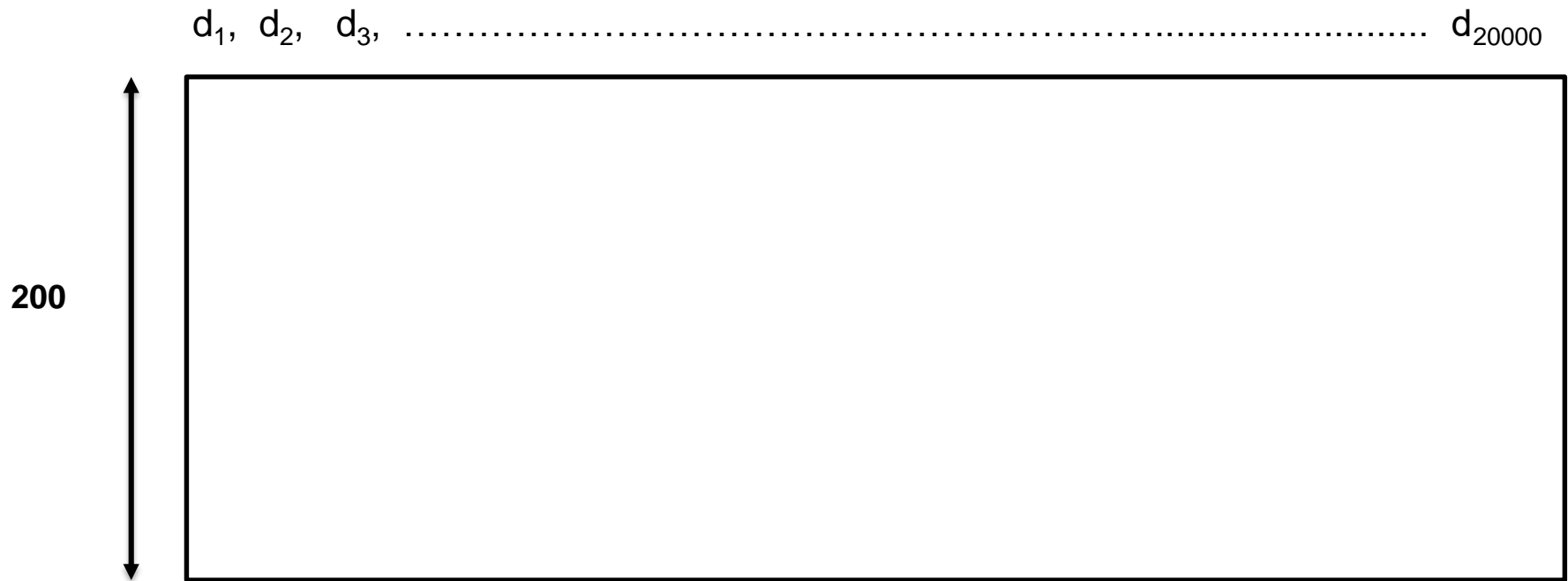
- Notice the distance between Surf and Wave is less than the distance between Surf and Car or Surf and Jeep.

Learning Word Embeddings

- Word embeddings are typically determined by learning word-occurrence statistics (observations about what words co-occur in sentences or documents).
- There are two ways that you can incorporate word embeddings:
- Learn word embeddings as part of the NLP task that you are implementing (such as document or sentiment classification). When taking this approach you start with a random word embedding matrix and subsequently learn word vectors in the same way you learn the weights of a neural network.
- Load into your model pre-trained word embeddings. This is a type of **transfer learning**.

To gain an understanding of words embeddings consider the matrix depicted below. We are working with a **vocabulary size of 20,000 words**. We want to encode each word as a **vector of size 200**.

This matrix is referred to as the **embedding matrix**. Originally all the values in this matrix are randomly initialized. Each word in your vocabulary will not be defined as a 20,000 element one hot vector but instead as a 200 element embedded vector. We refer to the matrix below as EM.



Common Bag Of Words (CBOW) – Word2Vec

- Word2Vec is one of the most popular methods of learning word embeddings. Full details can be found in the following 2013 paper - [Efficient Estimation of Word Representations in Vector Space](#) by Tomas Mikolov et al.
- Effectively Word2Vec takes a text corpus and allows to us to learn an embedding matrix.
- There are **two variants** (model architectures) of Word2Vec proposed in the paper (Skip Grams and Common Bag of Words (CBOW)).
- The following slides describe the CBOW variant.
- **The objective of CBOW is to take the words that represent the context of a target word as the input and to use these to predict the target word.**

CBOW – Word2Vec

The objective of CBOW is to take the words that represent the context of a target word as the input and to use these to predict the target word.

- Let's assume we had the following sentence in our text corpus.
- I like to surf when the swell and period are big.
- Let's assume the target word is swell.
- The objective of CBOW is to create a model that can take in the context words (maybe surf, period) and use that to predict the target word.

CBOW – Creating a Training Dataset

- First we need to build a supervised training dataset containing context words and mappings to target words.
- I like to surf when the swell and period are big.
- To create a supervised classification problem we identify what is referred to as a target and context from the sentence.
- For example let's assume we pick a target word that might be swell. We then pick a context by randomly selecting a word within a certain word window of the target (perhaps within 1, 2, 3, ... words from the target). Therefore, the context word might end up being surf or period for example.
- This is how we build up our supervised training set.
- So given a word context we want our neural network to predict the target word. Not easy. Note we don't expect the model to perform well on this task. The objective is to learn good word embeddings.

CBOW (Window Size = 1) Word2Vec

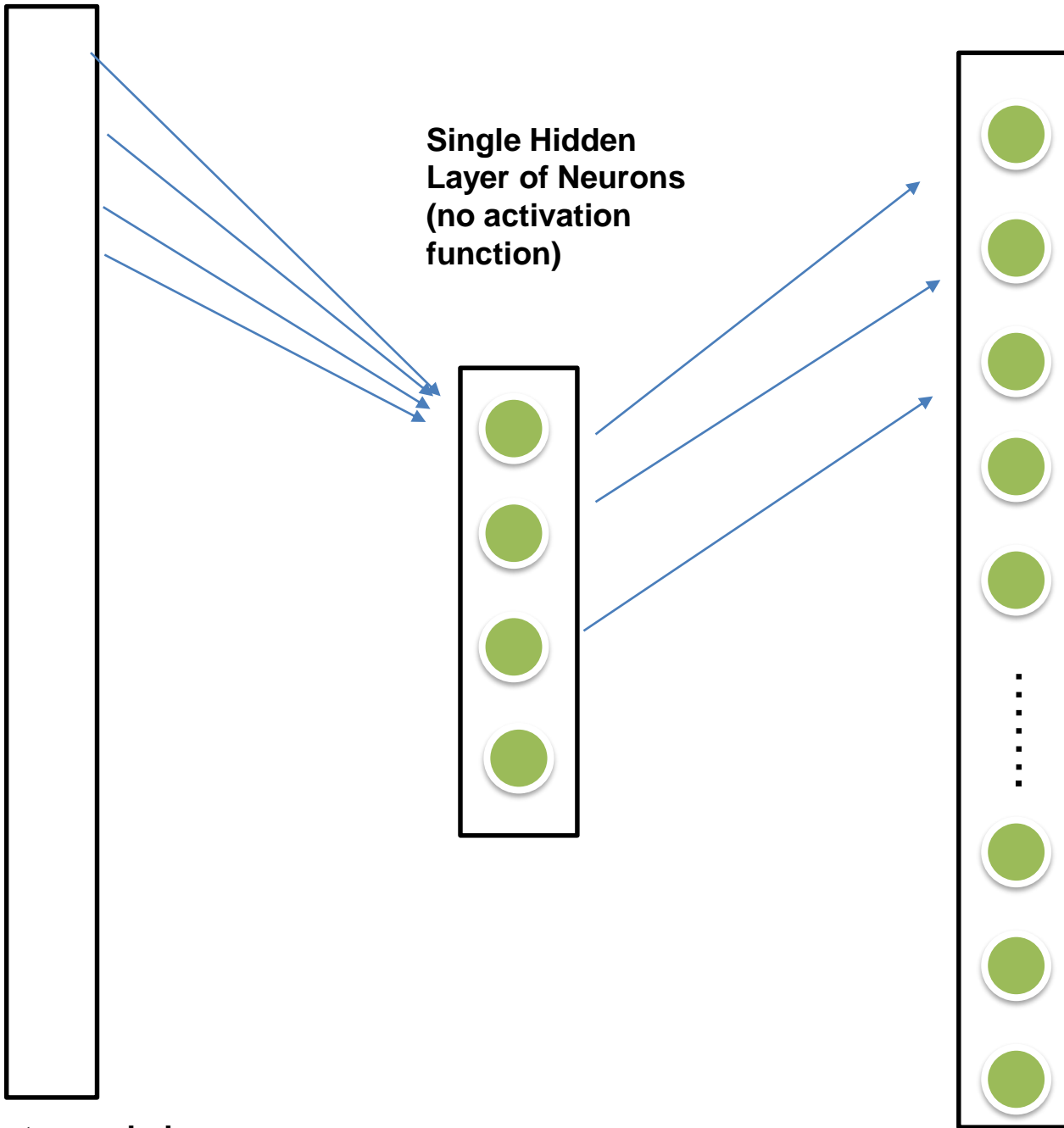
- Let's take a very basic example. We have two sentences.
- Surf big waves
- Drive fast cars
- If we employ a **CBOW** approach with a window size of 1 then we can create the following supervised training set.
- Once we have the dataset we can train our Word2Vec model with the context as input and the target as the correct output

Context	Target
surf	big
big	surf
big	waves
waves	big
drive	fast
fast	drive
fast	cars
cars	fast

One hot encoded
version of a word

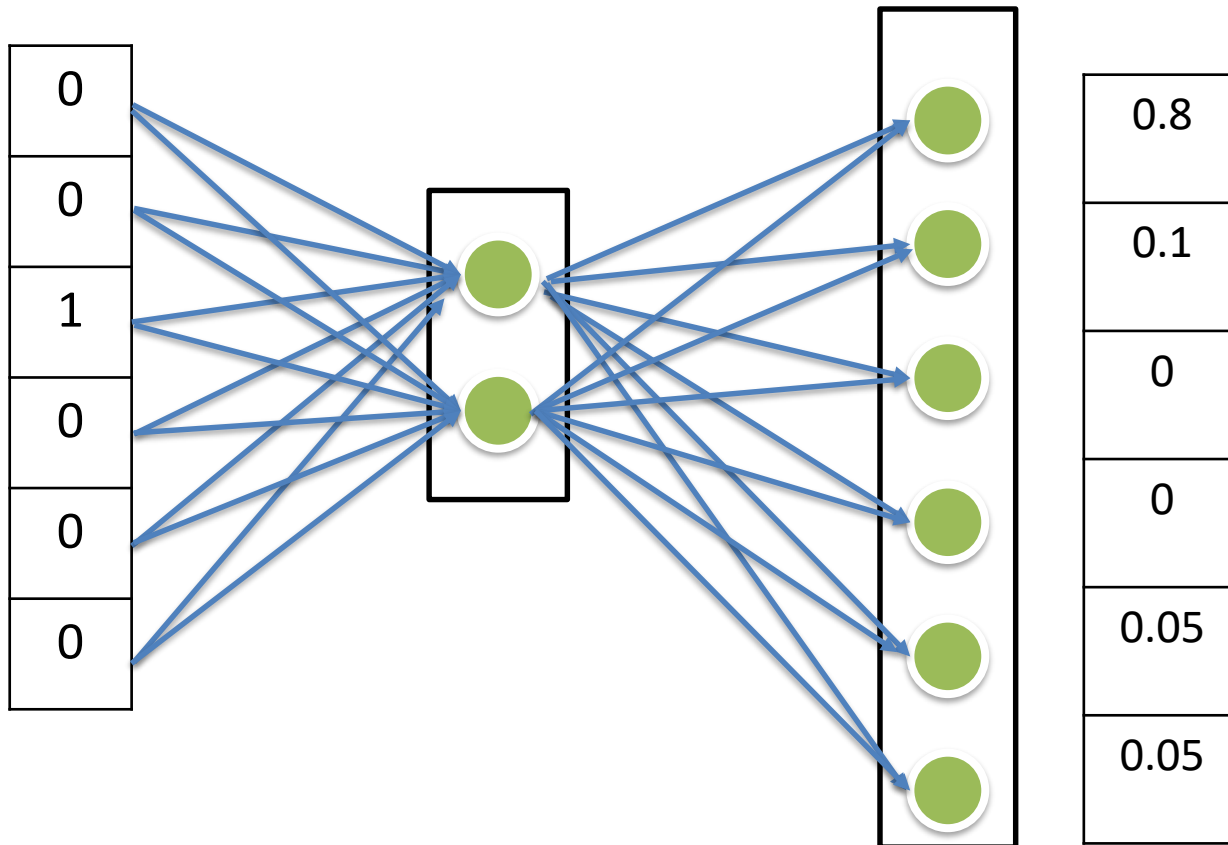
Single Hidden
Layer of Neurons
(no activation
function)

Softmax layer with
the same number
of neurons as
there are words in
the vocab.



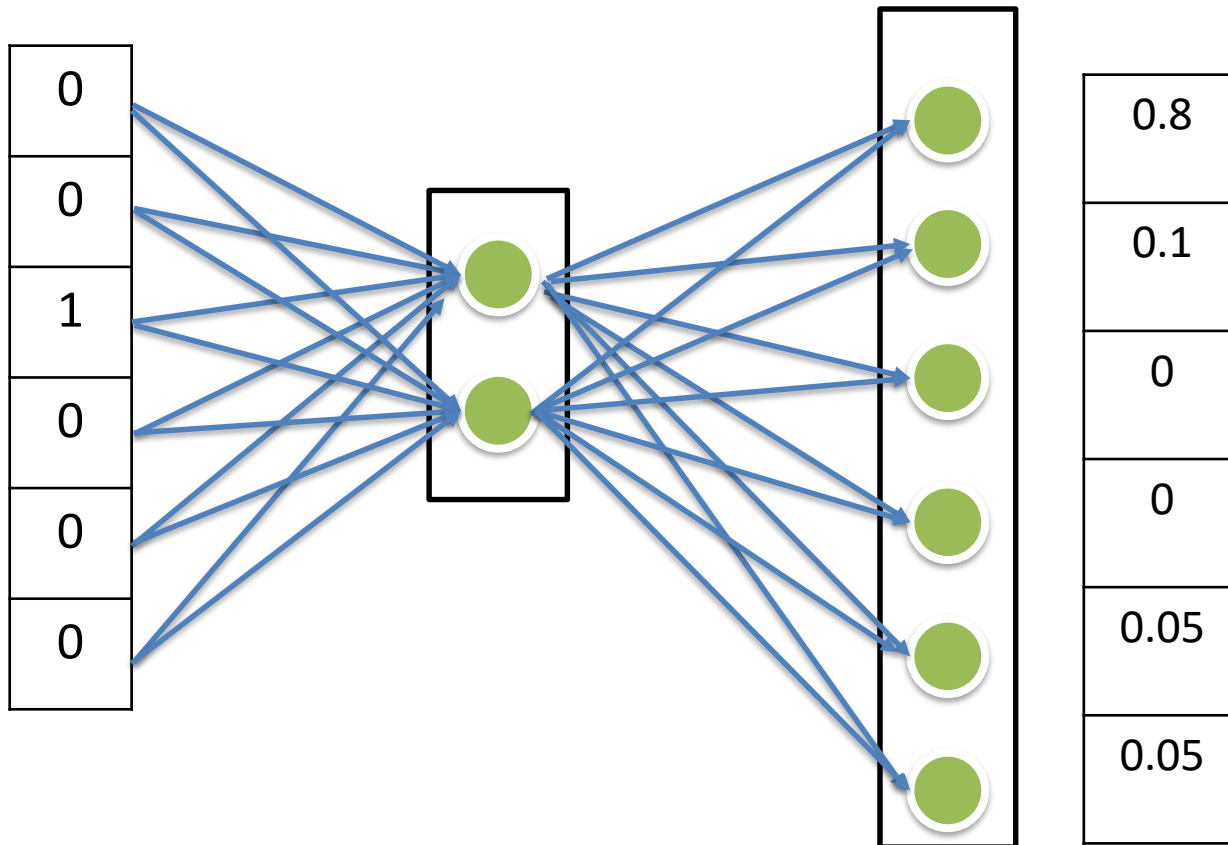
CBOW (Window Size = 1) Word2Vec

- In this example we may be inputting the **one hot representation of the word fast**. It is multiplied by the weight of the first hidden layer (no activation function). The results are then passed to a Softmax layer. We use normal cross entropy to calculate the error between the predicted one hot representation and the correct value which could be **cars**. This error is back propagated in the normal way.



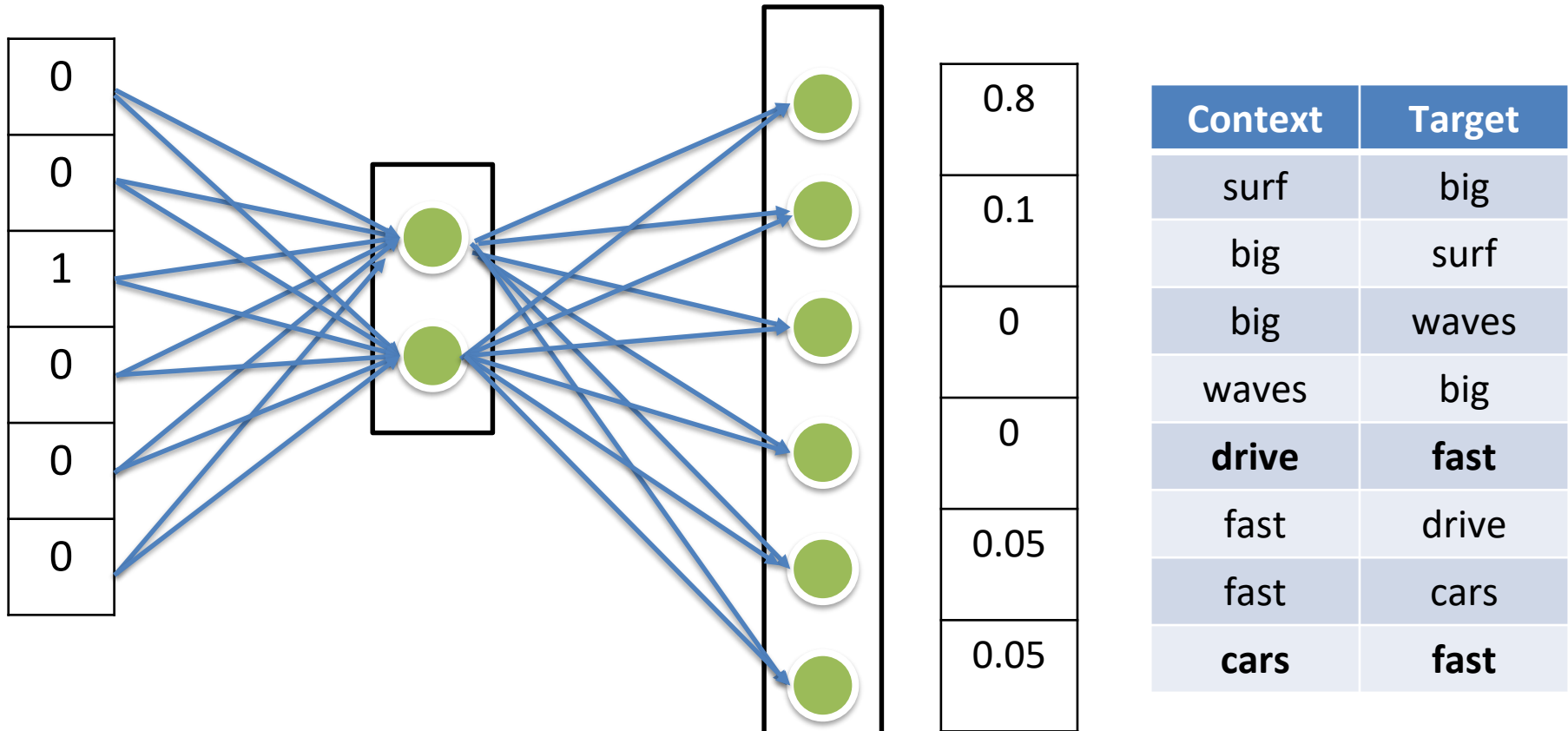
Obtain the Embedding Matrix

- Notice that the hidden layer in the network below **forces the compression of the one hot vector representation** to a 2 element vector representation. Once we have finished the training process we can **obtain our embedding matrix (EM)** by pushing each one hot vector representation through the hidden layer and collecting the output of the hidden layer. Each 6 element one-hot-encoded vector is transformed into a 2 element embedded vector.

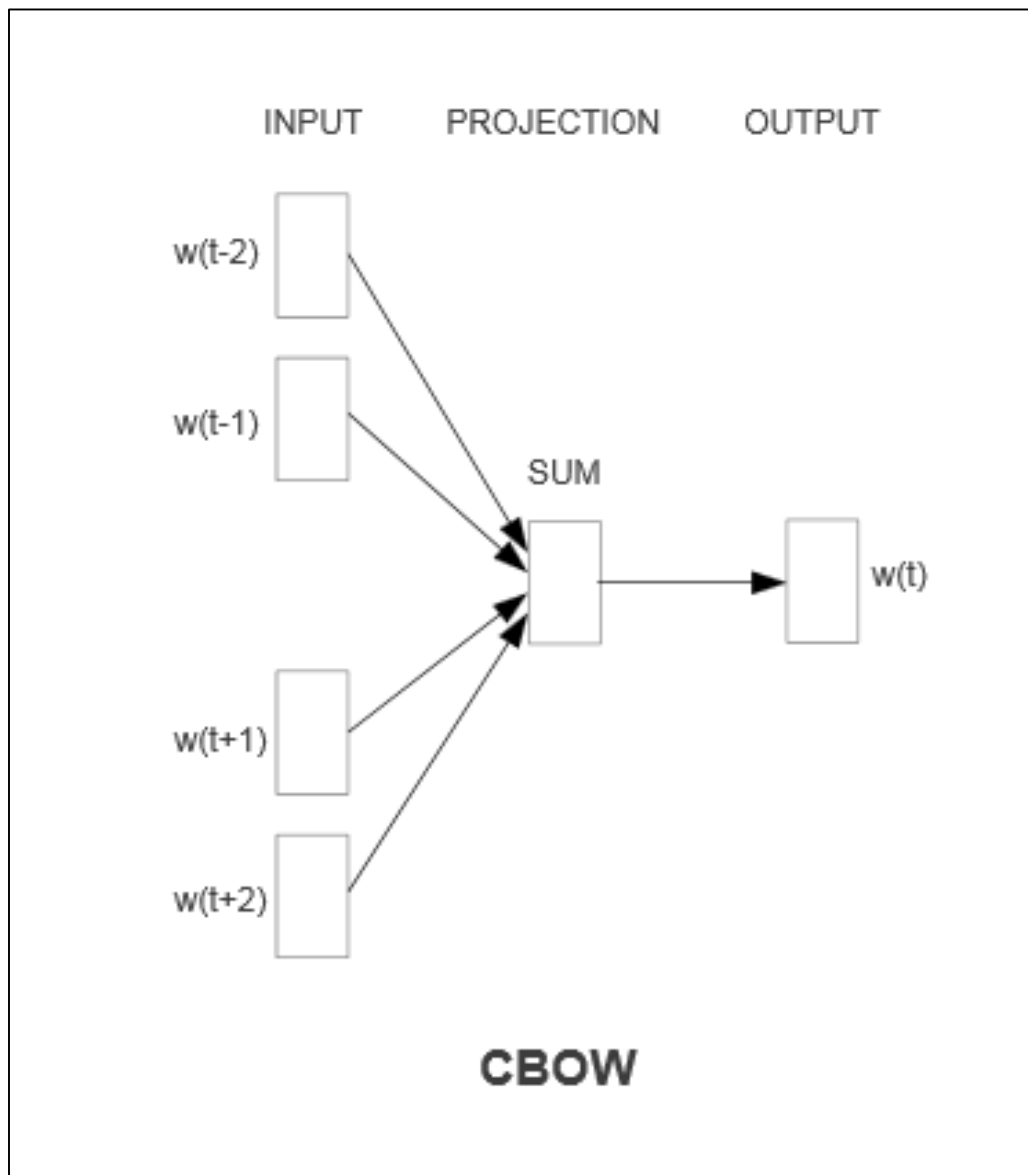


Obtain the Embedding Matrix

- Remember this network is trying to learn a mapping between the context words and the target word.
- Therefore, it will likely find that the word **drive** has a relationship to the word **fast** and the word **cars** has a relationship with the word **fast**. Hence the output of the hidden layer from both of these words will be similar.
- In essence is learning that learning word-occurrence statistics



- The model architecture we see in previous slide can be **expanded** so that we can have more than one context words as input to our network (in other words we increase the window word size).



CBOW (Window Size = 1) Word2Vec

- We have six unique words in the example. The representation for the EM is as follows:
- Notice the word embedding we have for the terms surf big waves all represented relatively **close to each other in 2D space** compared to the terms drive fast and cars.

surf	big	waves	drive	fast	cars
1.1	0.7	1.0	4.2	3.4	5.3
1.2	0.9	0.9	5.2	4.5	3.2

- Also remember it's not that we expect the model we saw in the previous slide to obtain a high level of accuracy but that instead it enables us to build a rich and compressed feature representation.

Using Pre-trained Word Embeddings

- Rather than learning your own word embeddings you can instead **load embedding vectors** from existing and publically available embeddings.
- Quite often you will do this if you have **limited data available** (if you don't have enough data available to learn powerful features then it makes sense to reuse features learned on a different problem).
- This is a type of **transfer learning**.
- There are a range of precomputed databases of word embeddings that you can download and use in a Keras Embedding layer.
- A popular option is called Global Vectors for Word Representation (GloVe, <https://nlp.stanford.edu/projects/glove>)
- Its developers have made available precomputed embeddings for **millions of English tokens, obtained from Wikipedia data**.

Deep Learning



Deep Learning

Lecture: Using Pre-trained Word
Embeddings

Ted Scully

Using Pre-trained Word Embeddings

- Over the next few slides we show how to use **pre-trained word embeddings** for the NewsGroup dataset.
- **Full code [here](#).**
- The NewGroup dataset is a collection of approximately **20,000 messages**, collected from **20 different newsgroups**. One thousand messages from each of the twenty newsgroups were chosen at random and partitioned by newsgroup name.
- The list of newsgroups are depicted on the right.
- You can download the dataset [here](#).
- In the example over the next few slides we will use the GloVe embeddings.

alt.atheism
talk.politics.guns
talk.politics.mideast
talk.politics.misc
talk.religion.misc
soc.religion.christian

comp.sys.ibm.pc.hardware
comp.graphics
comp.os.ms-windows.misc
comp.sys.mac.hardware
comp.windows.x

rec.autos
rec.motorcycles
rec.sport.baseball
rec.sport.hockey

sci.crypt
sci.electronics
sci.space
sci.med

misc.forsale

Using Pre-trained Word Embeddings

- This example assumes you are using Google Colab. On downloading the NewsGroup data I stored it in the following directory in my Google Drive (gdrive/'My Drive'/'Colab Notebooks'/NewsGroupExample).
- To extract the contents of this file run the following command in Colab (as usual you must first mount your Google Drive).

```
!tar -xvf gdrive/'My Drive'/news20.tar.gz
```

Using Pre-trained Word Embeddings

- To download the pre-trained embeddings run the following in Colab (this will extract a number of embedding files. We will use glove.6B.100d.txt).
- It can take a little time to download the full zip file.
- Each line in the embedding contains a word following by the embedded values for that word all separated by spaces.

```
!wget http://nlp.stanford.edu/data/glove.6B.zip  
!unzip glove.6B.zip
```

```
[11] !ls
```

```
20_newsgroup  glove.6B.100d.txt  glove.6B.300d.txt  glove.6B.zip  
gdrive        glove.6B.200d.txt  glove.6B.50d.txt   sample_data
```


Newsgroups Code – Part 1

- In the initial code presented over the next few slides we **read in all news articles** and associate one of 20 class labels with each.
- We **tokenize each text article** so that each string review is stored as a list of integers where each unique integer value represents one specific word.
- Finally we take the data and **pad it** so that each article will have exactly the same length.

```
import os
import sys
import numpy as np
import tensorflow as tf
```

```
MAX_SEQUENCE_LENGTH = 150
MAX_NUM_WORDS = 20000
EMBEDDING_DIM = 100
VALIDATION_SPLIT = 0.2
```

```
embeddings_index = {}
```

```
f = open('glove.6B.100d.txt')
```

```
for line in f:
```

```
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
```

```
f.close()
```

Each line of the embeddings file ('glove.6B.100d.txt') contains a word and **100 float embeddings** values that define that word.

Here we iterate through each line of the file. We store the first element (the word) as a key in the dictionary `embeddings_index`. An array containing the float embedding values is stored as the value.

This will load 400,000 word vectors.

```
texts = [] # list of text samples
labels_index = {} # dictionary mapping label name to numeric id
labels = [] # list of label ids

for name in sorted(os.listdir("./20_newsgroup")):
    path = os.path.join("./20_newsgroup", name)
    if os.path.isdir(path):

        label_id = len(labels_index)
        labels_index[name] = label_id

        # for each post in this newsgroup category
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)

                # dictionary args will specify the encoding used
                args = {} if sys.version_info < (3,) else {'encoding': 'latin-1'}

                with open(fpath, **args) as f:
                    t = f.read()
                    # Remove any empty space at the start of the file
                    i = t.find('\n\n') # skip header
                    if 0 < i:
                        t = t[i:]

                    # Add text from this post to text list
                    texts.append(t)
                labels.append(label_id)
```

```
texts = [] # list of text samples
labels_index = {} # dictionary mapping label name to numeric id
labels = [] # list of label ids

for name in sorted(os.listdir("./20_newsgroup")):
    path = os.path.join("./20_newsgroup", name)
    if os.path.isdir(path):

        label_id = len(labels_index)
        labels_index[name] = label_id

        # for each post in this newsgroup category
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)

                # dictionary args will specify the encoding used
                args = {} if sys.version_info < (3,) else {'encoding': 'latin-1'}

                with open(fpath, **args) as f:
                    t = f.read()
                    # Remove any empty space at the start of the file
                    i = t.find('\n\n') # skip header
                    if 0 < i:
                        t = t[i:]

                # Add text from this post to text list
                texts.append(t)
                labels.append(label_id)
```

It is important to understand that *texts* is a list in which we will store each news posting.

Each news posting is added as a string to *texts*.

The array *labels* will contain a corresponding numerical class label.

We iterate through each directory in the dataset (each corresponding to one news group). Here we store a numerical class label for each directory through which we iterate.

```

texts = [] # list of text samples
labels_index = {} # dictionary mapping label name to numeric id
labels = [] # list of label ids

for name in sorted(os.listdir("./20_newsgroup")):
    path = os.path.join("./20_newsgroup", name)
    if os.path.isdir(path):

        label_id = len(labels_index)
        labels_index[name] = label_id

        # for each post in this newsgroup category
        for fname in sorted(os.listdir(path)):
            if fname.isdigit():
                fpath = os.path.join(path, fname)

                # dictionary args will specify the encoding used
                args = {} if sys.version_info < (3,) else {'encoding': 'latin-1'}

                with open(fpath, **args) as f:
                    t = f.read()
                    # Remove any empty space at the start of the file
                    i = t.find("\n\n") # skip header
                    if 0 < i:
                        t = t[i:]

                    # Add text from this post to text list
                    texts.append(t)
                    labels.append(label_id)

```

For each file in the current directory, open the file, read it's contents as a string and append the string to the list texts.

Keras - Tokenizer

- Keras provides a [text tokenizer](#) class that enables us to easily vectorize a text corpus into a sequence of integer values.
- When you create an instance of the **Tokenizer** you specify the **num_words**. This specifies the maximum number of words to keep, based on word frequency. Only the most common num_words-1 words will be kept.
- The Tokenizer class contains a number of methods and attributes:
- tokenizer.**fit on texts** just specifies the list of texts to train on.
- tokenizer.**text to sequence** returns a list of **integer sequences** so that each list of integers represents one training document.
- **word index** is an attribute of the tokenizer class. It is a **dictionary** that returns a **mapping between words (str) to their index** (int). Only set after fit_on_texts was called.

Below **sequences** is a list of lists. Each list corresponds to a document and the content of the lists are integer values (the encoding between integers and words are specified in the dictionary word_index)

```
# Vectorize the text samples into a 2D integer tensor
tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=MAX_NUM_WORDS)

tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)

print ("Total number of documents: ", len(sequences))

# dictionary containing the word as key and an associated integer id as a value
# mapping between words and associated integer IDs
word_index = tokenizer.word_index
print('Found ',len(word_index),' unique tokens.')
```

```
Total number of documents: 19997
Found 174074 unique tokens.
```

```
data = tf.keras.preprocessing.sequence.pad_sequences(sequences,  
maxlen=MAX_SEQUENCE_LENGTH)
```

```
print('Shape of data tensor:', data.shape)
```

```
# split the data into a training set and a validation set
```

```
labels = np.array(labels)
```

```
indices = np.arange(data.shape[0])
```

```
np.random.shuffle(indices)
```

```
data = data[indices]
```

```
labels = labels[indices]
```

```
num_validation_samples = int(VALIDATION_SPLIT * data.shape[0])
```

```
x_train = data[:-num_validation_samples]
```

```
y_train = labels[:-num_validation_samples]
```

```
x_val = data[-num_validation_samples:]
```

```
y_val = labels[-num_validation_samples:]
```



```
data = tf.keras.preprocessing.sequence.pad_sequences(sequences,  
maxlen=MAX_SEQUENCE_LENGTH)
```

```
print('Shape of data tensor:', data.shape)
```

```
# split the data into a training set and a validation set
```

```
labels = np.array(labels)
```

```
indices = np.arange(data.shape[0])
```

```
np.random.shuffle(indices)
```

```
data = data[indices]
```

```
labels = labels[indices]
```

```
num_validation_samples = int(VALIDATION_SPLIT * data.sh
```

```
x_train = data[:-num_validation_samples]
```

```
y_train = labels[:-num_validation_samples]
```

```
x_val = data[-num_validation_samples:]
```

```
y_val = labels[-num_validation_samples:]
```

As each of the sequences are different lengths we need to ensure all sequences are the same length in order to use in our neural network.

Therefore, we limit each sequence to **150 elements**. Where longer they are truncated, where shorter they are padded with 0 values.

Shape of data tensor: (19997, 150)
Shape of label tensor: (19997,)

For the rest of the code we partition it into training and validation data.

Newsgroups Code – Part 2

- In the code presented over the next few slides deal with integrating the pretrained GLOVE word embedding into our Keras model.
- Remember we had previously read the pretrained embedding into a dictionary (called **embeddings_index**. This dictionary had the word as a key and the embedding vector as an associated value).
- Also we have a dictionary called **word_index** that contained the mapping between each unique word (key) and the integer (value) used to represent the word from our newsgroup dataset.
- The first thing we do in this code is create the embedding matrix. We do this by iterating through each word in **word_index**. We pull out the pretrained embedding for this word and add it to our own embedding matrix.
- We next create our neural network model. The initial part of our model will have an **embedding layer**, which will take in a fixed sized sequence of integer values and will return the document encoded using word embedding vectors.

- In this code we create an **empty embedding matrix**. The dimensions are the num of words by the dimensionality of the embedding matrix itself.
- Next we iterate through **every word in our dictionary** (dictionary contains word – corresponding integer pairs).
- If the word is contained in the downloaded embedding then we add it's encoding (embedding vector) to the embedding matrix here.
- More specifically, we embed the vector for a specific word to the embedding matrix at the row corresponding to the ID of the word.

```
num_words = min(MAX_NUM_WORDS, len(word_index)) + 1

# The embedding matrix dimension will be the num of words in our vocabulary
# by the embedding dimensionality
embedding_matrix = np.zeros((num_words, EMBEDDING_DIM))

# iterate through every word and ID in the word/ int ID dictionary

for word, i in word_index.items():
    if i < num_words:

        # if this word is contained in the downloaded embedding vector
        # then add it to our embedding matrix.
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

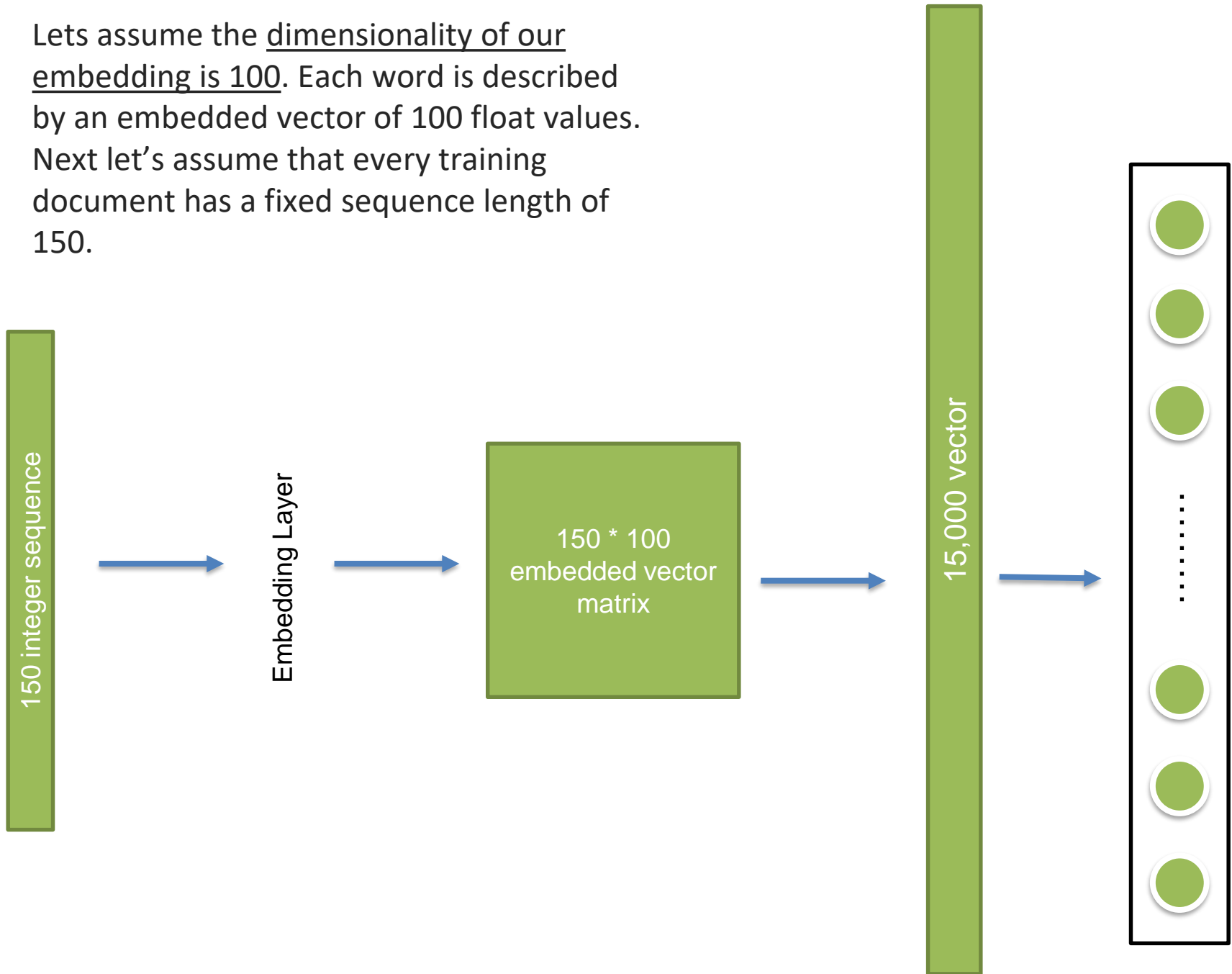
Keras Embedding Layer

- Before we look at the rest of the code let's look at the [embedding layer](#) in Keras.
- The embedding layer turns **positive integers (which represent words) into dense vectors** of a fixed size.
- The best way to think of the embedding layer is as a dictionary that **maps integer indices (which stand for specific words) to dense vectors**. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors.

More specifically you can view the embedding layer as taking as **input** a fixed sized vector (representing **a training document**) where each entry is a **sequence of integers**. Each integer corresponds to a word.

The layer will take the input vector and **returns a 2D floating-point tensor of shape (sequence_length, embedding_dimensionality)**. Each integer (word) in the sequence is translated into its embedded dense representation.

- Lets assume the dimensionality of our embedding is 100. Each word is described by an embedded vector of 100 float values.
- Next let's assume that every training document has a fixed sequence length of 150.



The embedding layer takes as input three main arguments.

- **input_dim**: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
- **output_dim**: int >= 0. Dimension of the dense embedding
- **input_length**: int>0. Length of input sequences, when it is constant

```
model = tf.keras.models.Sequential()

# The first layer is an embedded layer

model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,
input_length=MAX_SEQUENCE_LENGTH))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(20, activation='softmax'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_data=(x_val, y_val))
```

You will notice below that once we have specified the architecture of the network we can then set the weights of the first layer (the embedding layer) to be those that we downloaded earlier and constructed earlier.

```
model = tf.keras.models.Sequential()

# The first layer is an embedded layer
model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,
input_length=MAX_SEQUENCE_LENGTH))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(20, activation='softmax'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32,
validation_data=(x_val, y_val))
```

Epoch 6/10

500/500 [=====] - 7s 14ms/step - loss: 0.1710 - acc: 0.9582 -
val_loss: 2.1746 - val_acc: 0.5406

Epoch 7/10

500/500 [=====] - 7s 14ms/step - loss: 0.1513 - acc: 0.9632 -
val_loss: 2.1178 - val_acc: 0.5626

Epoch 8/10

500/500 [=====] - 7s 14ms/step - loss: 0.1520 - acc: 0.9618 -
val_loss: 2.4870 - val_acc: 0.5534

Epoch 9/10

500/500 [=====] - 7s 14ms/step - loss: 0.1459 - acc: 0.9631 -
val_loss: 2.2724 - val_acc: 0.5701

Epoch 10/10

500/500 [=====] - 7s 14ms/step - loss: 0.1329 - acc: 0.9647 -
val_loss: 2.4437 - val_acc: 0.5806

Deep Learning



Deep Learning

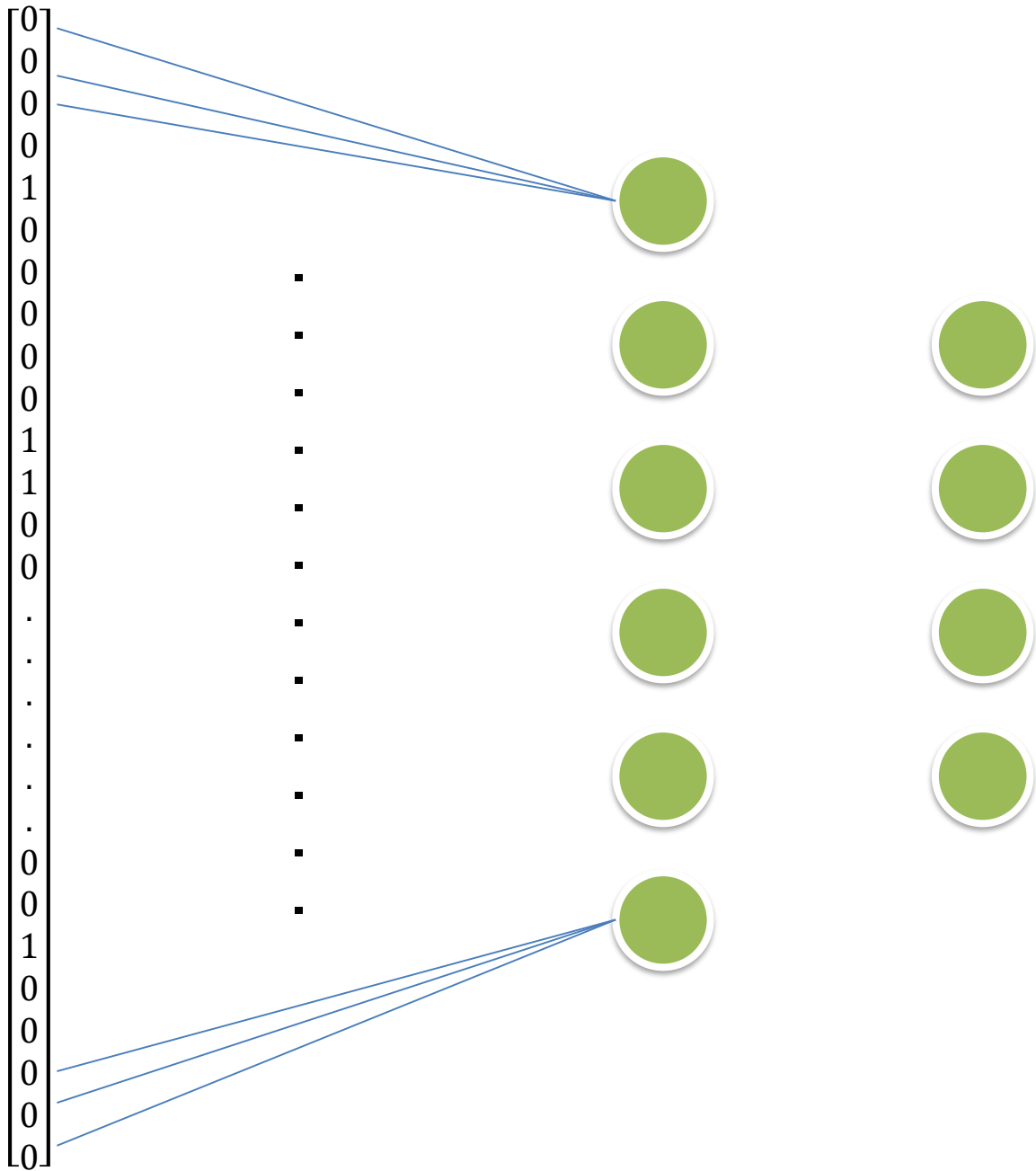
Lecture: Recurrent Networks

Ted Scully

Here we illustrate a single review being fed into a standard densely connected network.

With such networks, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once: turn it into a single data point.

Notice we now lose all information related to the sequence of words. For example, if “not good” was a sequence of words in the review this would just show that the word good and the word not occur somewhere within the review.



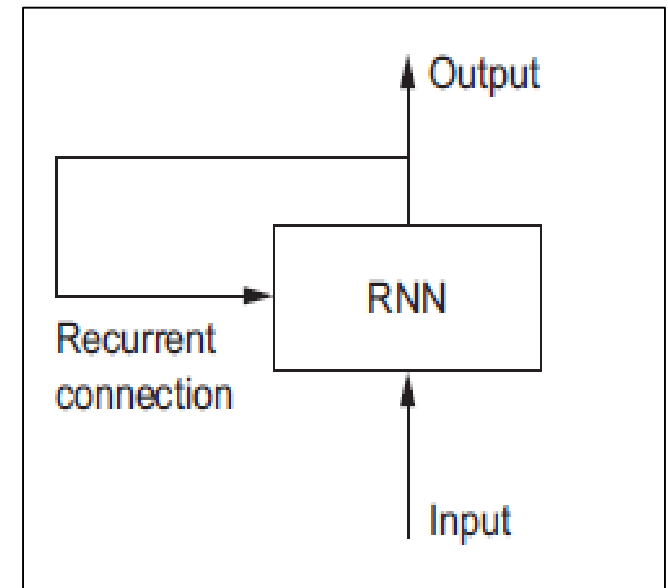
Recurrent Neural Networks

A recurrent neural network (RNN) processes sequences by:

1. **Iterating through the original sequence** of elements (words in a sentence for example) and
2. **Maintaining a state** which provides information about the elements of the sequence it has seen so far.

You can think of an RNN as a type of neural network that has an **internal loop**.

Notice with this network architecture, we no longer provide a sequence as a single input. Instead we provide the first element of the sequence as input, then the next and so on (the network internally loops over sequence elements)



Recurrent Neural Networks

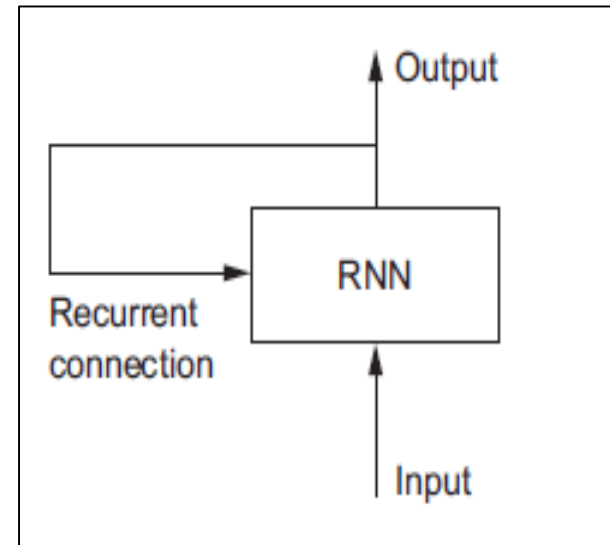
- The following pseudocode gives a good high level perspective of what is happening in an RNN.
- Notice we iterate over the current sequence (again this may be all the words in a single sentence or an entire movie review).
- The input for our RNN on the iteration i is the i th element of the sequence and the output received from the RNN from the previous sequence. .

```
state = 0
```

```
for input_t in input_sequence:
```

```
    output_t = RNN(input_t, state)
```

```
    state = output_t
```



Recurrent Neural Networks

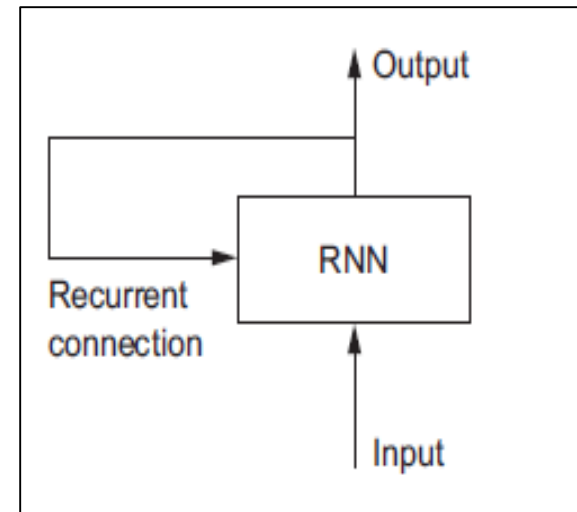
Notice when we expand on what is happening within an RNN it is very similar to what we are used to when we take the input values multiply them by weights, add a bias and push through an activation function.

```
state = 0
```

```
for input_t in input_sequence:
```

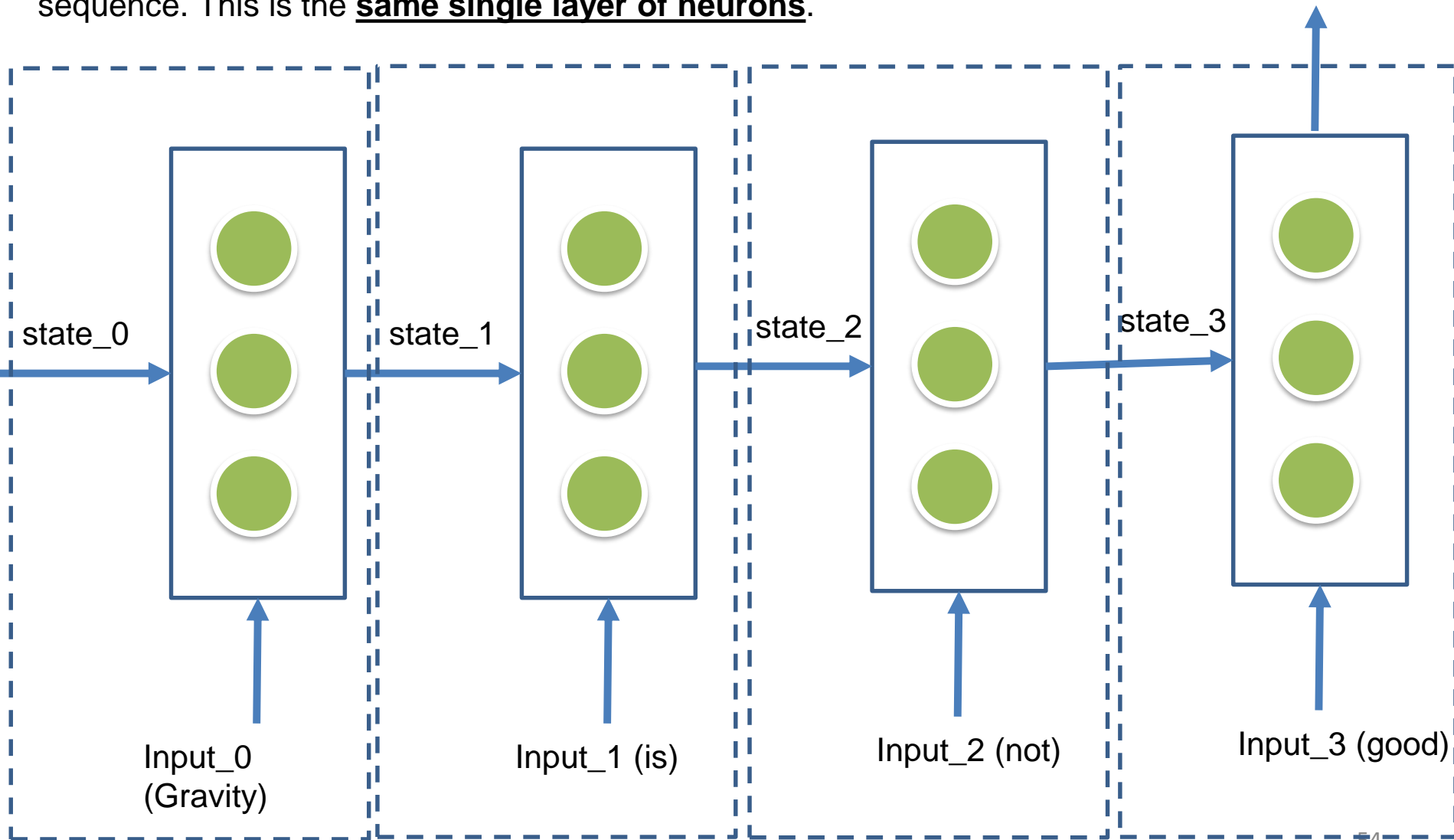
```
    state = tanh( dot(W, input_t) + dot(U, state) )
```

```
    output_t = tanh(dot (Z, state))
```



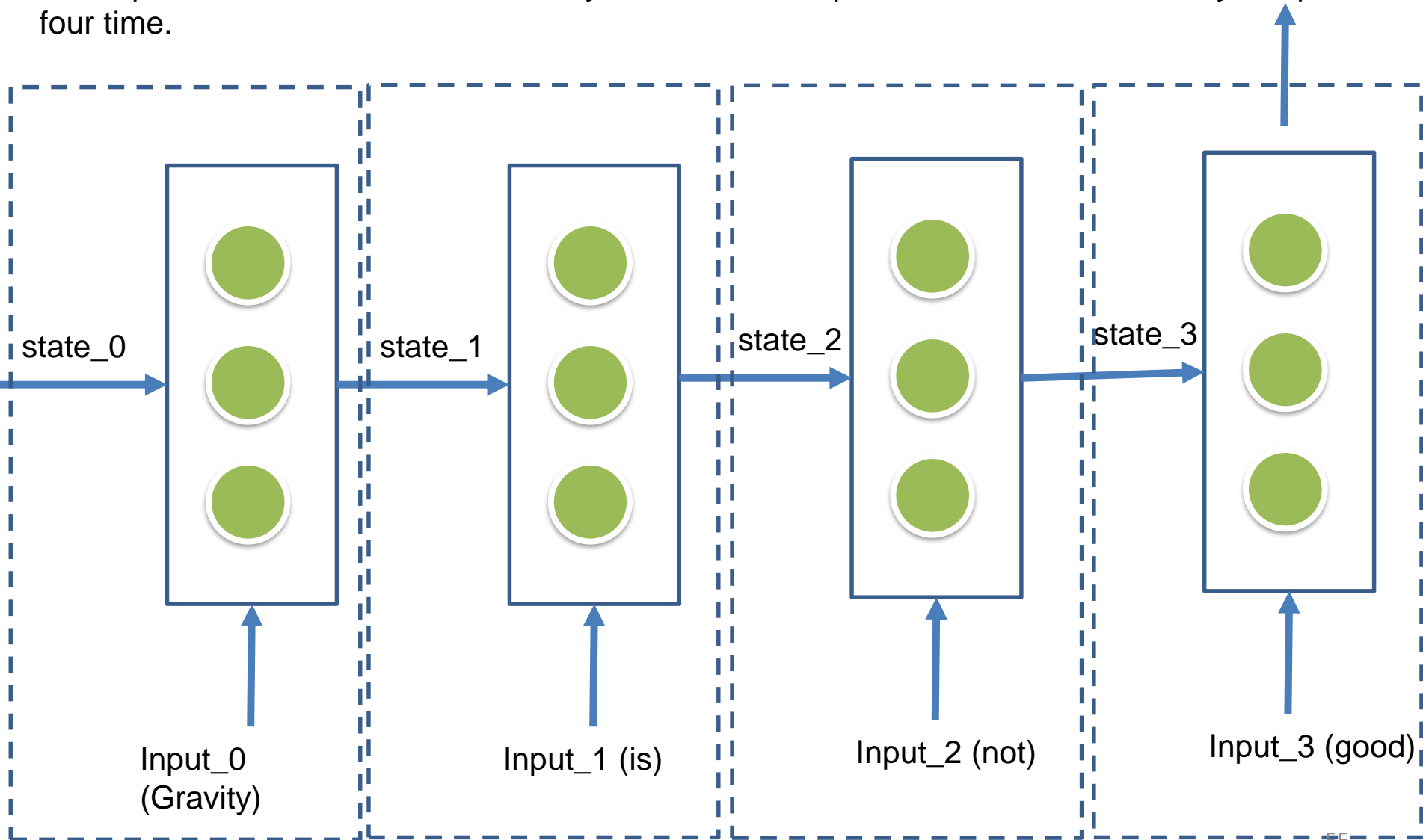
In this example let's assume we have a small sequence X **[Gravity is not good]**.

It is important to understand the image below depicts the **four iterations** of the loop over the sequence. This is the **same single layer of neurons**.

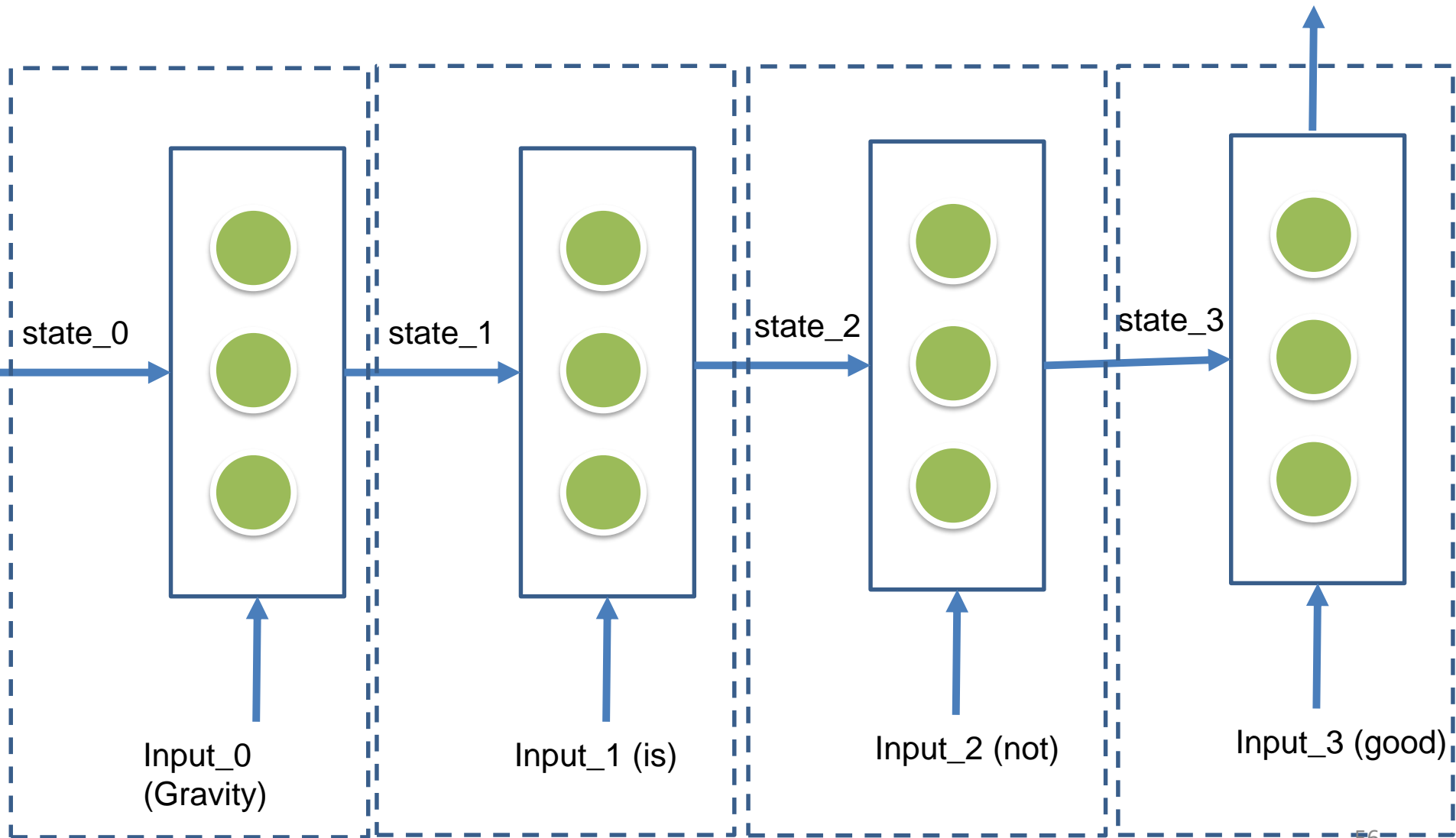


Clearly a recurrent unit allows the **network to model the sequence of items as opposed to just the presence or absence of items within the sequence.**

It is important to understand that the layer of neurons depicted below is the same layer repeated four time.



An RNN learns by using **back propagation**. Here the predicted output of the network is Y' . We compute the usual cross entropy error and propagate back through the network in the normal way to update all relevant weights.



Different RNN Architectures

The RNN we have looked at so far has the capability of taking as input a **sequence** and predicting a **single output**.

This type of model might be appropriate for sentiment analysis of documents or tweets. The model architecture is what is referred to as a **many to one** model architecture. That is the RNN receives multiple inputs as part of a sequence but will only have a single predictive output.

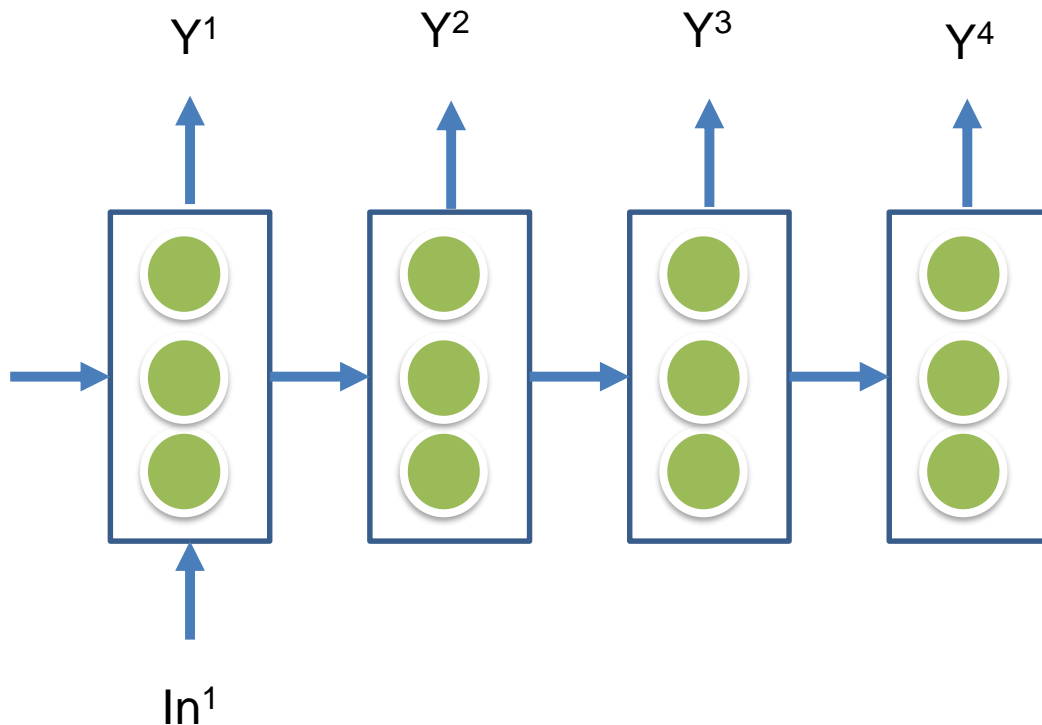
However, we may have other types of sequence problem that cannot be accommodated by this structure.

For example:

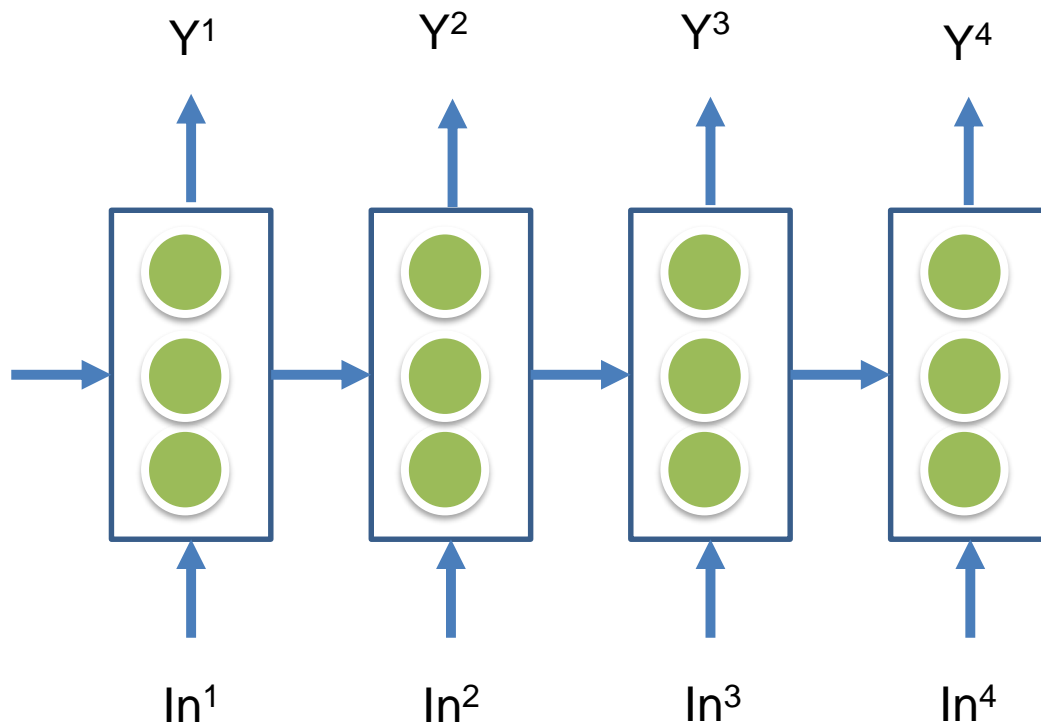
1. If you were performing music or language generation then you may have one input and many outputs (**one to many**).
2. Some problems such as machine translation and name entity recognition problems are many to many problems (**many to many**).

There are recurrent network architectures to accommodate each of these types of problems.

In the example below we illustrate the architecture for a **one to many RNN**. In the case of **language generation** the input In^1 might be a single word and the output a sequence of words. In the case of **music generation** it might be an integer value indicating the genre and the output might be a sequence of notes.



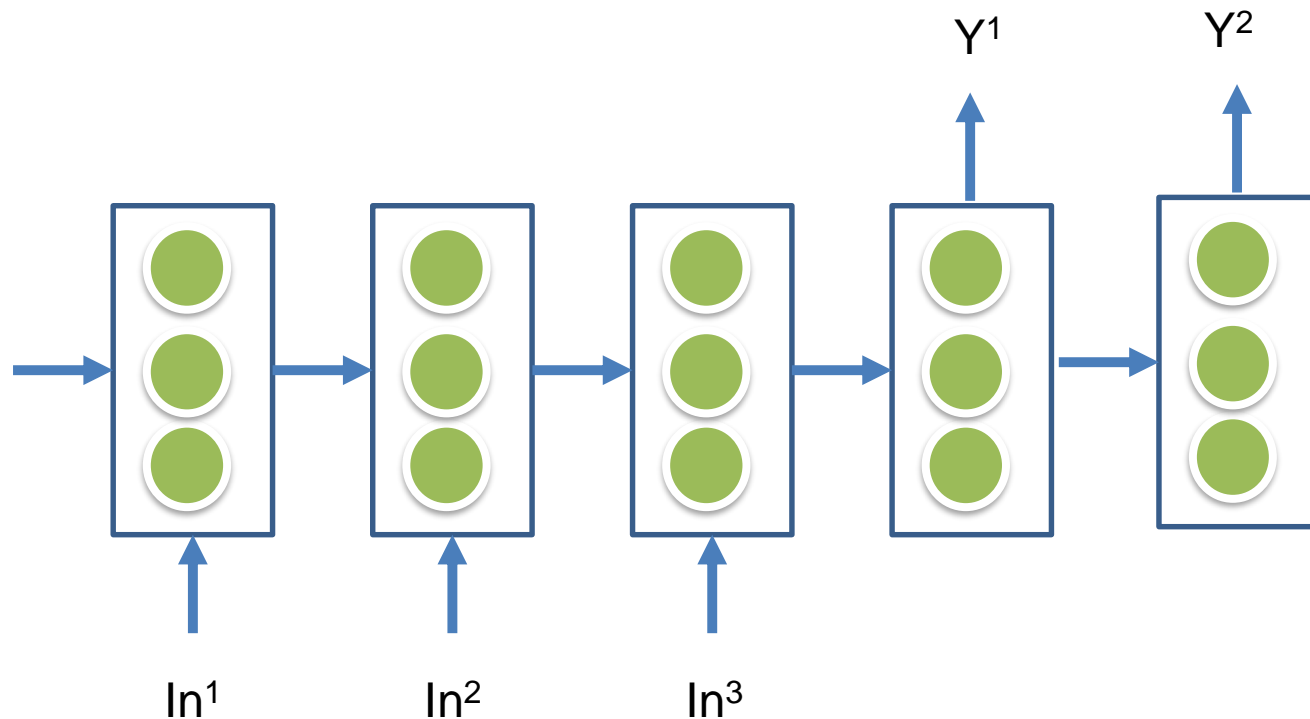
The architecture below illustrates a **many to many** RNN architecture. This occurs when for every input sequence we have an associated output prediction. An example of this would be a problem such as **name entity recognition**. There is also another variants of the many to many architecture depicted on the next slide.



RNN Architectures

- There is also another **variant of the many to many architecture** where we have a sequence as input but the predicted output sequence does not match the number of values in the input sequence exactly.
- For example notice if performing machine translation between English and Spanish the input might be a sentence “how are you” and the output is “cómo estás”. Notice the input length doesn’t match the output length.
- In such cases you employ a slightly different RNN architecture.
- It is split into two phases, the first is referred to as the **encoder** where you take in the sequence of inputs.
- After the sequence of inputs has been accepted then you predict the output in the second part with is the called **decoder**.

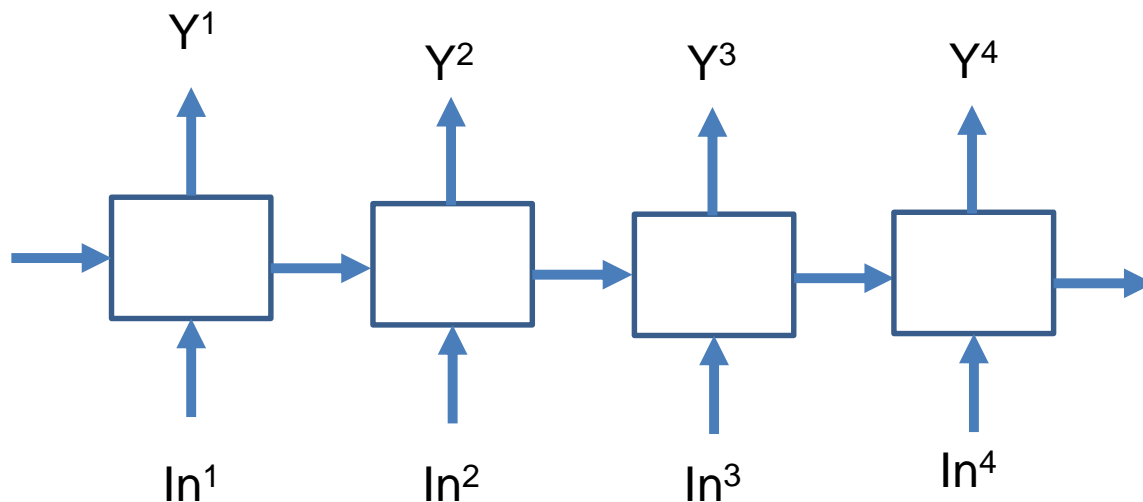
This variant of the many to many architecture is depicted below. Notice that only after we read in all the contents do we then begin outputting the translation.



Stacking Recurrent Units (Deep Recurrent Networks)

While the diagram below illustrates a many to many architecture we can make these networks **deeper by simply stacking them on top of each other**.

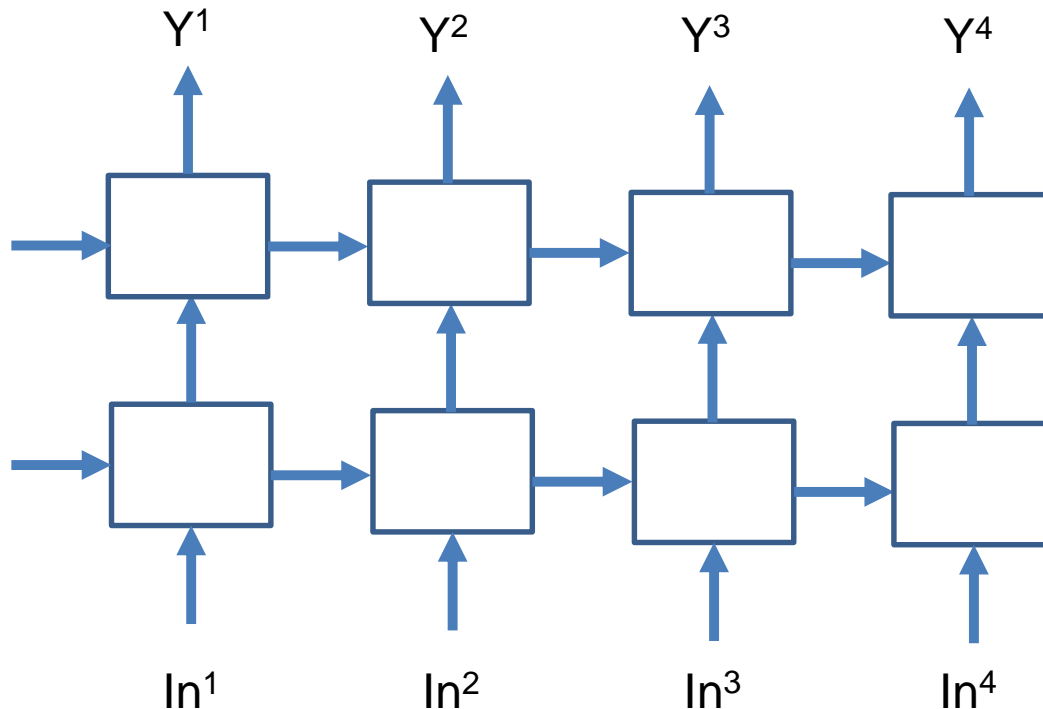
In practice the **level of depth is limited**. In other words there may typically be one or two additional layers stacked on top of an existing layer. This is because of the temporal component of recurrent layers these layers can already get very big.



Stacking Recurrent Units (Deep Recurrent Networks)

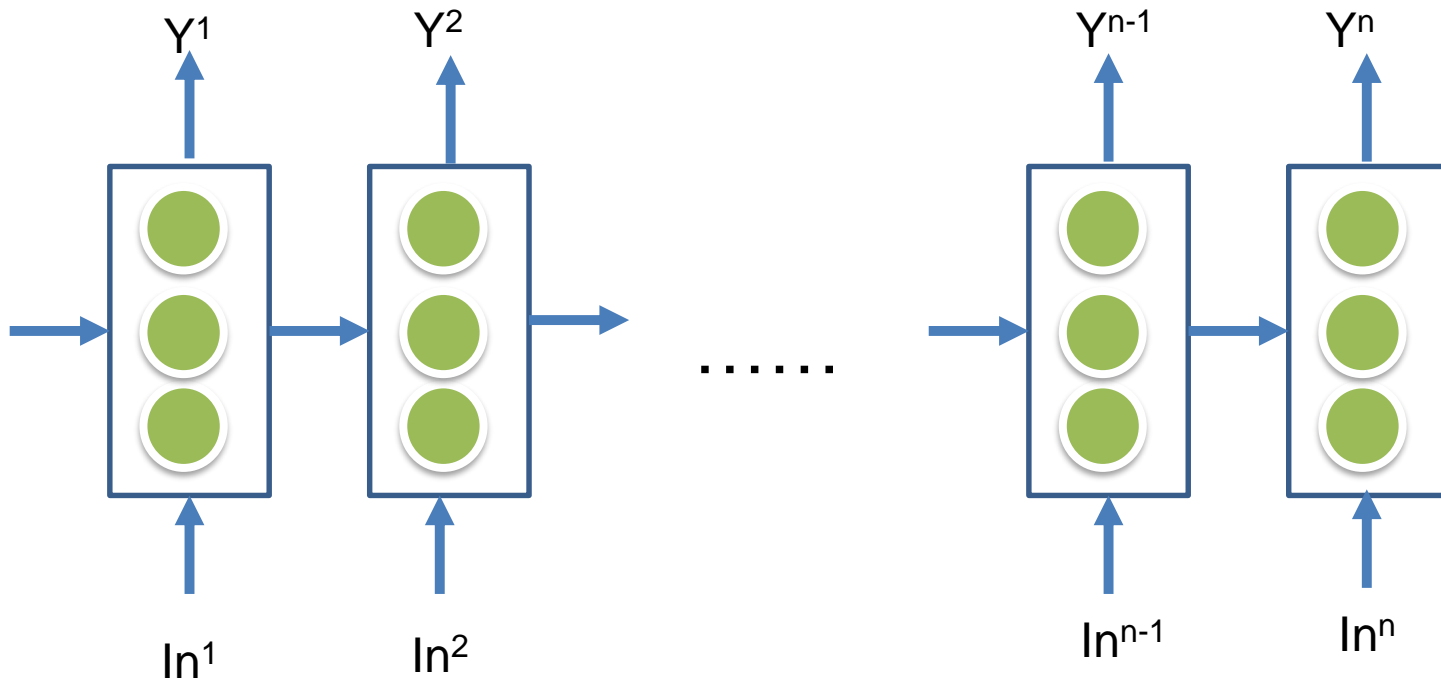
While the diagram below illustrates a many to many architecture we can make these networks **deeper by simply stacking them on top of each other**.

In practice the **level of depth is limited**. In other words there may typically be one or two additional layers stacked on top of an existing layer. This is because of the temporal component of recurrent layers these layers can already get very big.



RNN Limitation – Vanishing Gradient Problem

- Vanishing gradient is a issue that occurs with deep neural networks **where the change in the gradient of the learnable parameters becomes smaller and smaller** as you move back through the network.
- As we know we propagate the error backwards from the output to input layer.
- As we move back through the network the gradient often get smaller, which means the weights in the earlier layers may never be updated sufficiently.
- RNNs are quite susceptible to this problem. The information from an entry late in the sequence can fail to propagate backwards.



RNN Limitation – Vanishing Gradient Problem

- As already mentioned one of the powerful components of RNNs is that they can **connect previous information in the sequence to the task** we are performing.
- For example if we consider a **language model**, which takes in a sequence of words and attempts to predict the next word in the sequence for the following two sentences.
- I **cycle** my **bicycle**
- When I get home after my **cycle** I go to the shed and lock up my **bicycle**
- If we provide the sequence underlined then the next word in the sequence is bicycle. Clearly the words that have gone before in the sequence (in this case 'cycle') will help identify the correct word.
- However, due to the vanishing gradient problem the **further apart the target** we want to predict and the **relevant information the less likely the RNN is to connect the previous relevant information with the target**.
- In other words in the second example where we have a larger gap between the target and relevant words, RNNs become less able to learn to connect the information.

Deep Learning



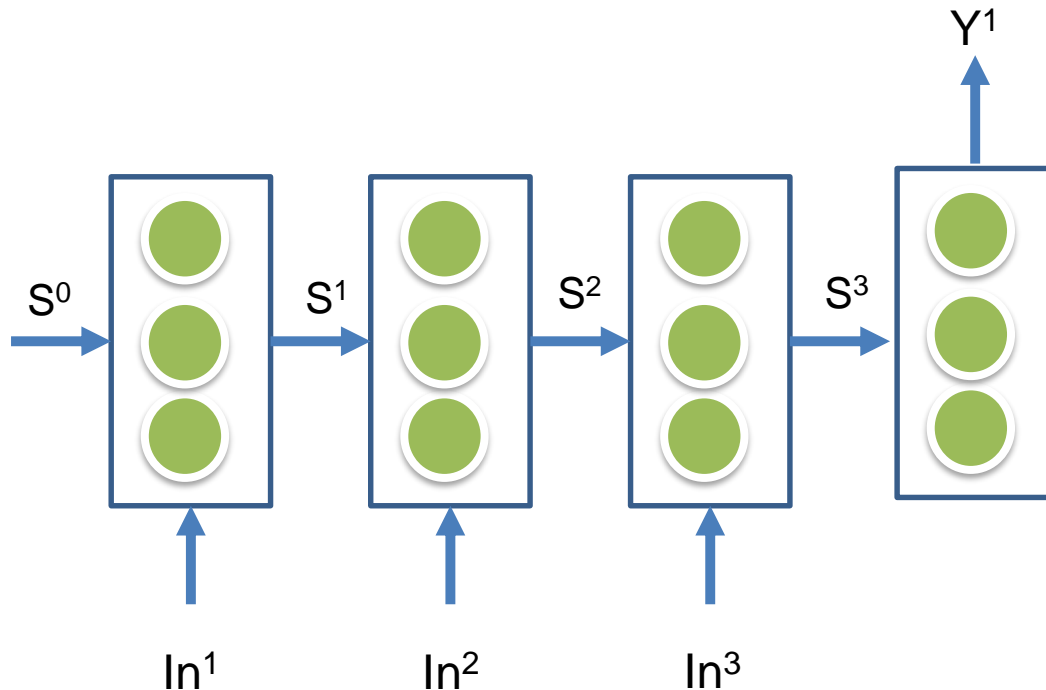
Deep Learning

Lecture: LSTM Networks

Ted Scully

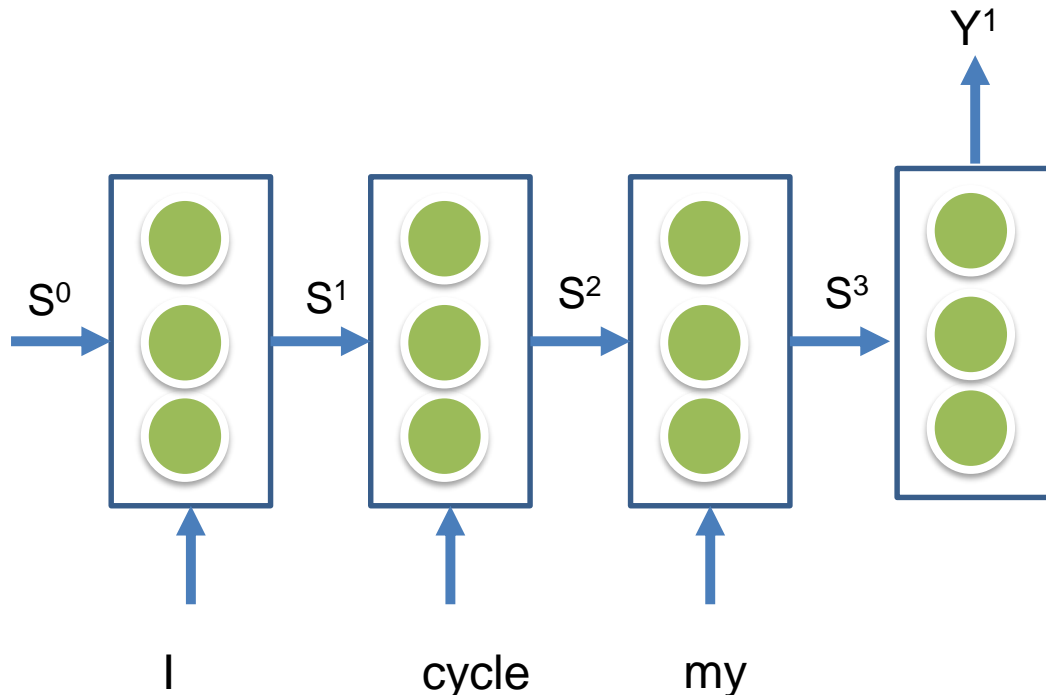
Recurrent Neural Networks

- As already mentioned A recurrent neural network (RNN) processes sequences by:
 1. **Iterating** through the original sequence of elements and
 2. Maintaining a **state** which provides information about the elements of the sequence it has seen so far.



Recurrent Neural Networks

- As already mentioned A recurrent neural network (RNN) processes sequences by:
 1. **Iterating** through the original sequence of elements and
 2. Maintaining a state which provides information about the elements of the sequence it has seen so far.
- One of the powerful components of RNNs is that they can **connect previous information** in the sequence to the task we are performing.
- Assume we have a sentence completion task: I cycle my bicycle

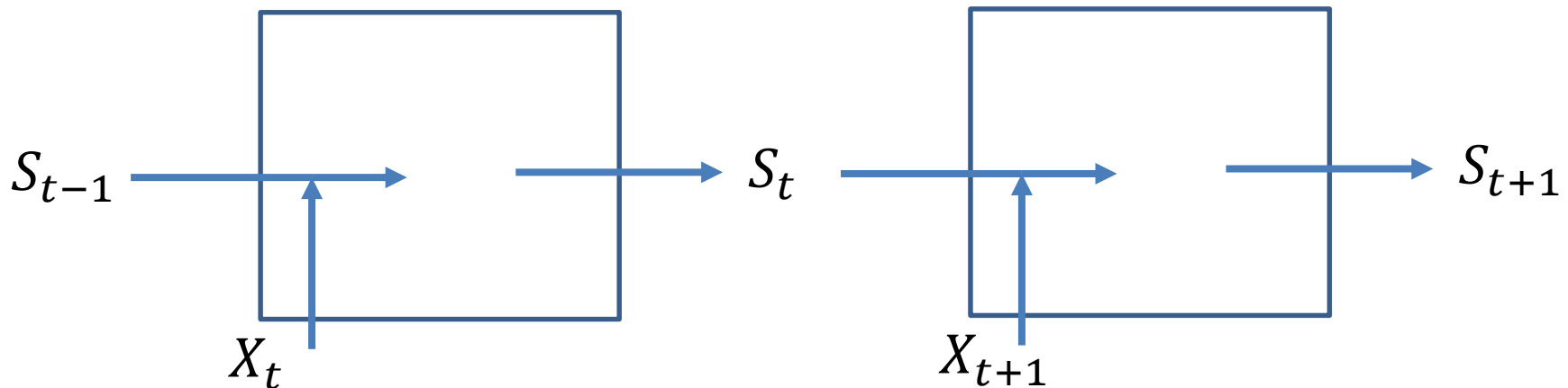


Recurrent Neural Networks

- As already mentioned A recurrent neural network (RNN) processes sequences by:
 1. **Iterating** through the original sequence of elements and
 2. Maintaining a state which provides information about the elements of the sequence it has seen so far.
- One of the powerful components of RNNs is that they can **connect previous information** in the sequence to the task we are performing.
- Assume we have a sentence completion task: I **cycle** my **bicycle**
- The important context word within the sequence here is the word cycle
- Unfortunately standard RNNs perform **poorly** when the important context word(s) are far away from the target in the inputted sequence.
- When I get home after my **cycle** I go the shed and lock up my **bicycle**

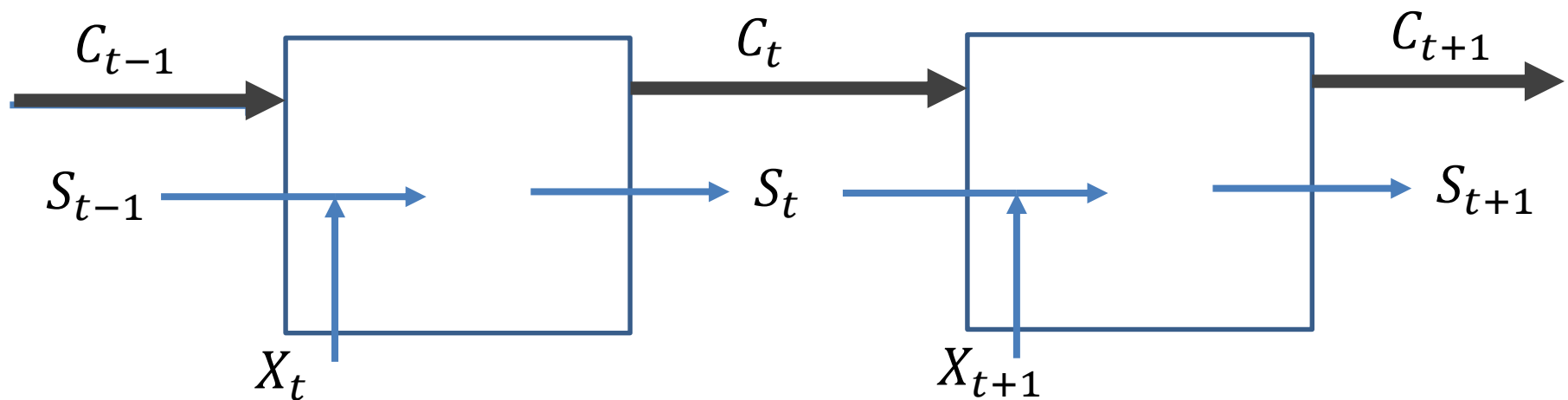
LSTM (Long Short Term Memory) Networks

- Long short term memory (LSTM) networks are a type of recurrent neural network.
- LSTM networks allow us to **model longer term dependencies between elements in a sequence.**
- We represent them in a similar repetitive structure (as we depicted in RNNs) but unlike RNNs which just have a **single layer of neurons**, LSTM networks contain four **separate layers** that each play a very specific role.



The core concept behind the LSTMs is what is referred to as the 'cell state'. We will refer to the cell state (which is a vector of values) after sequence element t as C_t **This is the connection that runs along the top of the diagram.**

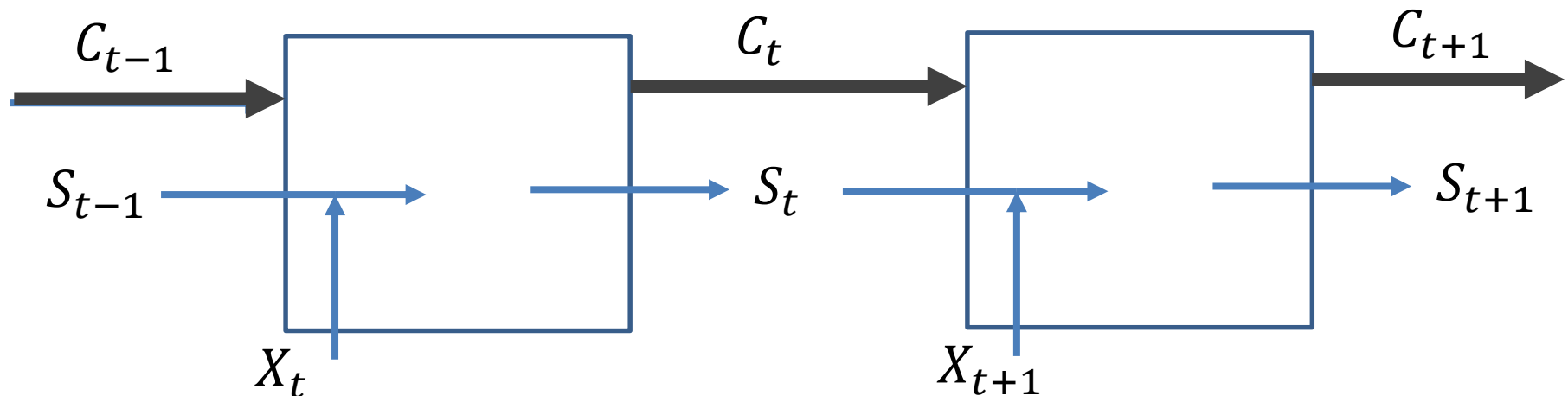
Notice below we just represent one iteration but clearly there will be as many as there are sequence element. The cell state runs the full length of the chain of sequence elements , with only limited interaction with other components. This makes it very easy for information to flow along this link.



Notice the cell state is different from what we looked at previously in recurrent networks where all information has to pass through a neural network layer and undergo the normal transformation process (weight multiplication and activation)

Information is free to run along the cell body without directly passing through a layer of neurons.


An interesting aspect about LSTMS is that they can also be set up to be **bidirectional**.

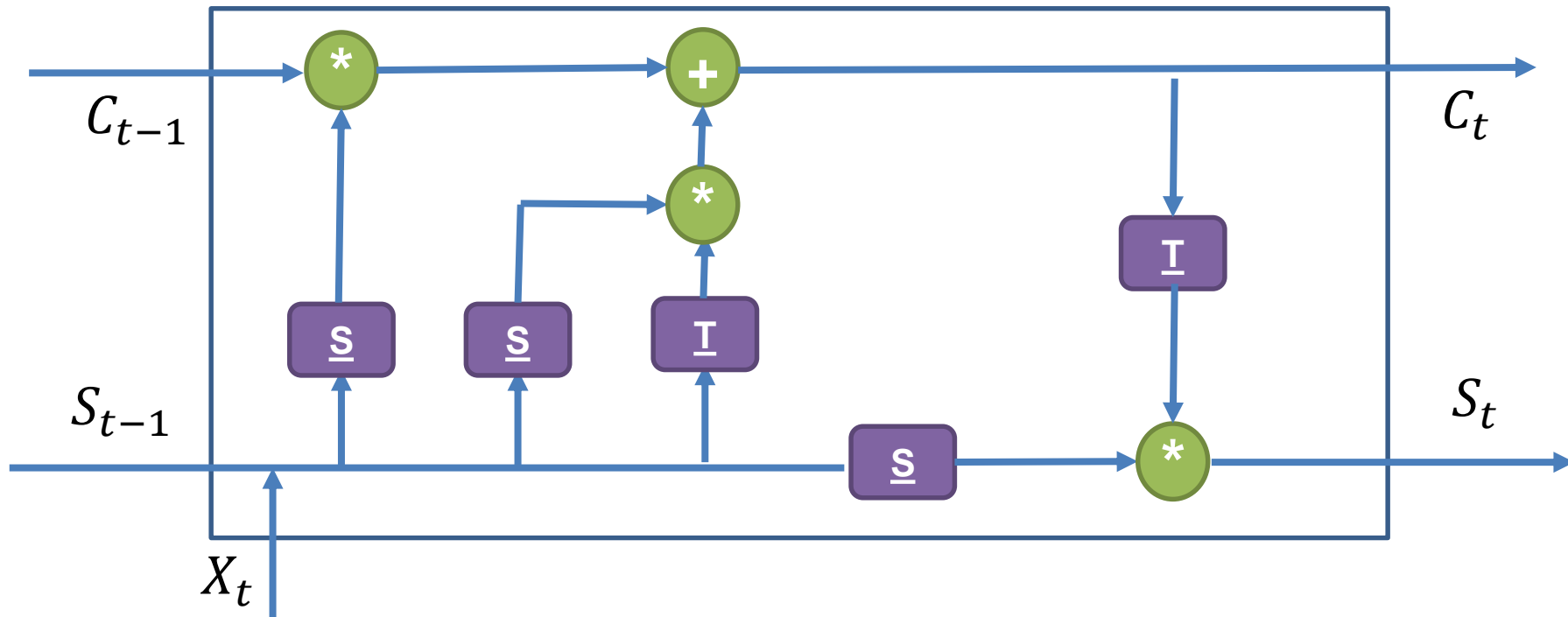


 = element-wise multiplication

 = element-wise addition

 = sigmoid layer of neurons

 = tanh layer of neurons



LSTM (Long Short Term Memory) Networks

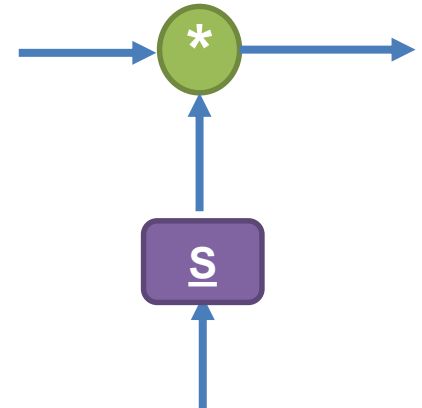
A core concept in LSTM is the idea of a gate.

The interaction between the LSTM and the cell state, is controlled through gates.

A gate allow the LSTM to alter the cell state C_t .

Each gate consists of a **sigmoid neural network layer** followed by an **element-wise multiplication**. Remember the output of a sigmoid layer of neurons is going to be a vector of values between 0 and 1. These values control the rate of change we wish to apply to current values flowing along the cell body (C_t).

A value of 1 will leave a vector element unchanged. A value of 0.5 will reduce it's value by half.



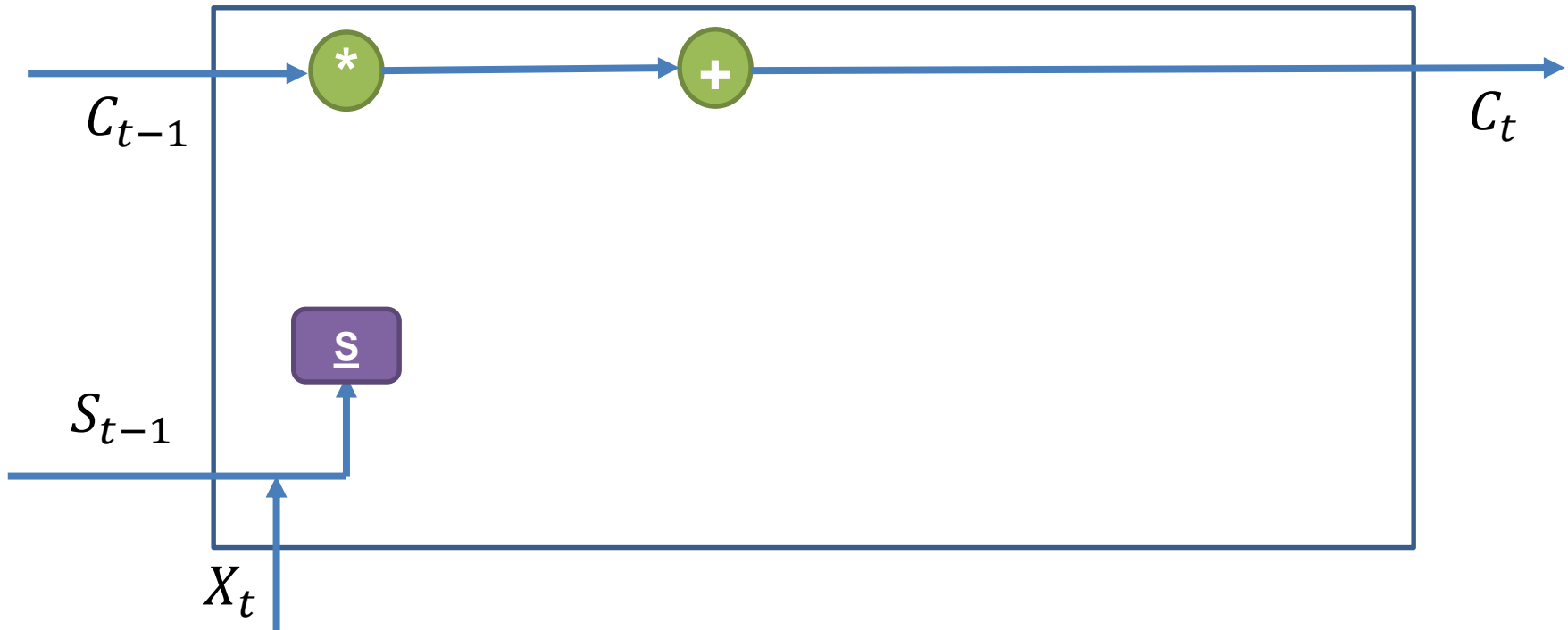
Gate 1 – Forget

The first gate in the LSTM is what is referred to as the ‘forget gate’.

It takes the output from the previous sequence, S_{t-1} and the current t^{th} item from the sequence, X_t and outputs a vector of numbers between 0 and 1.

The length of the vector is the same length as the cell state C_{t-1}

We refer to the output as f_t $f_t = \text{Sigmoid}(W_f \cdot [S_{t-1}, x_t] + b_f)$

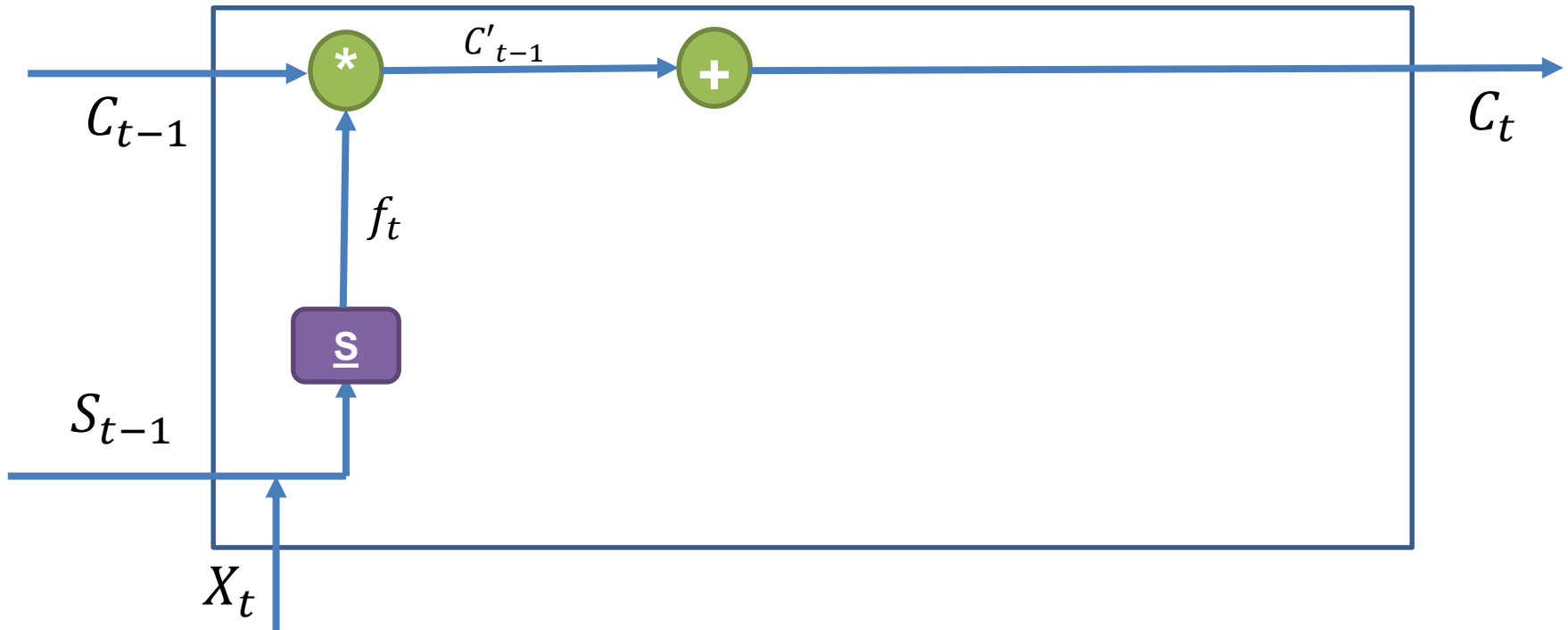


Gate 1 - Forget

Next we take the output f_t and perform an element-wise multiplication between this value and C_{t-1} .

We refer to the output as C'_{t-1}

$$C'_{t-1} = f_t * C_{t-1}$$

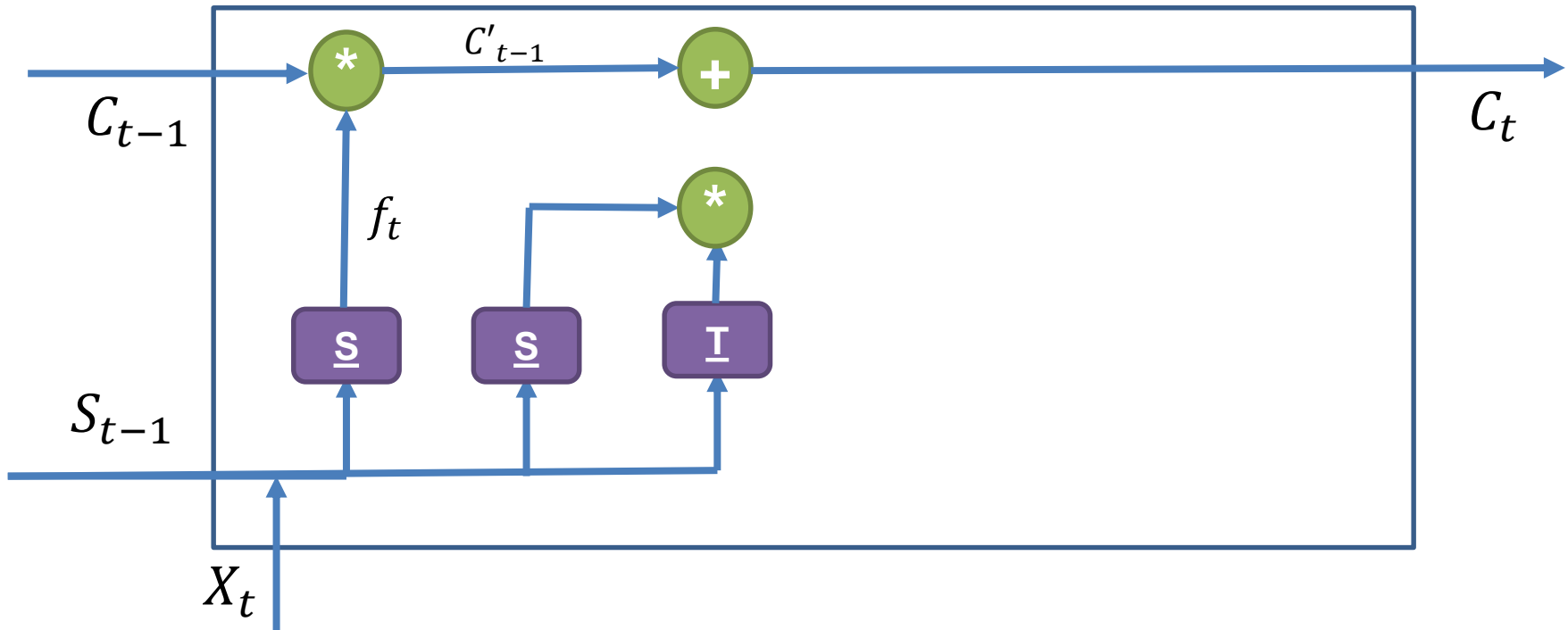


Gate 2 - Update

This step determines the **new information that will be added** to the cell state based on the sequence entry for this period X_t and the old state value S_{t-1} .

The **Tanh layer** of neurons decides the values for updating the current cell value.

These values are **modified** again by another **sigmoid gate** called the update gate.

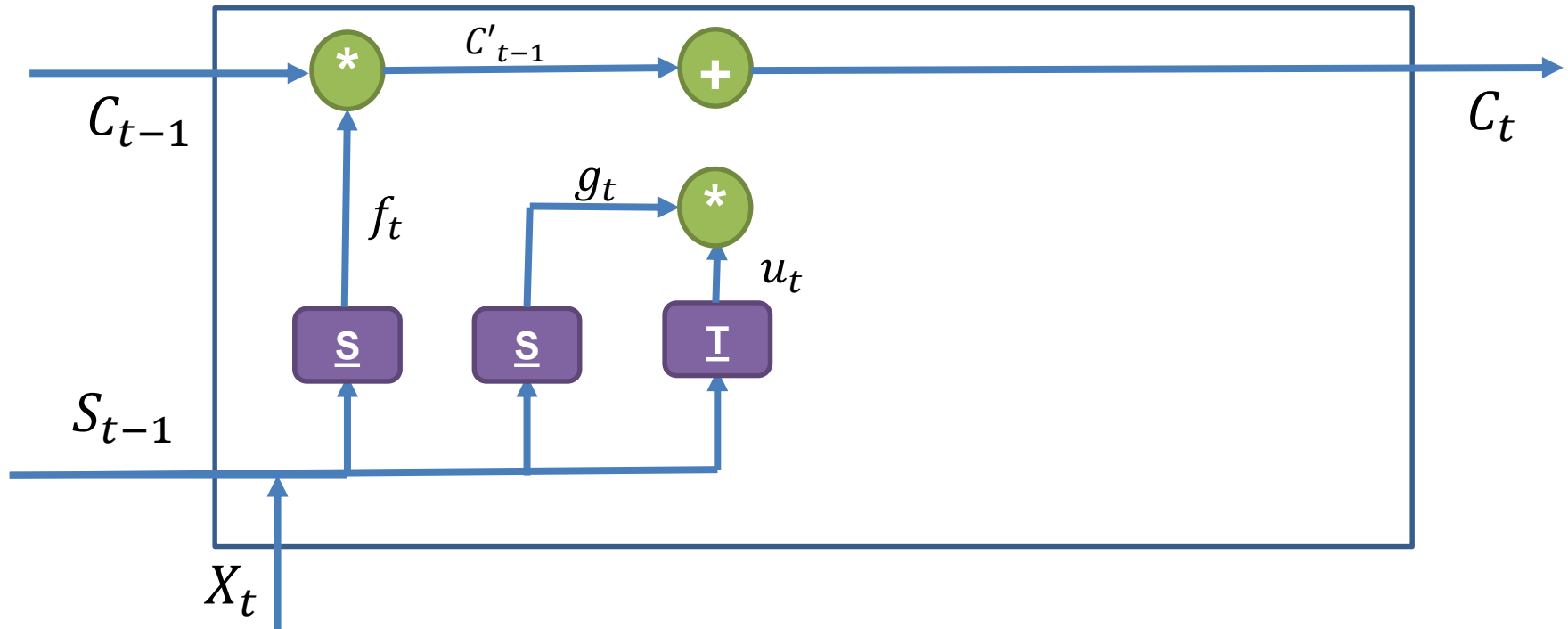


Gate 2 - Update

$$g_t = \text{Sigmoid}(W_g \cdot [S_{t-1}, x_t] + b_g)$$

$$u_t = \text{Tanh}(W_u \cdot [S_{t-1}, x_t] + b_u)$$

$$u_t = (u_t * g_t)$$

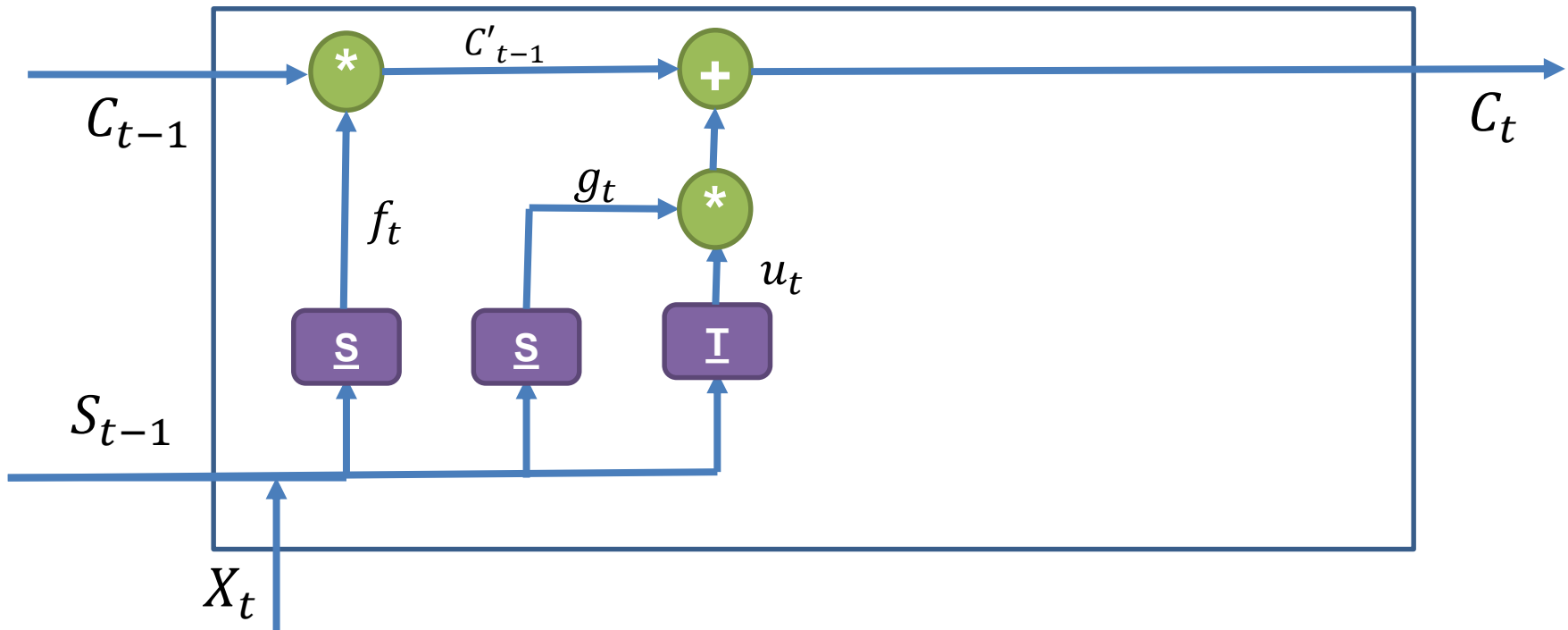


Gate 2 - Update

Next we can update the cell body value with the new information flowing from the update gate.

This new value for the cell body (C_t) flows forward to the next iteration of the recurrent unit.

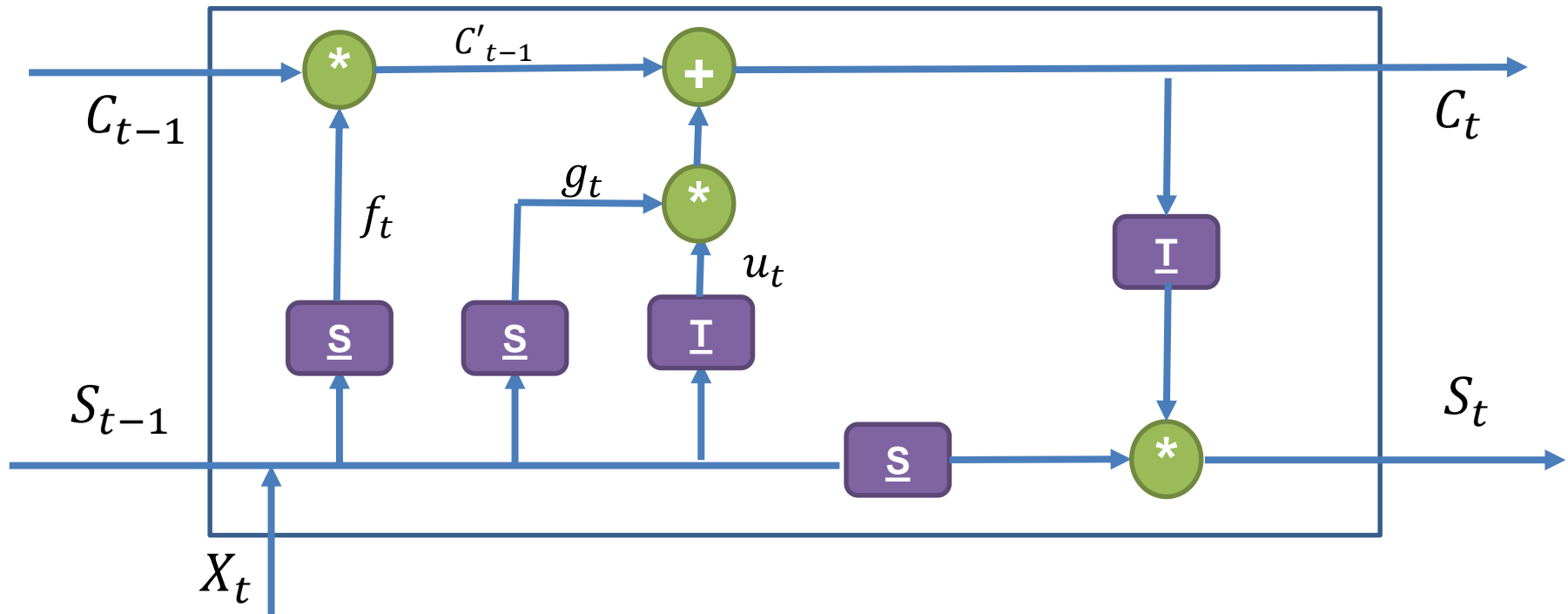
$$C_t = C_{t-1} + u_t$$



Gate 3 – Output

Each iteration of a LSTM unit outputs an updated value for the cell body C_t and also outputs an updated state value S_t , which can then influence the next iteration of the unit.

You will notice that the new state value is calculated by taking the updated cell body value C_t and passing it through a Tanh layer of neurons. As with each of the other outputs in the LSTM this one is also gated by a sigmoid layer and a multiplication operation.



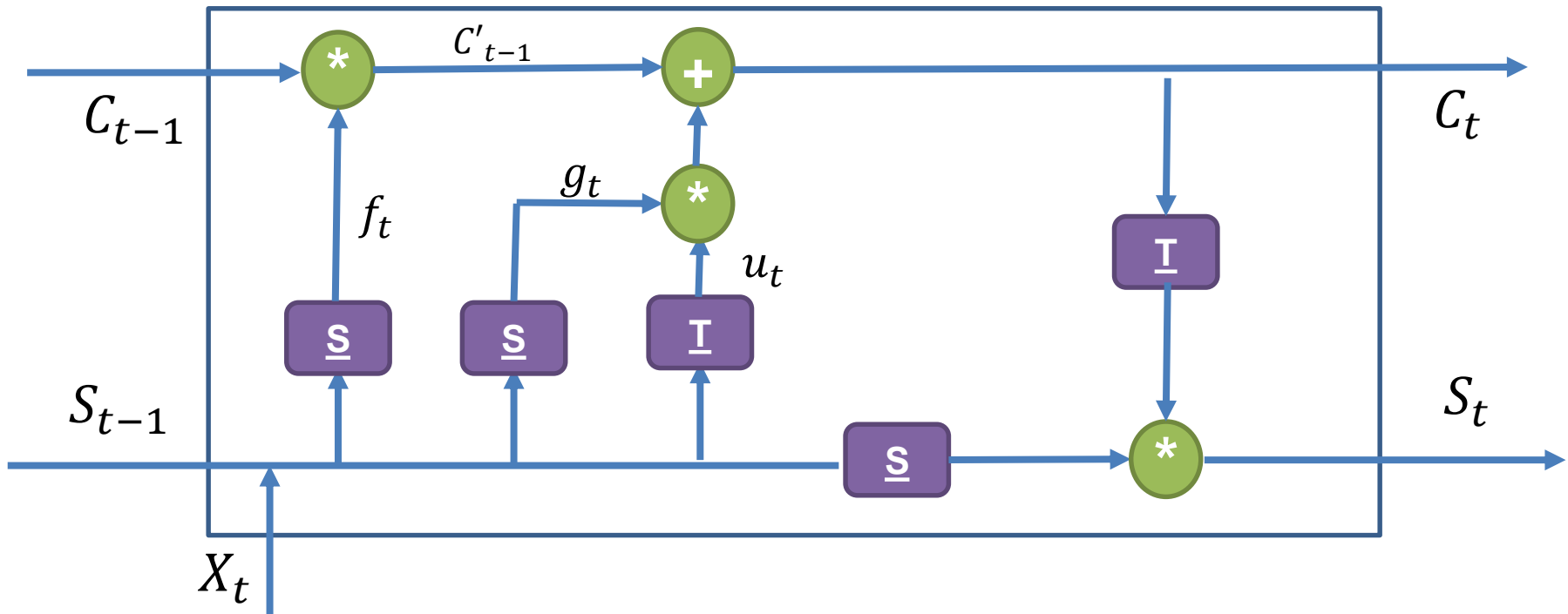
Gate 3 – Output

More formally the calculation of the final output state S_t can be calculated as follows:

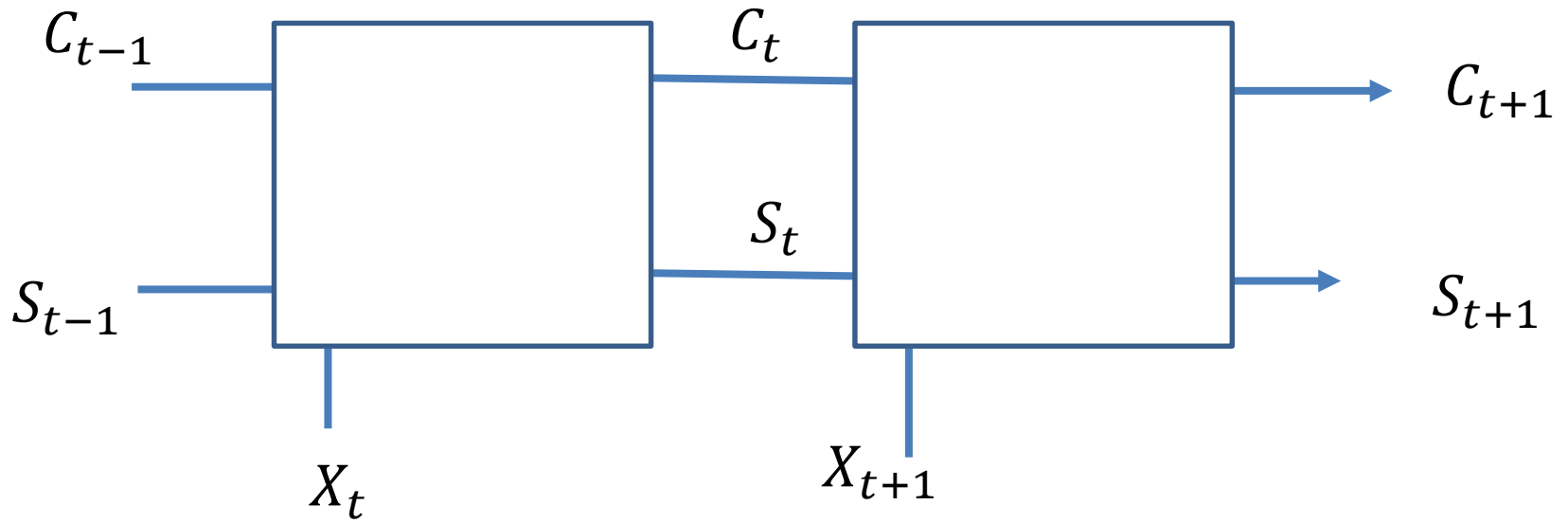
$$h_t = \text{Tanh}(W_h \cdot C_t + b_h)$$

$$k_t = \text{Sigmoid}(W_k \cdot [S_{t-1}, x_t] + b_k)$$

$$S_t = (h_t * k_t)$$



LSTM



Deep Learning



Deep Learning

Lecture: Building an LSTM network with Keras

Ted Scully

Using LSTM in Keras

Typically LSTM units will outperform basic recurrent units. Therefore, we will work through a number of examples of building [tf.keras.layers.LSTM](https://keras.io/layers/recurrent/#lstm-layer).

The core components of the LSTM unit have the following arguments.

- **units**: Positive integer, dimensionality of the output space. Essentially this specifies the **number of neurons** in each of the layers present in the LSTM. Clearly the number of neurons is going to influence the dimensionality of the output.
- **input_shape**: An LSTM expects a 3D data structure as input (**batch size, number of time steps, number of features**). However, as normal with Keras, we don't typically specify the batch size therefore, we just specify the input shape as (**number of times steps * number of features**) [more on this in the next slide]. Remember you only need to specify the input shape if it is the first layer in a network.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence. [more on this in next few slides]

Using LSTM in Keras

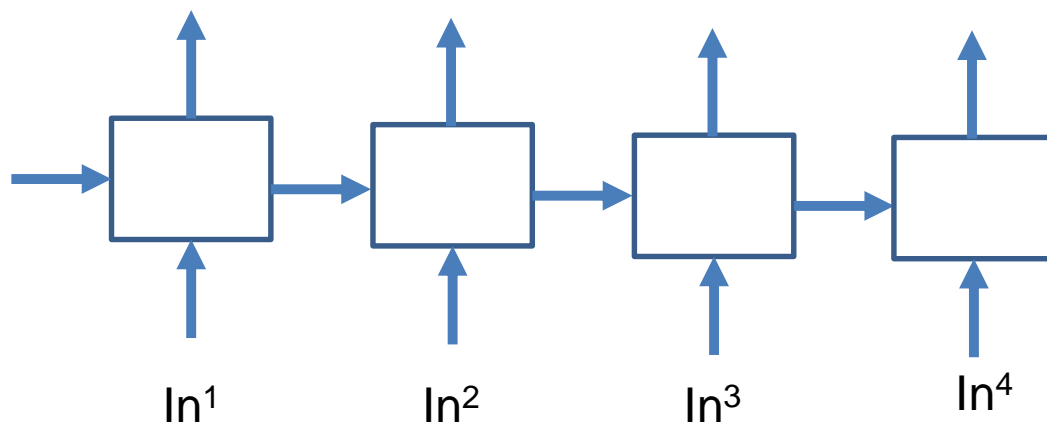
As mentioned on the previous slide the input shape we specify is the (**number of times steps** * **number of features**). To understand the difference between these two consider the example below.

We want to build a model that will predict the temperature for the next hour using the temperature reading from the **last four hours**.

In this example the **number of time steps is 4** (you can think of this as the number of iterations of your recurrent unit). In^1 might input the temperature for hour1, In^2 the temperature for hour 2 and so on.

If this was the case then the **number of features would be 1** because each vector In^t below would just be a vector with one single value (the temperature for a specific hour t).

Therefore, the input shape will be (4*1) (number of times steps * number of features).



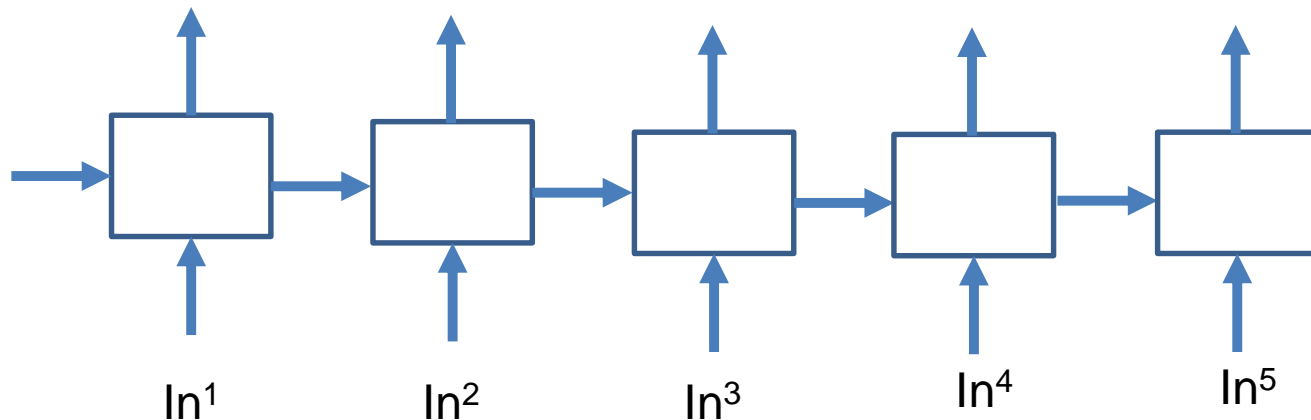
Using LSTM in Keras

Now let's assume we want to predict temperature for the next hour using the last 5 hourly temperature values as well as the **irradiance value for the last five hours**.

The number of times steps is 5. To predict the temperature for the next hour we are taking as input a sequence that stretches of five hours.

Now the number of features is 2. Each vector In^t would input two values the temperature and the irradiance value for the hour t .

Therefore, the input shape will be $(5*2)$ (number of times steps * number of features).

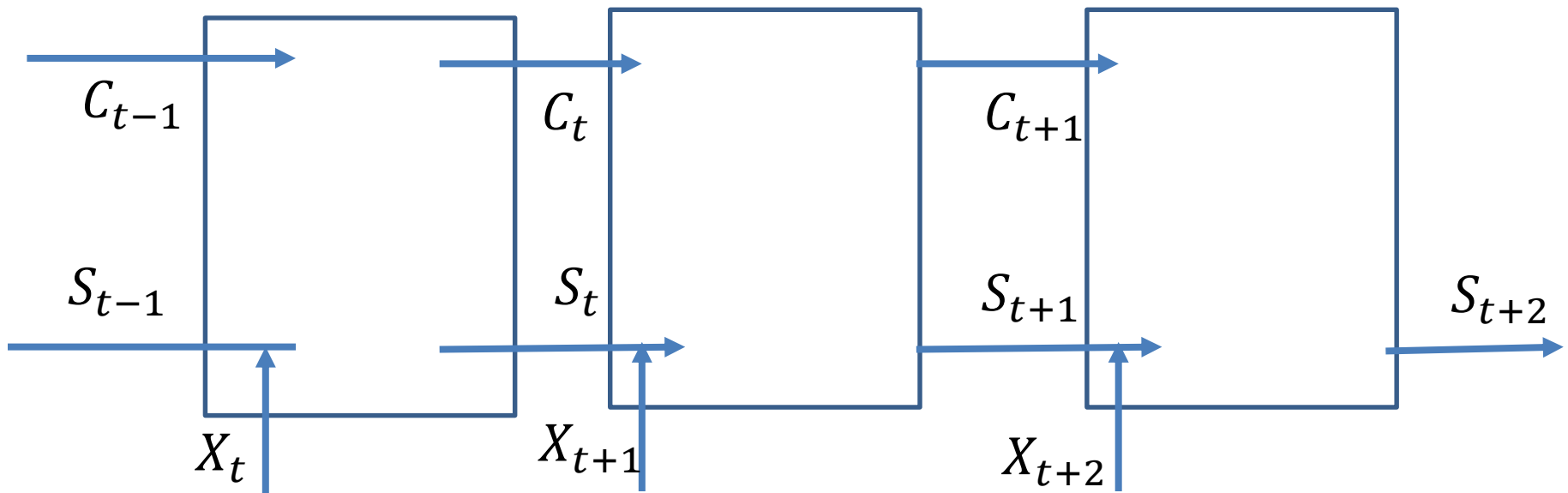


Using LSTM in Keras

As we saw **return_sequences** is a Boolean argument that specifies whether to return the last output in the output sequence, or the full sequence itself.

If we set **return_sequences to False** (which it is set to by default) it will only return the output state of our LSTM (S_{t+2} in the image below).

If set to **True** then it will return all state values output by the LSTM (S_{t+2} , S_{t+1} , S_t , etc). Clearly there will be one state value outputted for each time step. It is necessary to set `return_sequences=True` when **stacking LSTM layers** on top of each other.



Basic LSTM Example in Keras

In this problem we are going to look at building an LSTM model for a very basic sequence problem. We are going to provide an input sequence of numbers such as 1, 2, 3, 4, 5 and we want to build a model that will predict the next number in the sequence 6. You can find the full code for this example [here](#).

A portion of our input sequence is depicted on the left and the corresponding output sequence on the right.

```
[[ 0.  1.  2.  3.  4.]  
 [ 1.  2.  3.  4.  5.]  
 [ 2.  3.  4.  5.  6.]  
 [ 3.  4.  5.  6.  7.]  
 [ 4.  5.  6.  7.  8.]  
 [ 5.  6.  7.  8.  9.]  
 [ 6.  7.  8.  9. 10.]  
 [ 7.  8.  9. 10. 11.]  
 [ 8.  9. 10. 11. 12.]  
 [ 9. 10. 11. 12. 13.]
```

```
[ 5. 6. 7. 8. 9. 10. 11. 12. 13. 14.....].
```



```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

np.random.seed(10)

data = []
for i in range(100):
    sequence = list(range(i, i+5))
    data.append(sequence)
labels = list(range(5, 105))

data = np.array(data, dtype=float)
labels = np.array(labels, dtype=float)

# The range of values is between 0 and 100
# We can normalize by dividing by 100
data = data/100.0
labels = labels/100.0

print (data.shape)
print (labels.shape)

# reshape input to be [samples, time steps, features]
data = data.reshape((data.shape[0], data.shape[1], 1))
```

In the initial code we generate a 2D NumPy array containing the sequence data and the predicted output is generated and stored in labels. See previous slide for a visualization of the contents of data and labels.

Next we normalize the data and print out the shape of data and labels which is (100, 5) and (100,) respectively.

As we mentioned previously the shape of the input data for an LSTM should be the **(number of instances * number of times steps * number of features)**. Which in this case would be (100*5*1). Therefore we must reshape the data to match this shape.

Next we split the data into test and training. Next we create a sequential model. The first layer we add to our model is an LSTM layer.

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2)

model = tf.keras.models.Sequential()

model.add(tf.keras.layers.LSTM((4), input_shape=(5, 1),return_sequences=False))
model.add(tf.keras.layers.Dense(1))

model.compile(loss='mean_absolute_error', optimizer='adam')
model.fit(x_train, y_train, epochs=400, validation_data=(x_test, y_test))

results = model.predict(x_test)

plt.scatter(range(20), results, c='r')
plt.scatter(range(20), y_test, c='g')
plt.show()
```

The first parameter for LSTM is the number of neurons in each layer within the LSTM.

The input size is the **number of time steps * number of features**

The parameter `return_sequences` is set to `False`. Meaning the model only returns the last output of the LSTM unit.

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2)

model = tf.keras.models.Sequential()

model.add(tf.keras.layers.LSTM((4), input_shape=(5, 1),return_sequences=False))
model.add(tf.keras.layers.Dense(1))

model.compile(loss='mean_absolute_error', optimizer='adam',metrics=['accuracy'])
model.fit(x_train, y_train, epochs=400, validation_data=(x_test, y_test))

results = model.predict(x_test)

plt.scatter(range(20), results, c='r')
plt.scatter(range(20), y_test, c='g')
plt.show()
```

Next we pipe the output of our model to a single linear activation neuron. We then complete our model in the normal way. The loss function we use is the mean absolute error. Once the model is build we pipe the test data into the model and collect the results. We then generate a graph that plots the values predicted by the model and the true test labels on a graph.

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2)

model = tf.keras.models.Sequential()

model.add(tf.keras.layers.LSTM(4), input_shape=(5, 1), return_sequences=False))
model.add(tf.keras.layers.Dense(1))

model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=400, validation_data=(x_test, y_test))

results = model.predict(x_test)

plt.scatter(range(20), results, c='r')
plt.scatter(range(20), y_test, c='g')
plt.show()
```

Epoch 396/400

3/3 [=====] - 0s 12ms/step - loss: 0.0087 - val_loss: 0.0113

Epoch 397/400

3/3 [=====] - 0s 16ms/step - loss: 0.0086 - val_loss: 0.0113

Epoch 398/400

3/3 [=====] - 0s 13ms/step - loss: 0.0088 - val_loss: 0.0113

Epoch 399/400

3/3 [=====] - 0s 12ms/step - loss: 0.0087 - val_loss: 0.0115

Epoch 400/400

3/3 [=====] - 0s 13ms/step - loss: 0.0089 - val_loss: 0.0113

```
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2)
```

```
model = tf.keras.models.Sequential
```

```
model.add(tf.keras.layers.LSTM
```

```
model.add(tf.keras.layers.Dense
```

```
model.compile(loss='mean_absolute
```

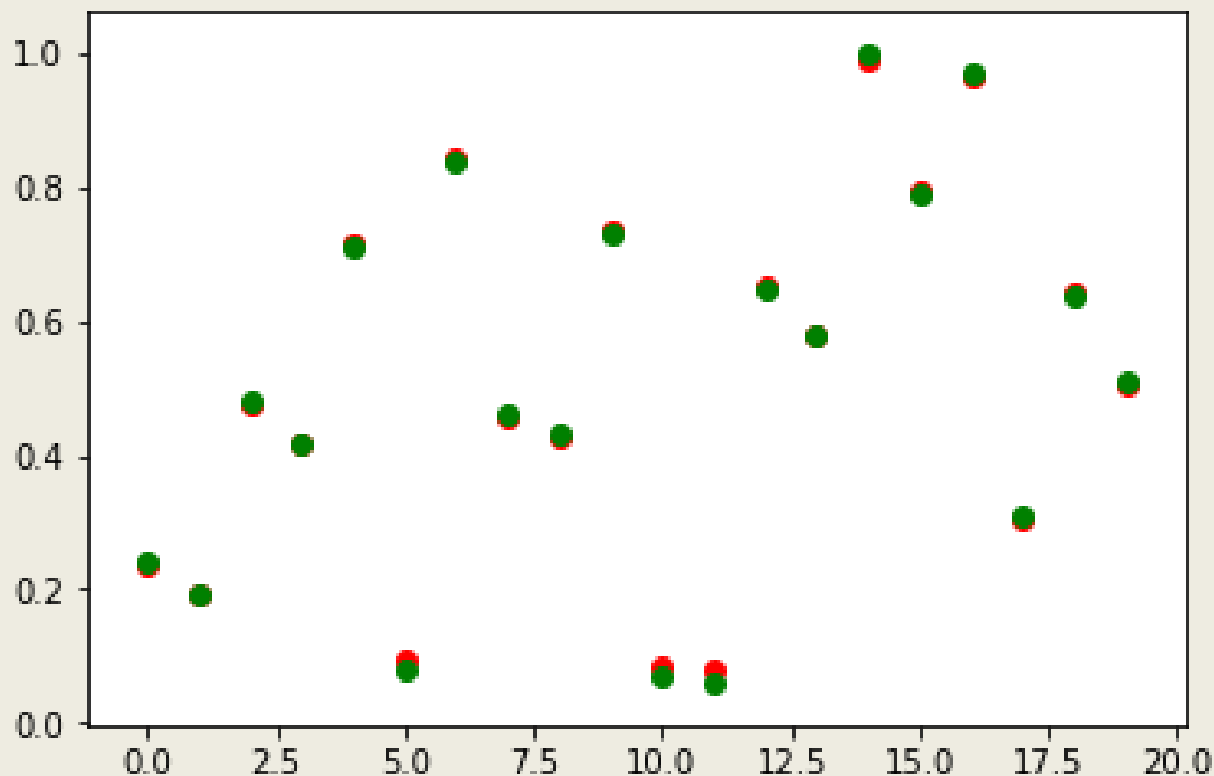
```
model.fit(x_train, y_train, epochs
```

```
results = model.predict(x_test)
```

```
plt.scatter(range(20), results, color
```

```
plt.scatter(range(20), y_test, color
```

```
plt.show()
```



LSTM Example 2

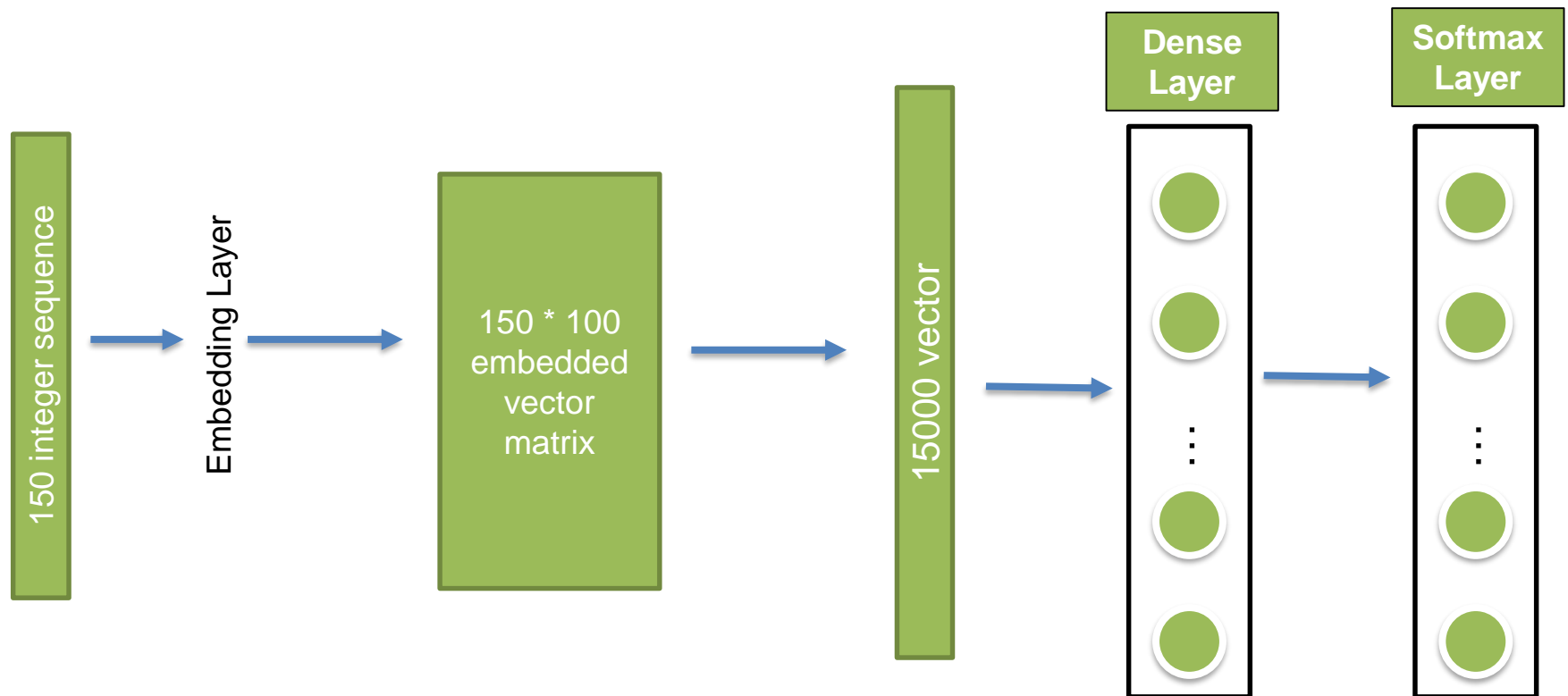
In this example we will return to the **NewsGroup** dataset. You will remember that previously we **preloaded word embeddings**. We used this in conjunction with a **standard neural network architecture**. That is, we fed the output of the word embeddings layer into a densely connected neural network. Unfortunately we were only able to obtain validation accuracy values of approximately 58%.

Now rather than feeding this content directly into a densely connected layer we will instead feed it's into an LSTM layer. You can find the full code for this example [here](#).

As a reminder the original architecture appears on the next slide.

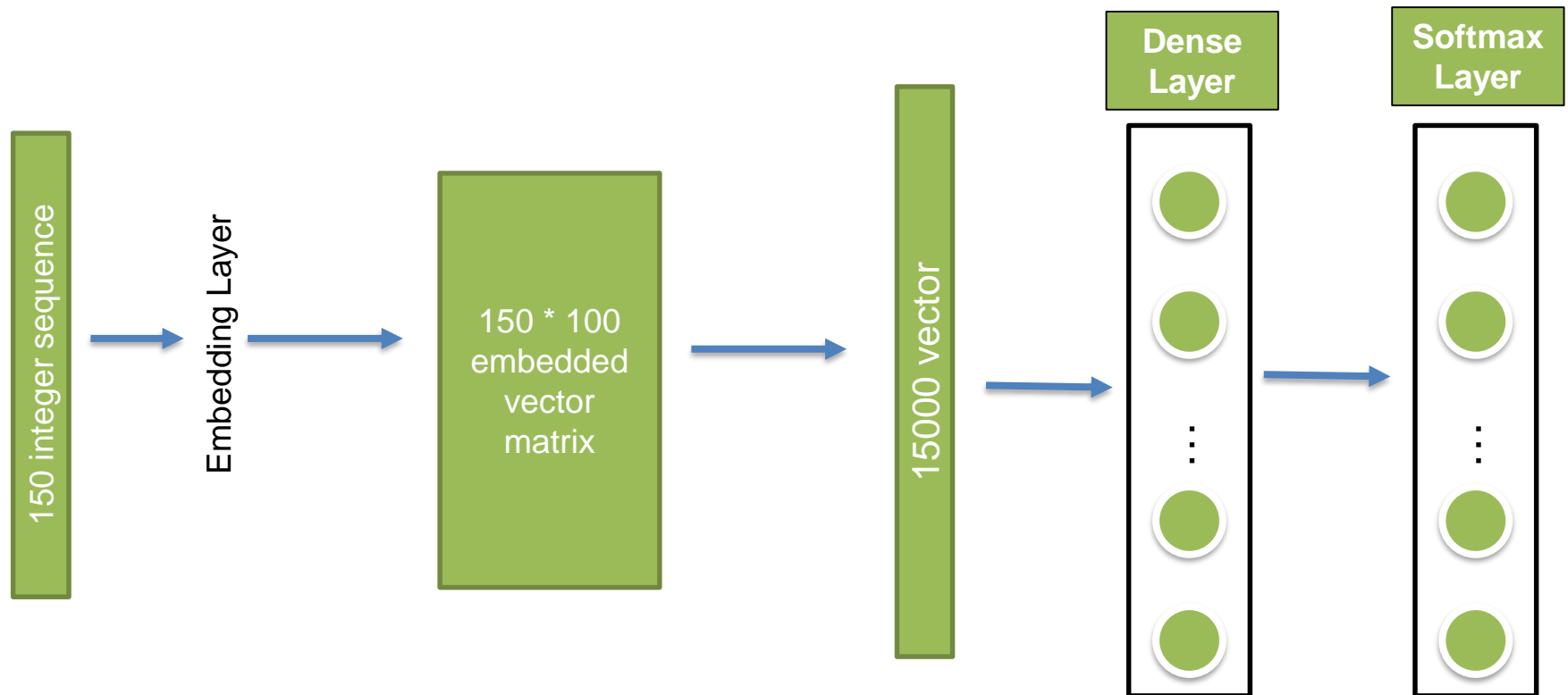
Original Architecture

- The dimensionality of our embedding is 100. Each word is described by an embedded vector of 100 float values.
- Each training document has a fixed sequence length of 150.
- The Embedding layer provides a 2D vector of 150×100 . We flatten this data structure and then feed it into a densely connect layer, which is turn if fed into a softmax layer with 20 neurons.



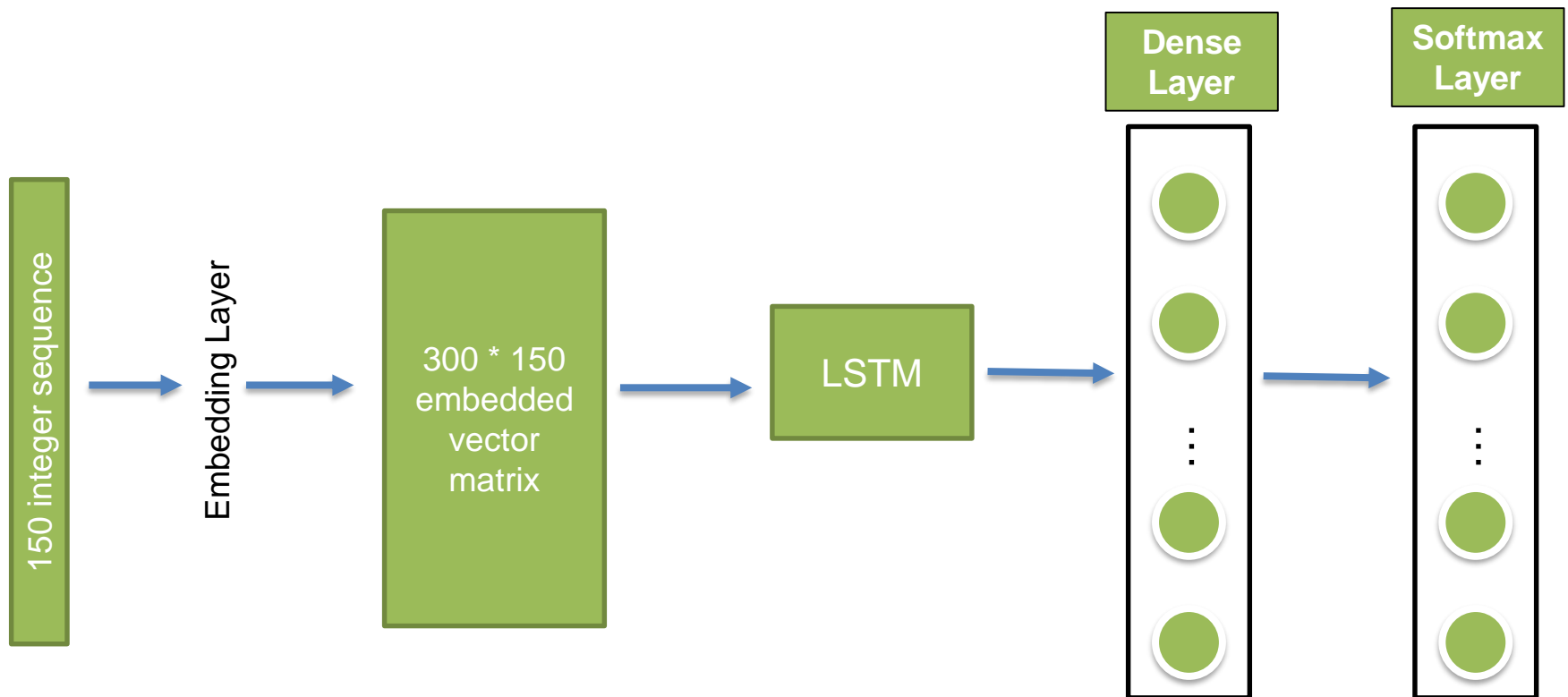
Original Architecture

- If we were to adopt to this an LSTM network. In other words feed the embedding into the network what shape should the result of the embedding layer be (number of steps * num of features)??



LSTM Architecture

- Embedding layer provides a 2D vector of 300×150 .
- The embedded matrix is fed to the LSTM, which is then connected to a densely connected layer, which is then connected to a layer with 20 softmax neurons.
- Notice the embedded matrix is already in the correct shape (**number of time steps** * **number of features**) \Leftrightarrow (words in document * embedding dimensionality)



Original Architecture

Rather than repeating the entire code again. We will just focus on the alteration to the previous version of the code. The **initial network architecture** is depicted below.

```
model = tf.keras.models.Sequential()

# The first layer is an embedded layer
model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,
input_length=MAX_SEQUENCE_LENGTH))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(20, activation='softmax'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy',
metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32,
validation_data=(x_val, y_val))
```

LSTM Example 2

Rather than repeating the entire code again. We will just focus on the alteration to the previous version of the code. The **new network architecture** is depicted below.

```
model = tf.keras.models.Sequential()

# The first layer is an embedded layer
model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,
input_length=MAX_SEQUENCE_LENGTH))

model.add(tf.keras.layers.LSTM(64))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(20, activation='softmax'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_data=(x_val, y_val))
```

LSTM Example 2

Here we are able to obtain an accuracy value of 77%. If we were to increase the length of each document we could push up the accuracy even further.

```
Epoch 1/10
500/500 [=====] - 20s 41ms/step - loss: 2.5728 - acc: 0.1686 - val_loss: 2.0453 - val_acc: 0.3031
Epoch 2/10
500/500 [=====] - 20s 40ms/step - loss: 1.7813 - acc: 0.3809 - val_loss: 1.4369 - val_acc: 0.5249
Epoch 3/10
500/500 [=====] - 20s 40ms/step - loss: 1.2527 - acc: 0.5679 - val_loss: 1.0920 - val_acc: 0.6357
Epoch 4/10
500/500 [=====] - 20s 40ms/step - loss: 0.9528 - acc: 0.6770 - val_loss: 1.0325 - val_acc: 0.6844
Epoch 5/10
500/500 [=====] - 20s 41ms/step - loss: 0.7608 - acc: 0.7507 - val_loss: 0.9401 - val_acc: 0.7109
Epoch 6/10
500/500 [=====] - 20s 40ms/step - loss: 0.6247 - acc: 0.7992 - val_loss: 0.8347 - val_acc: 0.7499
Epoch 7/10
500/500 [=====] - 20s 41ms/step - loss: 0.5139 - acc: 0.8336 - val_loss: 0.8604 - val_acc: 0.7637
Epoch 8/10
500/500 [=====] - 20s 40ms/step - loss: 0.4358 - acc: 0.8624 - val_loss: 0.8513 - val_acc: 0.7677
Epoch 9/10
500/500 [=====] - 20s 40ms/step - loss: 0.3673 - acc: 0.8822 - val_loss: 0.8728 - val_acc: 0.7712
Epoch 10/10
500/500 [=====] - 21s 41ms/step - loss: 0.3152 - acc: 0.8987 - val_loss: 0.8648 - val_acc: 0.7782
```

LSTM Example 2

Rather than repeating the entire code to the previous version of the code below.

Rather than using `tf.keras.layers.LSTM` to train your model it is worth trying the following variants on `tf.compat.v1.keras.layers.CuDNNLSTM`.

The current implementation of LSTM is slow to train. The **CuDNNLSTM** variant greatly accelerates training on some GPUs.

```
model = tf.keras.models.Sequential()
```

```
# The first layer is an embedded layer
```

```
model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,  
input_length=MAX_SEQUENCE_LENGTH))
```

```
model.add(tf.compat.v1.keras.layers.CuDNNLSTM(64))
```

```
model.add(tf.keras.layers.Dropout(0.2))
```

```
model.add(tf.keras.layers.Dense(32, activation='relu'))
```

```
model.add(tf.keras.layers.Dropout(0.2))
```

```
model.add(tf.keras.layers.Dense(20, activation='softmax'))
```

```
model.summary()
```

```
model.layers[0].set_weights([embedding_matrix])
```

```
model.layers[0].trainable = True
```

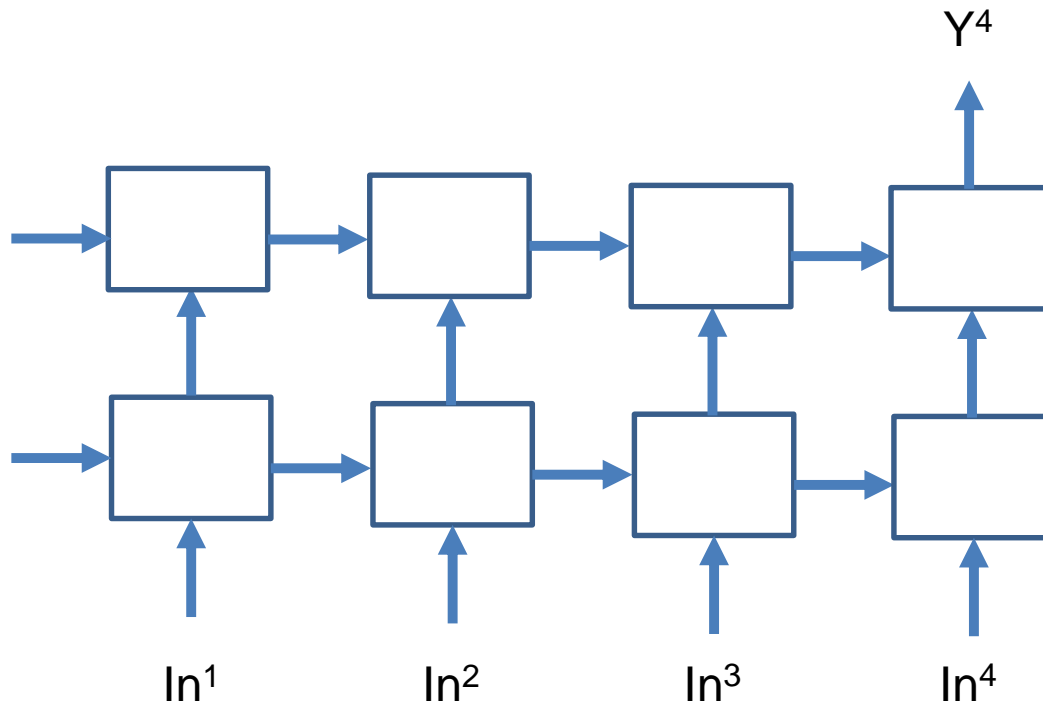
```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
```

```
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_data=(x_val, y_val))
```

LSTM Example 3 - Stacking

In this example we are going to stack two LSTM layers. You can find the full code for this example [here](#).

To facilitate stating the lower LSTM layer must have **return_sequences=True** as the output state for each time step (iteration) will be feed to the LSTM in the next layer.



LSTM Example 3

You will notice below the change we have had to make to the original code is very basic. Highlighted in bold.

```
model = tf.keras.models.Sequential()

model = tf.keras.models.Sequential()
# The first layer is an embedded layer
model.add(tf.keras.layers.Embedding(num_words, EMBEDDING_DIM,
input_length=MAX_SEQUENCE_LENGTH))

model.add(tf.keras.layers.LSTM(32, recurrent_dropout=0.4, return_sequences=True))
model.add(tf.keras.layers.LSTM(64, recurrent_dropout=0.1))
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(20, activation='softmax'))
model.summary()

model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = True

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=20, batch_size=32, validation_data=(x_val, y_val))
```

LSTM Example 3

The deeper network in this case does perform as well as the original LSTM network.

Epoch 1/10 - 593s 37ms/sample - loss: 2.4977 - acc: 0.1708 - val_loss: 2.0160 - val_acc: 0.2843

Epoch 2/10 - 591s 37ms/sample - loss: 1.8778 - acc: 0.3207 - val_loss: 1.6463 - val_acc: 0.4111

Epoch 3/10 - 591s 37ms/sample - loss: 1.5021 - acc: 0.4495 - val_loss: 1.4271 - val_acc: 0.5044

Epoch 4/10 - 592s 37ms/sample - loss: 1.2137 - acc: 0.5639 - val_loss: 1.2009 - val_acc: 0.5819

Epoch 5/10 - 588s 37ms/sample - loss: 0.9869 - acc: 0.6575 - val_loss: 1.0494 - val_acc: 0.6584

Epoch 6/10 - 590s 37ms/sample - loss: 0.8051 - acc: 0.7310 - val_loss: 0.9619 - val_acc: 0.6967

Epoch 7/10 - 592s 37ms/sample - loss: 0.6562 - acc: 0.7872 - val_loss: 0.9076 - val_acc: 0.7199

Epoch 8/10 - 590s 37ms/sample - loss: 0.5526 - acc: 0.8197 - val_loss: 0.8634 - val_acc: 0.7432

Epoch 9/10 - 589s 37ms/sample - loss: 0.4601 - acc: 0.8515 - val_loss: 0.8625 - val_acc: 0.7564

Epoch 10/10 - 592s 37ms/sample - loss: 0.3854 - acc: 0.8773 - val_loss: 0.8896 - val_acc: 0.7509