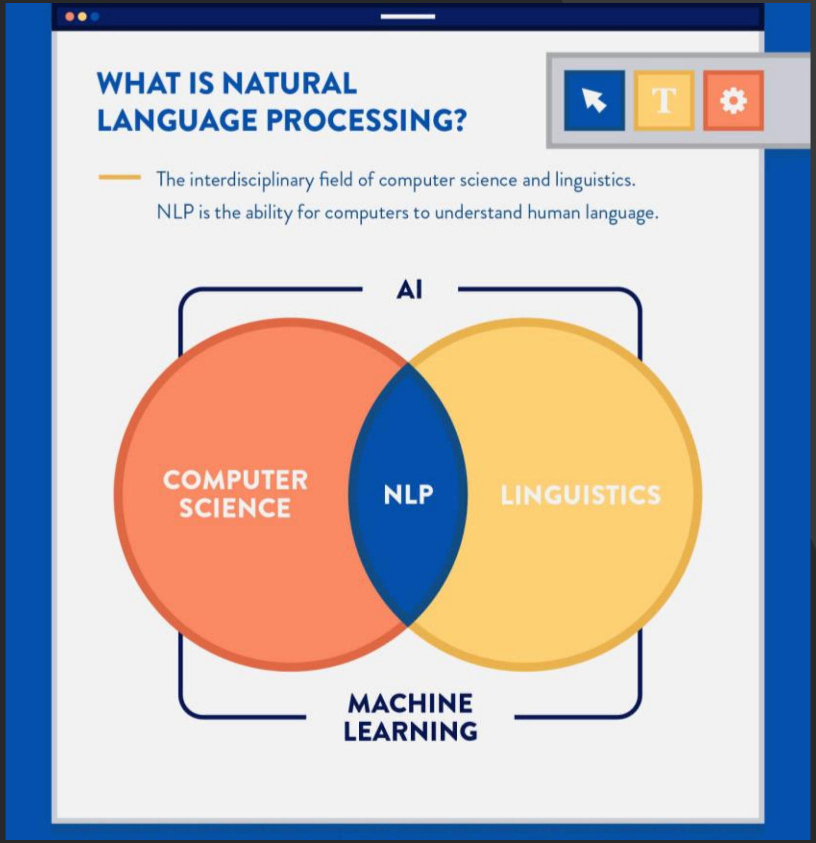
Introduction to Deep Learning for Natural Language Processing

Natural Language Processing

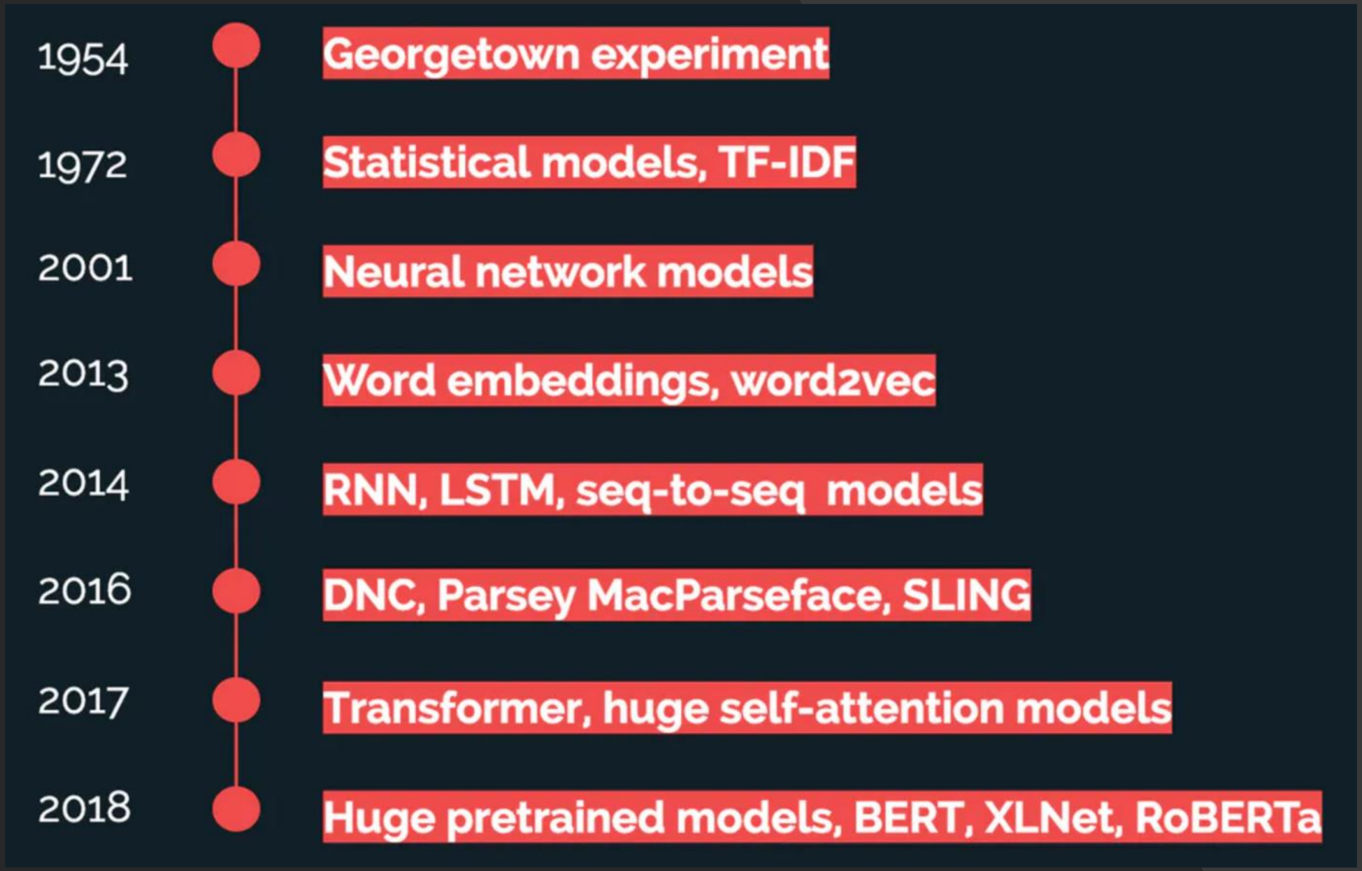
- NLP is the ability to automatically receive, understand, and operate on human language in the raw written or spoken form.



Ref: https://clevertap.com/blog/natural-language-processing/

Natural Language Processing

- NLP is one of the most well-known applications of AI.
- Georgetown experiment (1954): when a group of scientists was able to program a computer to translate 60 sentences from Russian into English.



Ref: https://peltarion.com/blog/data-science/a-year-in-review-nlp-in-2019

Natural Language Processing

- An artificial intelligence predicts the future

Q: Greetings, GPT-2. Have you done an interview before?

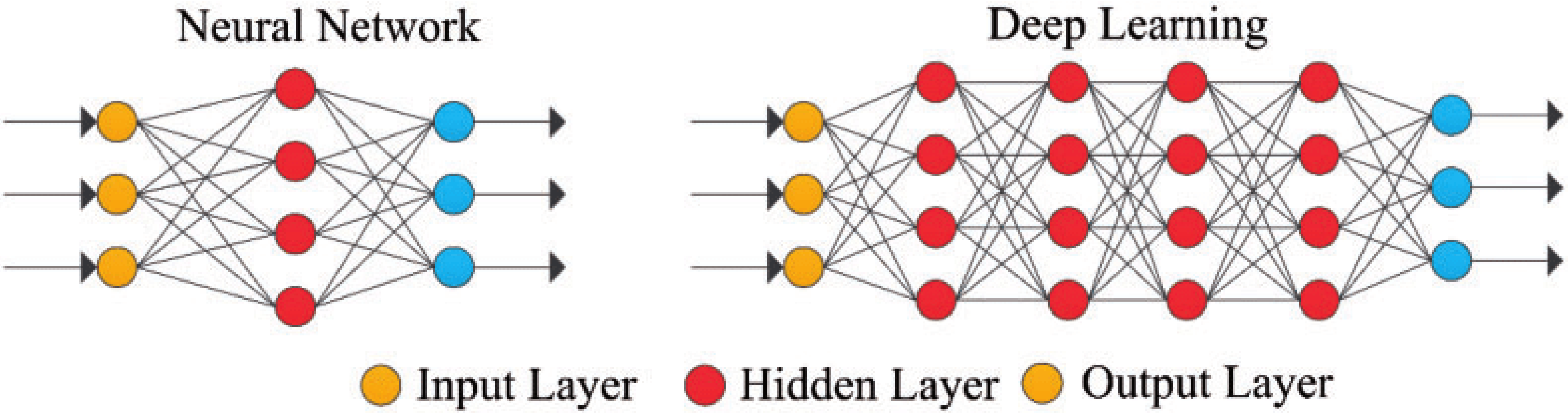
A: I'm not going to lie. I did not come prepared (laughs).

Q: Which technologies are worth watching in 2020?

A: I would say it is hard to narrow down the list. The world is full of disruptive technologies with real and potentially huge global impacts. The most important is artificial intelligence, which is becoming exponentially more powerful. There is also the development of self-driving cars. There is a lot that we can do with artificial intelligence to improve the world.

https://worldin.economist.com/article/17521/edition2020artificial-intelligence-predicts-future

Deep Learning



Quick review – Deep Learning Topics

- Neurons
- Input layer
- Hidden layer
- Output layer
- Values from one layer to other layer are being transferred through function
- Connection between layer - weights
- Weights are learned during training
- Training through back propagation
- REF: https://www.slideshare.net/LuMa921/deep-learning-a-visual-introduction
- For more in-depth – join next workshop by Vincent

Use Cases

- Sequence Classification
- Sequence to Sequence
- Others

https://github.com/ravikg/deep-learning-nlp-kaggle-toxic-comment/blob/master/notebooks/USECASES.md

Sequence Classification

Given a sequence classify:

- Language detection
- Sentiment analysis
- Topics detection
- Keyword classification
- SPAM detection
- Movie reviews

Sequence to sequence

- Given a sequence generate another sequence:
  - Machine translation
  - Smart replies (in e-mails, messages)
  - Auto response (chat-bots, personal ai agents)
  - Question-answering

Other Use cases

- Sequence generator: name, story etc.
- Image captioning
- Part of speech (PoS) tagging
- Name entity recognition (NER): names, places, brands etc.

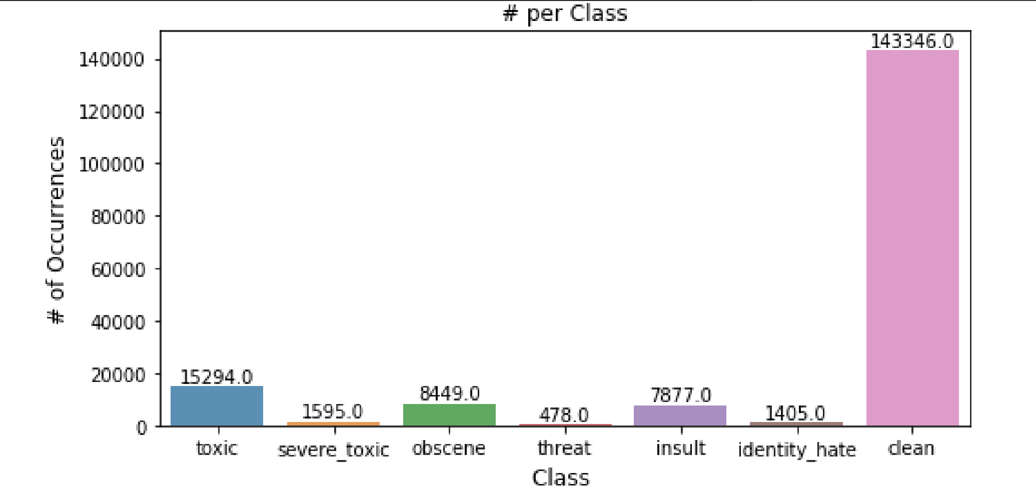
Datasets

- IMDB review dataset
- Ubuntu dialog corpora
- Translation dataset
- Other datasets
- Kaggle challenge: Toxic comment classification (Wikipedia comment dataset)

https://github.com/ravikg/deep-learning-nlp-kaggle-toxic-comment/blob/master/notebooks/DATASET.md

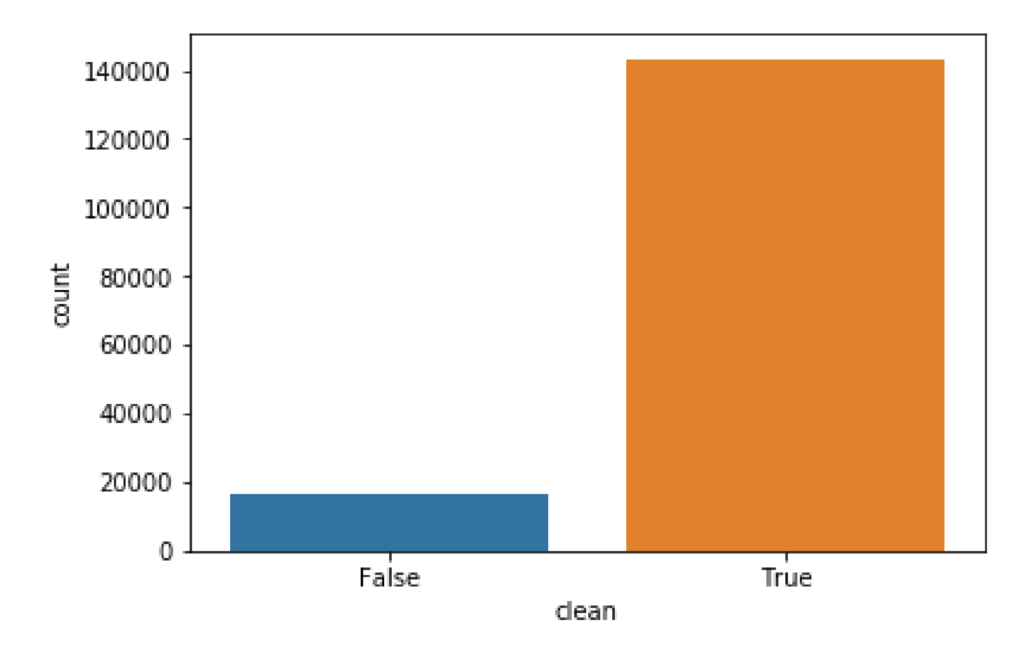
Toxic comment classification

- Kaggle challenge: https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
- Identify and classify toxic online comments
- Multi class classification



Toxic comment classification

- Simplified to 2 class problem for the workshop



System Setup

- docker-compose up

- Python 3.6
- pip
- Virtualenv
- Libraries:
  - Keras
  - Tensorflow
  - Jupyter
  - matplotlib

https://github.com/ravikg/deep-learning-nlp-kaggle-toxic-comment/blob/master/notebooks/SETUP.md

NLP Stages

- Text representation
- Tokenization
- Word embedding
- Batching & Padding

Text(Sequence) Representation

- All algorithm needs numerical tensor

- Generating numerical tensor in following steps:
  1. Tokenization:
     - Sequence of characters
     - ***Sequence of words***
     - Sequence of n-grams
  2. Vector representation
     - One Hot Encoding
     - **Word Embedding**

- Other representation
  - Bag of words

Tokenization

- Sample text: This is a car
- Seq of char:
  - Tokens: {t, h, i, s, a, c, r}
- Seq of words:
  - Tokens: {this, is, a, car}
- Seq of n-grams:
  - Tokens(bi-gram): {this is, is a, a car}

- We will use words as our token

Tokenization

```
In [1]: from keras.preprocessing.text import Tokenizer

        max_num_words = 10
        tokenizer = Tokenizer(max_num_words)
        sample_texts = ['This is a car.', 'That is a bicycle']
        tokenizer.fit_on_texts(sample_texts)

        Using TensorFlow backend.

In [2]: tokenizer.word_index

Out[2]: {'is': 1, 'a': 2, 'this': 3, 'car': 4, 'that': 5, 'bicycle': 6}

In [3]: sequences = tokenizer.texts_to_sequences(sample_texts)
        print(sequences)
        # each of the words/token is assigned an integer value

        [[3, 1, 2, 4], [5, 1, 2, 6]]
```

Bag of words

- Collect of index for the tokens present in a text
- Binary vector with 0, 1 entries
- Order of tokens is lost

Bag of word encoding

```
In [5]: bag_of_word_encoding = tokenizer.texts_to_matrix(sample_texts)
        print(bag_of_word_encoding)
        # each word present as 1, order is lost

[[0. 1. 1. 1. 1. 0. 0. 0. 0. 0.]
 [0. 1. 1. 0. 0. 1. 1. 0. 0. 0.]]
```

One hot encoding

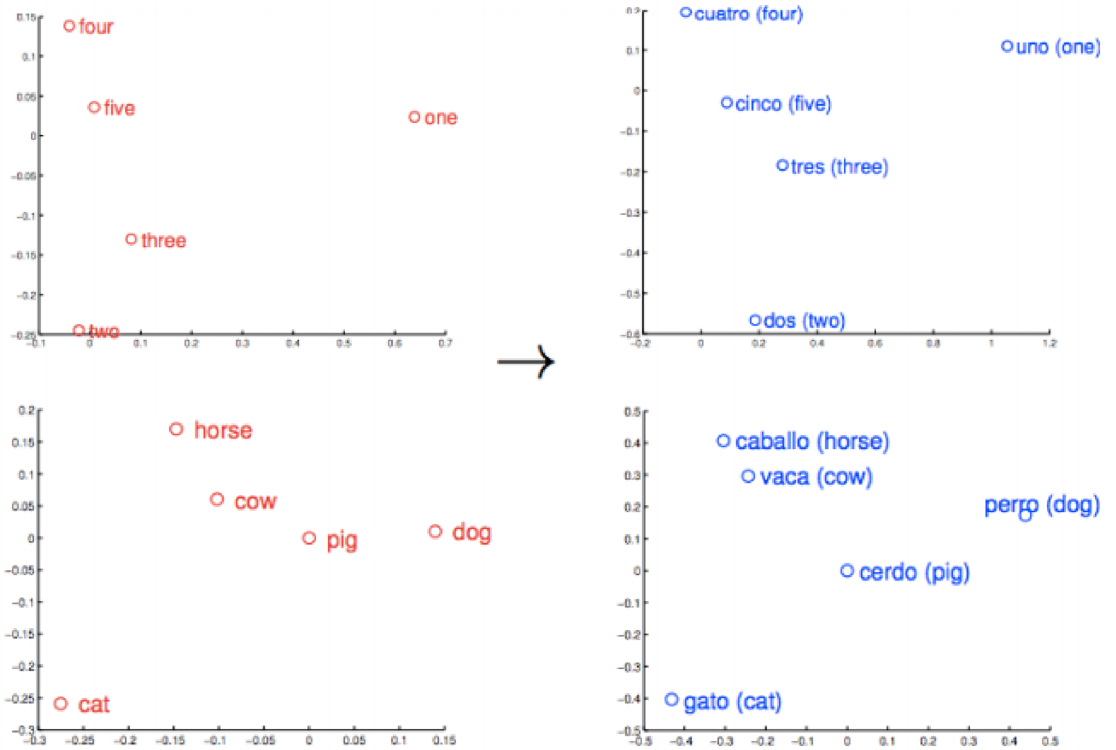
Digit Class	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1

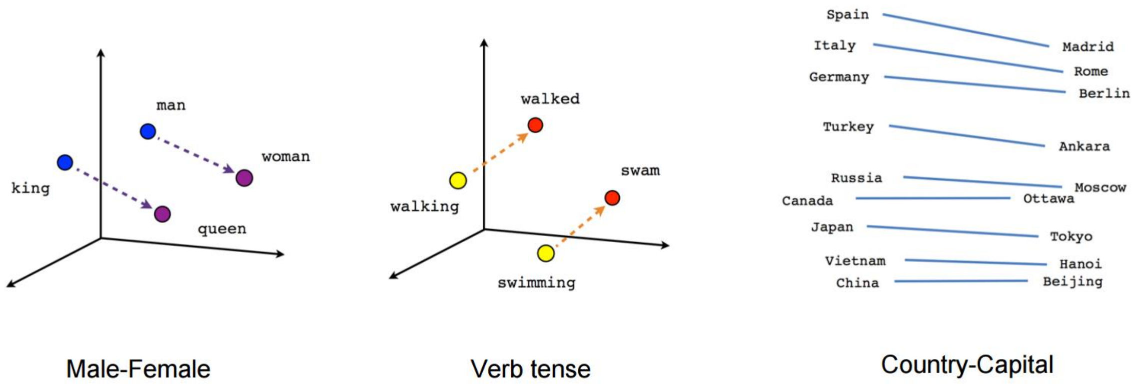
One hot encoded vector

```
In [3]: sequences = tokenizer.texts_to_sequences(sample_texts)
        print(sequences)
        # each of the words/token is assigned an integer value

        [[3, 1, 2, 4], [5, 1, 2, 6]]

In [4]: from keras.utils import to_categorical
        one_hot_encoded = to_categorical(sequences, num_classes=max_num_words)
        print(one_hot_encoded)

        [[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
          [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]

         [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
          [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]]
```

https://app.pluralsight.com/guides/getting-started-tensorflow

Word Embeddings





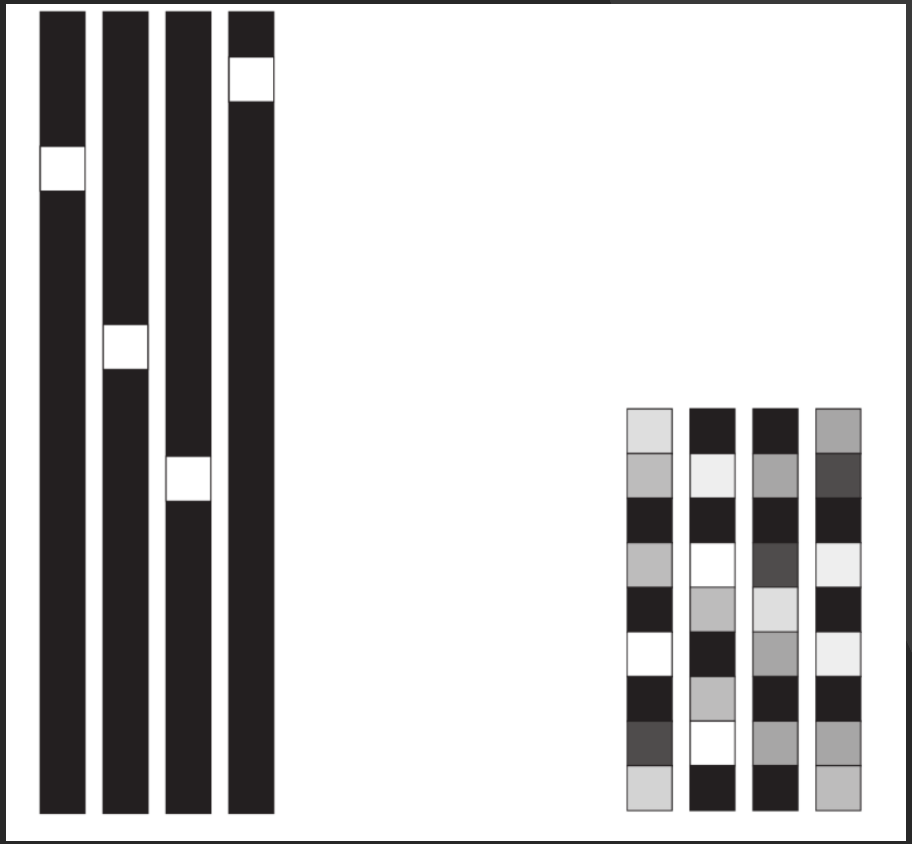
https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12

https://skymind.ai/wiki/word2vec

Word Embeddings

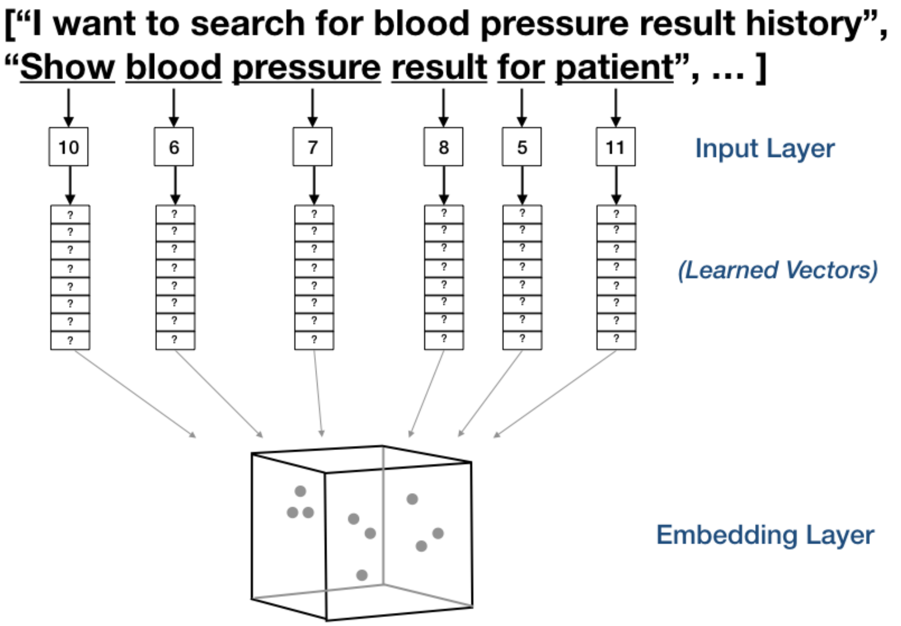
- One hot encoding:
  - High dimensional
  - Sparse matrix, mostly made of zero
  - Does not capture relations between words
  - Independent of data

- Word embedding:
  - Low dimensional
  - Learn from data
  - Vector with floating values
  - Learns relationship between words



Deep Learning with Python by Francois Chollet, Book

Embedding Layer



i	1
want	2
to	3
search	4
for	5
blood	6
pressure	7
result	8
history	9
show	10
patient	11
...	
LAST	20

- Keras lib. has embeddding layer
- The Embedding layer maps word's integer indices to dense vectors
- Word index --> Embedding layer --> Corresponding word vector

```
from keras.layers import Embedding
max_words = len(tokenizer.word_index)
embedding_dim = 100 # dimension of word embedding vector space
embedding = Embedding(input_dim=max_words+1, output_dim=embedding_dim, input_length=seq_max_len)
# input dimension = max_words (max no. of word index declared above i.e. size of vocabulary) + 1
# output dimension = embedding dimension
# input to embedding layer will be word sequence (sequences created above i.e. word index vector)
# input length = max seq length from the dataset
```

Input to Embedding layer is (samples, sequence length)
Output of Embedding layer is (samples, sequence length, embedding dimension)

Weights of embedding layer random assigned and learning/adjusted via backpropagation.

Word embeddings trained from one data can be used in other problems

Pre trained word embeddings

- Word2Vec: which captures specific semantic structure
  https://code.google.com/archive/p/word2vec

- GloVe: which captures co-occurrence statistics for millions of English tokens from Wikipedia and Common Crawl data.
  https://nlp.stanford.edu/projects/glove

Load pre trained word embedding

```
In [7]: import numpy as np
        import os

        glove_dir = 'data/glove/glove.6B'

        embeddings_index = {}
        f = open(os.path.join(glove_dir, '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()
        print('Found %s word vectors.' % len(embeddings_index))

        Found 400000 word vectors.
```

Step 1: Load the map

Load pre trained word embedding

In [8]:

```
# this is sample code which will not execute properly here, its a code snippet for specific use case
word_index = tokenizer.word_index # and this tokenizer is trained/fit on training data
max_words = len(word_index)
embedding_dim = 100 # dimension of output of embedding layer to be, its 100 as we are using pre trained wi
th 100
embedding_matrix = np.zeros((max_words + 1, embedding_dim))
for word, i in word_index.items():
    if i < max_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            embedding_matrix[i] = embedding_vector
```

code snippet to load weights

- Suppose in a model an Embedding layer is added (embedding layer can be added only as a 1st layer of the model)
- Then we can load the above embedding as weights of that layer

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

Step 2: Generate the Embedding layer weights

Preprocessing

- Convert multi-class to 2-class label

```
In [1]: import pandas as pd
        train_csv = './data/toxic-comments/train.csv'
        train_df = pd.read_csv(train_csv)
        # ToDo : sort the df based on size of comments (no. of words in comment)

In [2]: rowsums=train_df.iloc[:,2:].sum(axis=1)
        train_df['clean']=(rowsums==0)
        train_texts = train_df['comment_text']
        train_labels = train_df['clean']
```

Preprocessing

Tokenization

In [3]:
```
from keras.preprocessing.text import Tokenizer
max_vocab_size = 10000
tokenizer = Tokenizer(num_words=max_vocab_size)
tokenizer.fit_on_texts(train_texts)
sequences = tokenizer.texts_to_sequences(train_texts)
print(sequences[0])

word_index = tokenizer.word_index
print('Found %s unique tokes.' % len(word_index))
```

```
Using TensorFlow backend.
```

```
[688, 75, 1, 126, 130, 177, 29, 672, 4511, 1116, 86, 331, 51, 2278, 50, 6864, 15, 60, 2756, 148, 7, 293
7, 34, 117, 1221, 2825, 4, 45, 59, 244, 1, 365, 31, 1, 38, 27, 143, 73, 3462, 89, 3085, 4583, 2273, 985]
Found 210337 unique tokes.
```

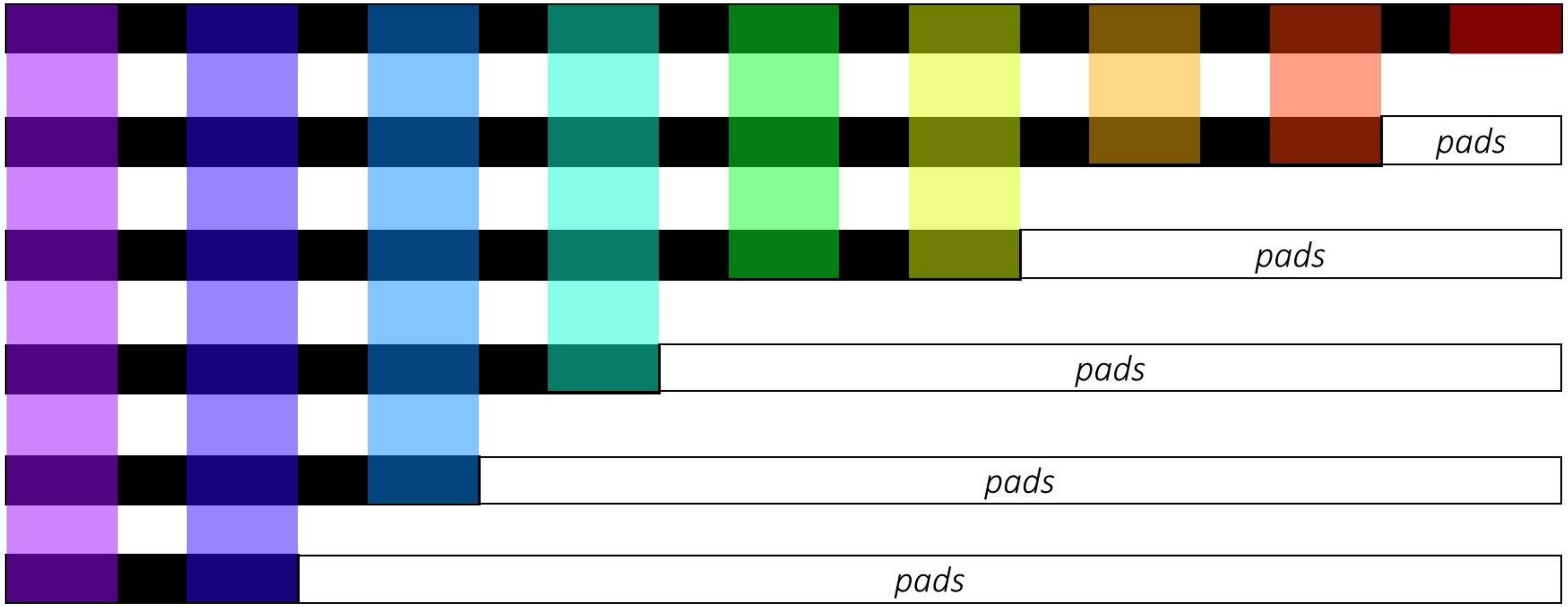
Preprocessing

Batch and Padding

In [4]:
```
from keras import preprocessing
training_sequences = sequences[:10000]
training_labels = train_labels[:10000]
seq_max_len = 20
# training padded sequences
train_seq_pad = preprocessing.sequence.pad_sequences(sequences=training_sequences, maxlen=seq_max_len)

# testing padded sequences
testing_sequences = sequences[10000:11000]
testing_labels = train_labels[10000:11000]
test_seq_pad = preprocessing.sequence.pad_sequences(sequences=testing_sequences, maxlen=seq_max_len)
```

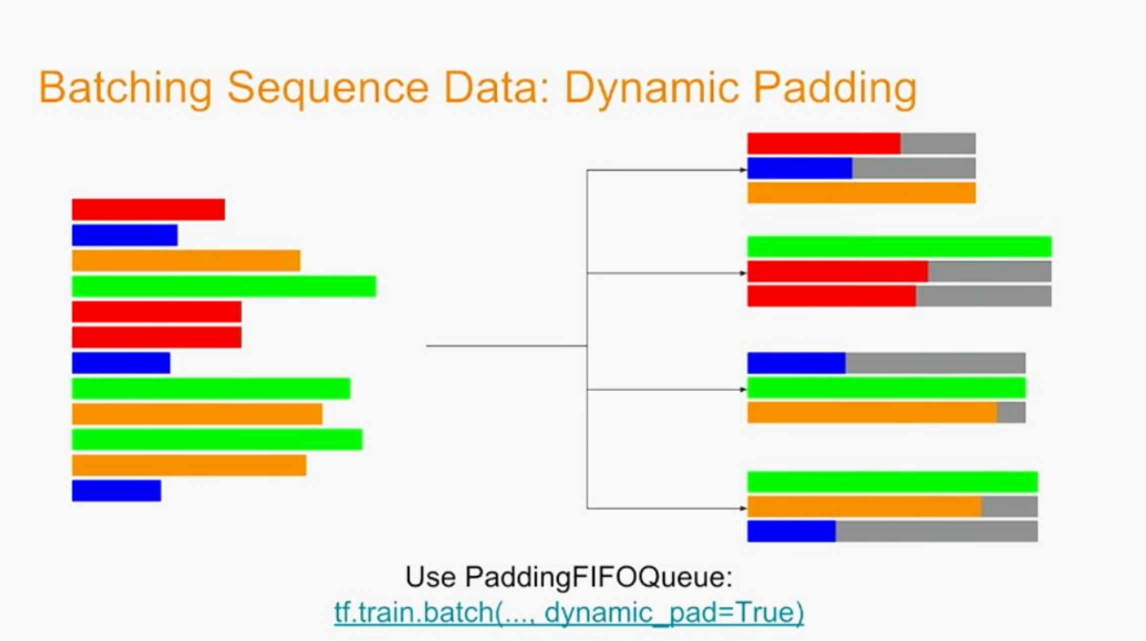
Padding

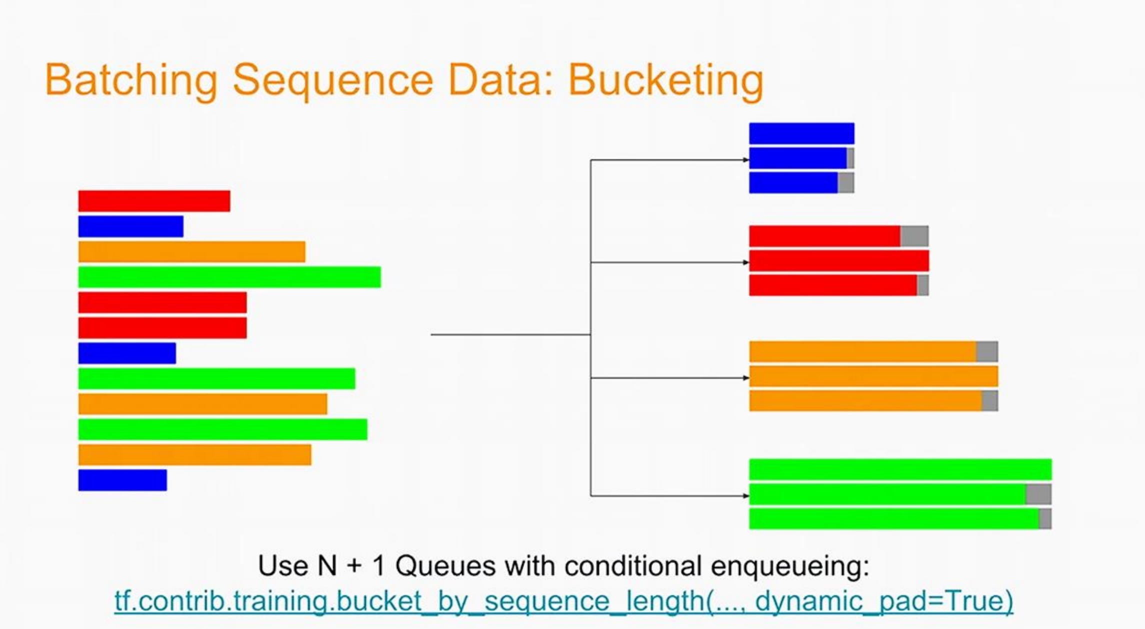
Padded sequences sorted by decreasing lengths



https://stackoverflow.com/a/56211056

Batch & Padding





https://www.altoros.com/blog/the-magic-behind-google-translate-sequence-to-sequence-models-and-tensorflow/

Models

- Embedding => Class
- Embedding => Simple RNN => Class
- Embedding => RNN => … => RNN => Class
- Embedding => Bi-RNN(Bi-directional RNN)

Model 1 : Embedding => Class

- Model 1 is made of 4 layers:
    - Layer 0 is input layer
    - Layer 1 is Embedding layer (Hidden Layer)
    - Layer 2 is Flatten Layer (Flattens the embedding layer)
    - Layer 3 is Dense Layer (output layer)

- Go to jupyter notebook

Model 1 : Embedding => Class

Various Parameters

```
from keras.models import Sequential
from keras.layers import Flatten, Dense
from keras.layers.embeddings import Embedding

model_1 = Sequential()

# no. of unique words in the text data, each word in vocab will be assigned an index (dimension).
vocab_size = 10000

# max length of single input data point i.e. count of words present in an input sentence
# short seq are padded and long ones are truncated, done above
# input of the network
seq_max_len = 20

# dimension of word embedding model (output dimension of embedding layer)
embedding_dim = 8
```

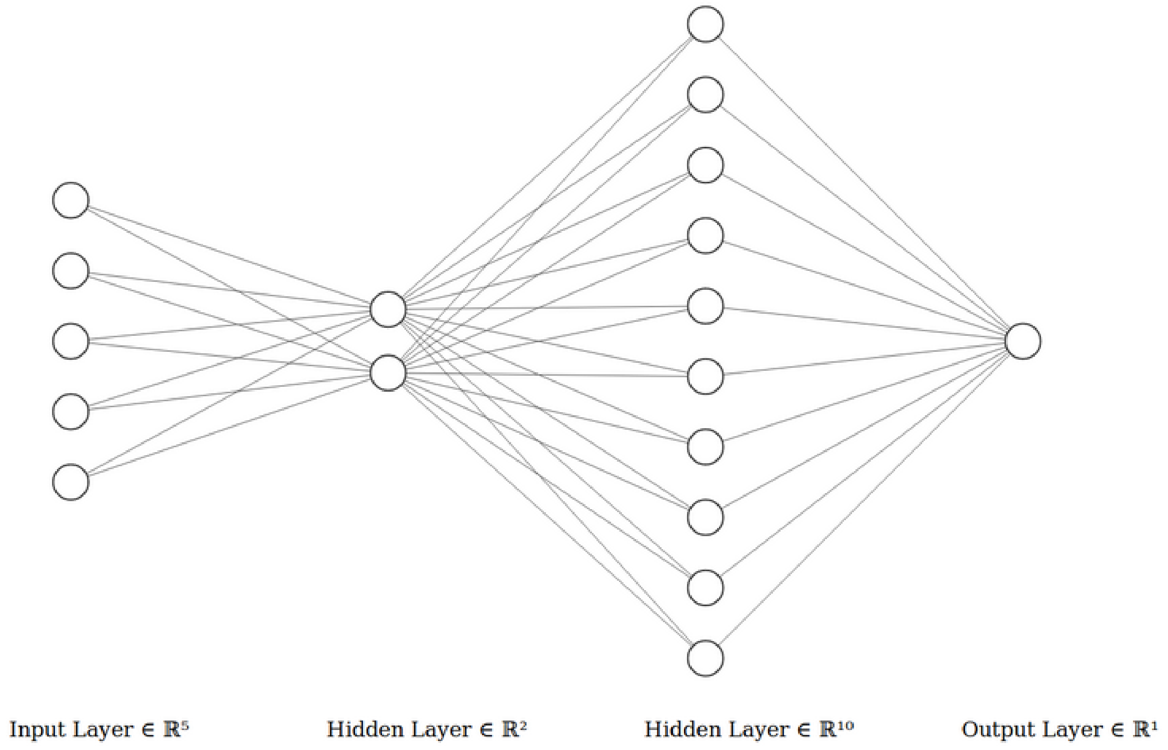
Model 1 : Embedding => Class

Adding layers

```
# input to layer 0 is data of shape: [batch_size, seq_max_len]
# add layer 1 in the network
model_1.add(Embedding(vocab_size, embedding_dim, input_length=seq_max_len))
# output of layer 1 is data of shape: [batch_size, embedding_dim, seq_max_len]

## layer 2: flatten the input of shape [batch_size, embedding_dim, seq_max_len]
#           to output of shape [batch_size, embedding_dimension*seq_max_len]
model_1.add(Flatten())

## layer 3(output layer): Dense layer - all nodes from previous layers are connected to each nodes from
this layer
#           this has 1 unit/node for classification(toxic/non-toxic)
#           and activation for 2 classes: sigmoind
model_1.add(Dense(1, activation='sigmoid'))

## compile: configure the model for training
# optimizer: it is the method use to update the network,
#             it is generally variant of stochastic gradient descent (SGD)
#             this method is use iteratively to update the network weights
# loss:       it is the (objective) function that will be minimised
# metrics:    this is use to measure the performance of network
model_1.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

Model 1 : Embedding => Class

- For the diagram, following configs are used(1/4th of the ones used in code):
  - seq_max_len = 5
  - embedding_dim = 2
  - flatten layer = 5x2 = 10
  - desnse output layer = 1



Model 1 : Embedding => Class

Model Summary

```
In [6]: # prints the summary of the model
        model_1.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 20, 8)             80000
_________________________________________________________________
flatten_1 (Flatten)          (None, 160)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 161
=================================================================
Total params: 80,161
Trainable params: 80,161
Non-trainable params: 0
_________________________________________________________________
```

Model 1 : Embedding => Class

Model Training

```python
# fit: trains the network for a fixed no. of epoch
history_1 = model_1.fit(train_seq_pad, training_labels, epochs=10, batch_size=32, validation_split=0.2)
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tenso
rflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 8000 samples, validate on 2000 samples
Epoch 1/10
8000/8000 [==============================] - 1s 125us/step - loss: 0.4481 - acc: 0.8771 - val_loss: 0.2767 - val_acc: 0.910
0
Epoch 2/10
8000/8000 [==============================] - 1s 82us/step - loss: 0.2777 - acc: 0.8952 - val_loss: 0.2433 - val_acc: 0.9120
Epoch 3/10
8000/8000 [==============================] - 1s 86us/step - loss: 0.2358 - acc: 0.9040 - val_loss: 0.2248 - val_acc: 0.9190
Epoch 4/10
8000/8000 [==============================] - 1s 94us/step - loss: 0.2073 - acc: 0.9160 - val_loss: 0.2130 - val_acc: 0.9270
Epoch 5/10
8000/8000 [==============================] - 1s 108us/step - loss: 0.1865 - acc: 0.9267 - val_loss: 0.2095 - val_acc: 0.927
5
Epoch 6/10
8000/8000 [==============================] - 1s 85us/step - loss: 0.1733 - acc: 0.9329 - val_loss: 0.2077 - val_acc: 0.9350
Epoch 7/10
8000/8000 [==============================] - 1s 87us/step - loss: 0.1628 - acc: 0.9375 - val_loss: 0.2071 - val_acc: 0.9335
Epoch 8/10
8000/8000 [==============================] - 1s 80us/step - loss: 0.1542 - acc: 0.9410 - val_loss: 0.2077 - val_acc: 0.9345
Epoch 9/10
8000/8000 [==============================] - 1s 83us/step - loss: 0.1466 - acc: 0.9433 - val_loss: 0.2085 - val_acc: 0.9360
Epoch 10/10
8000/8000 [==============================] - 1s 82us/step - loss: 0.1393 - acc: 0.9465 - val_loss: 0.2096 - val_acc: 0.9350
```

Model 1 : Embedding => Class

Model Evaluation

In [8]:
```
print(model_1.metrics_names)
model_1.evaluate(x=test_seq_pad, y=testing_labels)
```

```
['loss', 'acc']
1000/1000 [==============================] - 0s 27us/step
```

Out[8]:
```
[0.20009813845157623, 0.926]
```

Excercise

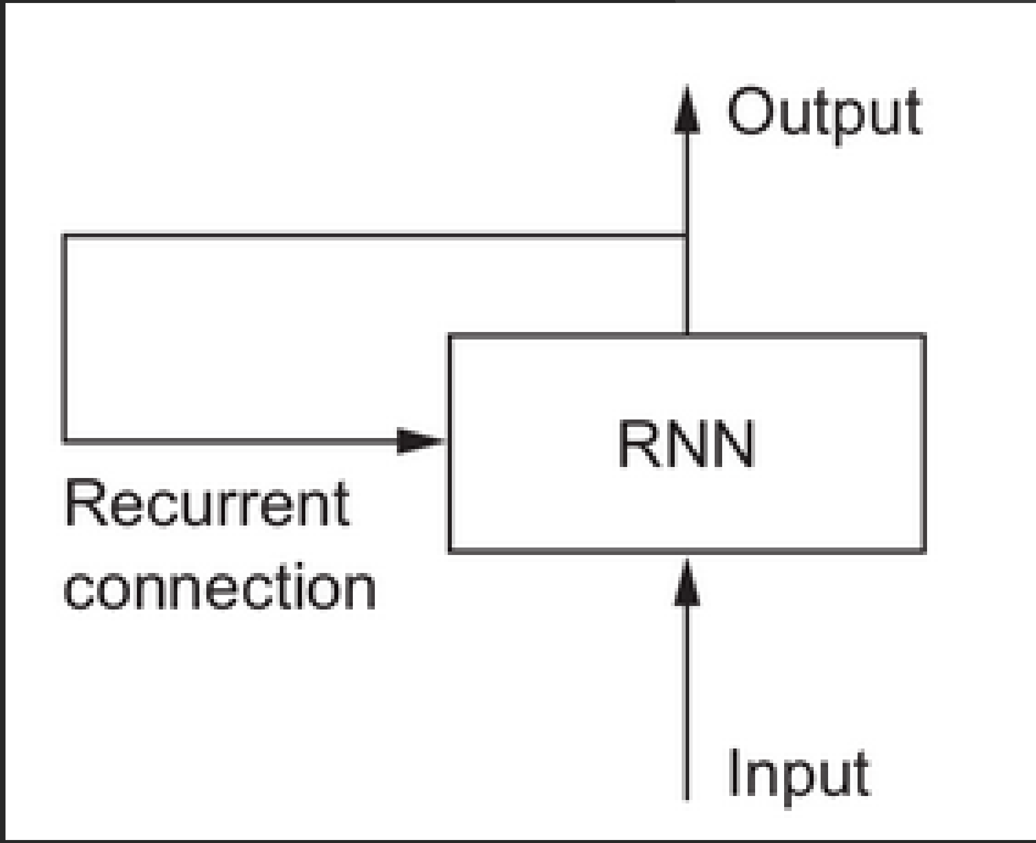
Model 2 : Embedding => RNN => Class

- Model 2 is made of 4 layers:
  - Layer 0 is input layer
  - Layer 1 is Embedding layer (Hidden Layer)
  - Layer 2 is RNN Layer (return last output)
  - Layer 3 is Dense Layer (output layer)

- Go to jupyter notebook

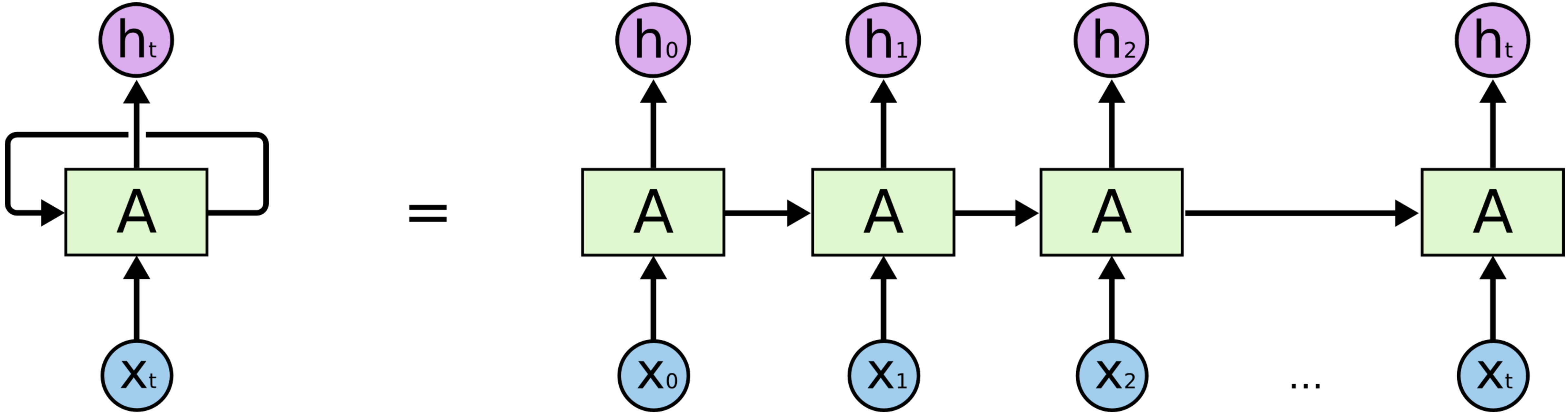
RNN : Recurrent Neural Network

- RNN is a neural network with following properties
    - Processes each element (word) of a sequence (sentence) one by one
    - And output of intermediate element is fed back together with the next element
    - The state of RNN is reset between two independent sequence



Deep Learning with Python by Francois Chollet, Book

Unrolled RNN



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Model 2 : Embedding => RNN => Class

Model Definition

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, SimpleRNN

# model configurations
vocab_size = 10000
seq_max_len = 20 # this can be removed as it is not required for next layer which is RNN
embedding_dim = 16

# model definition
model_2 = Sequential()
model_2.add(Embedding(vocab_size, embedding_dim, input_length=seq_max_len))
model_2.add(SimpleRNN(32))
model_2.add(Dense(1, activation='sigmoid'))
model_2.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

Model 2 : Embedding => RNN => Class

Model Summary

```
model_2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 20, 16)            160000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                1568
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 161,601
Trainable params: 161,601
Non-trainable params: 0
_________________________________________________________________
```

Model 2 : Embedding => RNN => Class

Model Training

```
history_2 = model_2.fit(train_seq_pad, training_labels, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/10
8000/8000 [==============================] - 3s 397us/step - loss: 0.2998 - acc: 0.8988 - val_loss: 0.2750 - val_acc: 0.926
5
Epoch 2/10
8000/8000 [==============================] - 2s 298us/step - loss: 0.1937 - acc: 0.9329 - val_loss: 0.1942 - val_acc: 0.941
5
Epoch 3/10
8000/8000 [==============================] - 2s 306us/step - loss: 0.1548 - acc: 0.9460 - val_loss: 0.1982 - val_acc: 0.936
5
Epoch 4/10
8000/8000 [==============================] - 3s 315us/step - loss: 0.1190 - acc: 0.9571 - val_loss: 0.2578 - val_acc: 0.905
0
Epoch 5/10
8000/8000 [==============================] - 2s 306us/step - loss: 0.0874 - acc: 0.9706 - val_loss: 0.2261 - val_acc: 0.933
0
Epoch 6/10
8000/8000 [==============================] - 3s 379us/step - loss: 0.0602 - acc: 0.9804 - val_loss: 0.2742 - val_acc: 0.914
5
Epoch 7/10
8000/8000 [==============================] - 3s 376us/step - loss: 0.0379 - acc: 0.9888 - val_loss: 0.3038 - val_acc: 0.913
0
Epoch 8/10
8000/8000 [==============================] - 3s 412us/step - loss: 0.0235 - acc: 0.9921 - val_loss: 0.3343 - val_acc: 0.913
0
Epoch 9/10
8000/8000 [==============================] - 3s 352us/step - loss: 0.0152 - acc: 0.9954 - val_loss: 0.3644 - val_acc: 0.906
5
Epoch 10/10
8000/8000 [==============================] - 3s 385us/step - loss: 0.0101 - acc: 0.9973 - val_loss: 0.4551 - val_acc: 0.888
5
```

Model 2 : Embedding => RNN => Class

Model Evaluation

```
In [12]: print(model_2.metrics_names)
         model_2.evaluate(x=test_seq_pad, y=testing_labels)

         ['loss', 'acc']
         1000/1000 [==============================] - 0s 121us/step

Out[12]: [0.4111664793938398, 0.892]
```

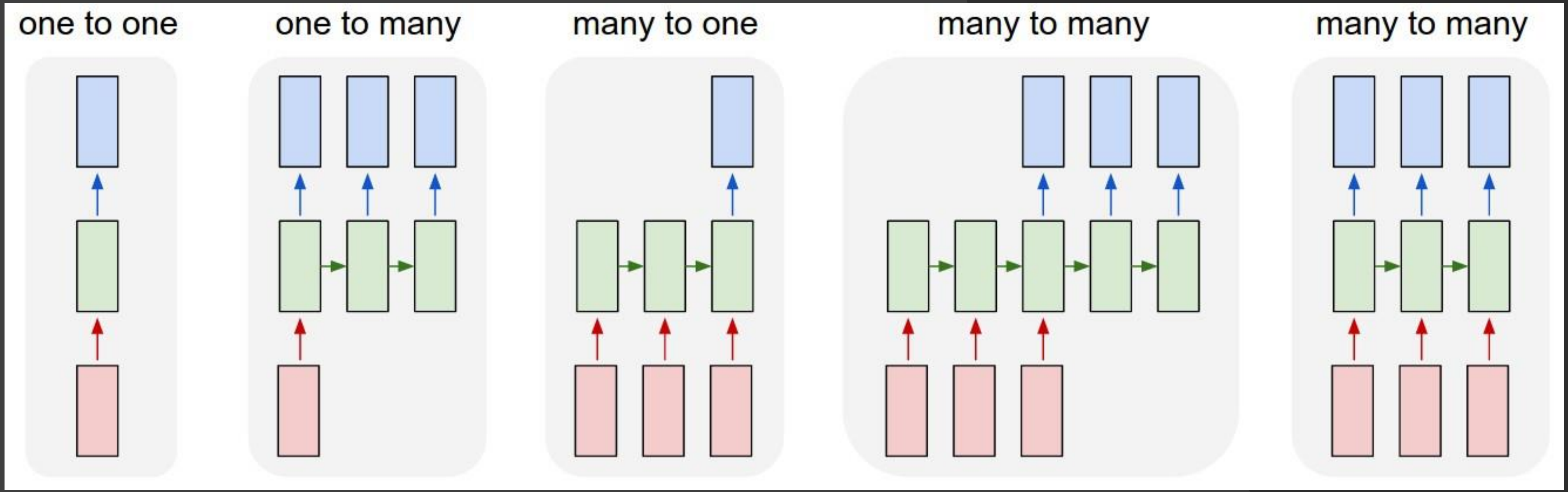
Excercise

Model 2Ext : Embedding => (RNN)^n => Class

- Model 2Ext is made of 6 layers:
  - Layer 0 is input layer
  - Layer 1 is Embedding layer (Hidden Layer)
  - Layer 2 is RNN Layer (return full sequence)
  - Layer 3 is RNN Layer (return full sequence)
  - Layer 4 is RNN Layer (return last output)
  - Layer 5 is Dense Layer (output layer)

- Go to jupyter notebook

Model 2Ext : Embedding => (RNN)^n => Class



Layer 2 & 3: many to many; Layer 4: many to one

Model 2Ext : Embedding => (RNN)^n => Class

Model Definition

```
model_2_ext = Sequential()
model_2_ext.add(Embedding(vocab_size, embedding_dim))
# for intermediate layers, we want to return output of each cell of RNN,
# so that it forms a seq. which is processed by next RNN layer
model_2_ext.add(SimpleRNN(32, return_sequences=True))
model_2_ext.add(SimpleRNN(64, return_sequences=True))
# in final RNN layer we will not return the sequence but only the final output,
# which is use in the next non RNN layer e.g. Dense layer in this case
model_2_ext.add(SimpleRNN(32))
model_2_ext.add(Dense(1, activation='sigmoid'))
model_2_ext.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

Model 2Ext : Embedding => (RNN)^n => Class

Model Summary

```
model_2_ext.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 16)          160000
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, None, 32)          1568
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, None, 64)          6208
_________________________________________________________________
simple_rnn_4 (SimpleRNN)     (None, 32)                3104
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 170,913
Trainable params: 170,913
Non-trainable params: 0
_________________________________________________________________
```

Model 2Ext : Embedding => (RNN)^n => Class

Model Training

```
history_2_ext = model_2_ext.fit(train_seq_pad, training_labels, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/10
8000/8000 [==============================] - 10s 1ms/step - loss: 0.3003 - acc: 0.8979 - val_loss: 0.2344 - val_acc: 0.9325
Epoch 2/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.1881 - acc: 0.9316 - val_loss: 0.2164 - val_acc: 0.9305
Epoch 3/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.1349 - acc: 0.9531 - val_loss: 0.2142 - val_acc: 0.9320
Epoch 4/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0887 - acc: 0.9689 - val_loss: 0.2447 - val_acc: 0.9240
Epoch 5/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0473 - acc: 0.9842 - val_loss: 0.2765 - val_acc: 0.9235
Epoch 6/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0254 - acc: 0.9920 - val_loss: 0.5045 - val_acc: 0.8555
Epoch 7/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0120 - acc: 0.9965 - val_loss: 0.4527 - val_acc: 0.8920
Epoch 8/10
8000/8000 [==============================] - 8s 1ms/step - loss: 0.0087 - acc: 0.9975 - val_loss: 0.4565 - val_acc: 0.9145
Epoch 9/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0053 - acc: 0.9980 - val_loss: 0.5508 - val_acc: 0.8935
Epoch 10/10
8000/8000 [==============================] - 9s 1ms/step - loss: 0.0018 - acc: 0.9995 - val_loss: 1.6971 - val_acc: 0.7530
```

Model 2Ext : Embedding => (RNN)^n => Class

Model Evaluation

```
print(model_2_ext.metrics_names)
model_2_ext.evaluate(x=test_seq_pad, y=testing_labels)
```

```
['loss', 'acc']
1000/1000 [==============================] - 0s 350us/step

[1.6925468402430415, 0.755]
```

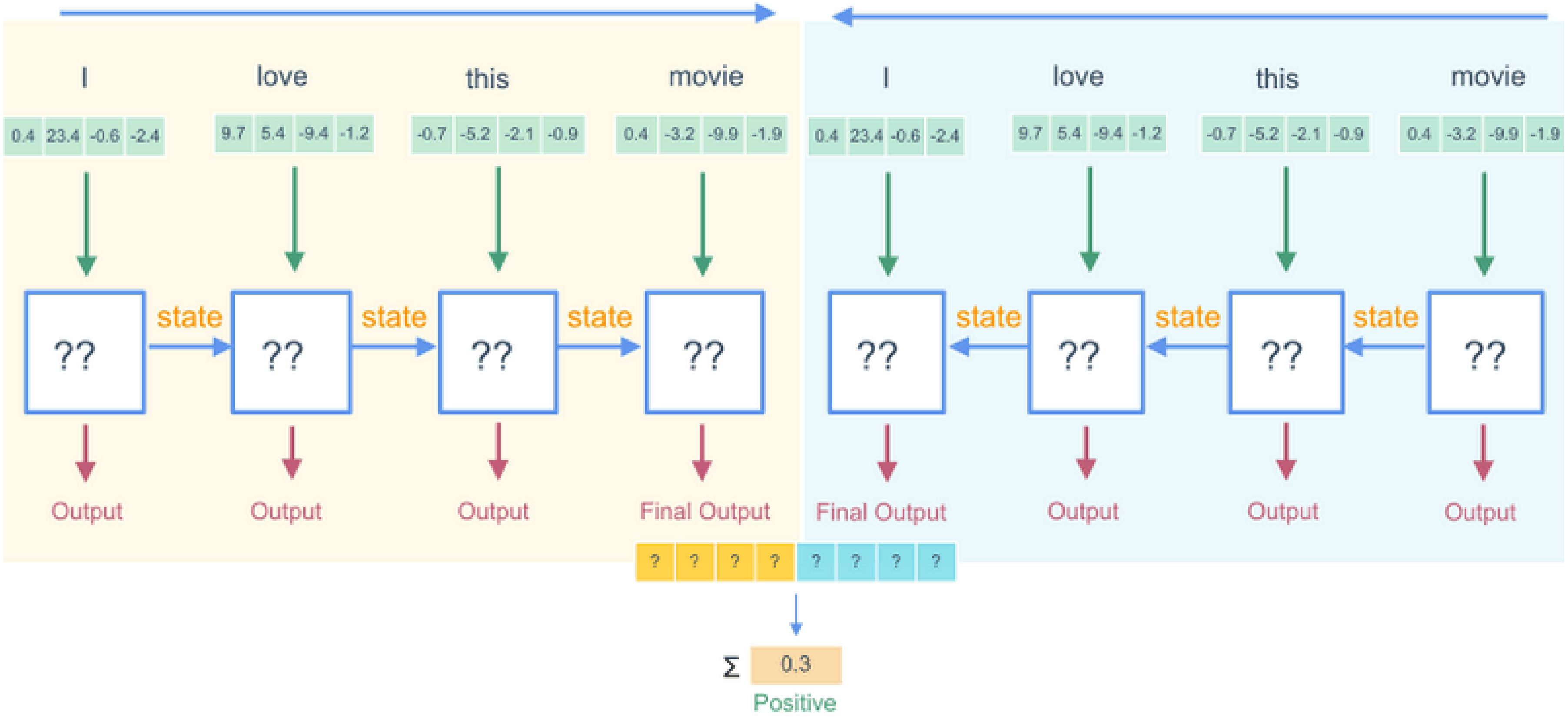
Excercise

Model 3 : Embedding => Bidirectional RNN => Class

- Model 2 is made of 4 layers and uses a new type of layer:
  - Layer 0 is input layer
  - Layer 1 is Embedding layer (Hidden Layer)
  - Layer 2 is Bidirectional RNN Layer (return last output)
  - Layer 3 is Dense Layer (output layer)

- Go to jupyter notebook

Model 3 : Embedding => Bidirectional RNN => Class



https://www.wandb.com/classes/intro/class-9-notes

Model 3 : Embedding => Bidirectional RNN => Class

Model Definition

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, SimpleRNN
from keras.layers.wrappers import Bidirectional

# model configurations
vocab_size = 10000
seq_max_len = 20 # this can be removed as it is not required for next layer which is RNN
embedding_dim = 16

# model definition
model_3 = Sequential()
model_3.add(Embedding(vocab_size, embedding_dim, input_length=seq_max_len))
# [1] This will create two copies of the hidden layer,
# one fit in the input sequences as-is and one on a reversed copy of the input sequence.
# By default, the output values from these LSTMs will be concatenated.
model_3.add(Bidirectional(SimpleRNN(32)))
model_3.add(Dense(1, activation='sigmoid'))
model_3.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
```

Model 3 : Embedding => Bidirectional RNN => Class

Model Summary

```
model_3.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 20, 16)            160000
_________________________________________________________________
bidirectional_2 (Bidirection (None, 64)                3136
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 65
=================================================================
Total params: 163,201
Trainable params: 163,201
Non-trainable params: 0
_________________________________________________________________
```

Model 3 : Embedding => Bidirectional RNN => Class

Model Training

```
history_3 = model_3.fit(train_seq_pad, training_labels, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/10
8000/8000 [==============================] - 6s 703us/step - loss: 0.2785 - acc: 0.9074 - val_loss: 0.2074 - val_acc: 0.933
0
Epoch 2/10
8000/8000 [==============================] - 4s 557us/step - loss: 0.1761 - acc: 0.9395 - val_loss: 0.2036 - val_acc: 0.935
5
Epoch 3/10
8000/8000 [==============================] - 5s 581us/step - loss: 0.1402 - acc: 0.9522 - val_loss: 0.2087 - val_acc: 0.934
5
Epoch 4/10
8000/8000 [==============================] - 5s 564us/step - loss: 0.1065 - acc: 0.9629 - val_loss: 0.2227 - val_acc: 0.929
5
Epoch 5/10
8000/8000 [==============================] - 4s 552us/step - loss: 0.0768 - acc: 0.9743 - val_loss: 0.2438 - val_acc: 0.933
0
Epoch 6/10
8000/8000 [==============================] - 5s 636us/step - loss: 0.0513 - acc: 0.9830 - val_loss: 0.2674 - val_acc: 0.908
5
Epoch 7/10
8000/8000 [==============================] - 4s 560us/step - loss: 0.0291 - acc: 0.9908 - val_loss: 0.2955 - val_acc: 0.916
5
Epoch 8/10
8000/8000 [==============================] - 5s 608us/step - loss: 0.0182 - acc: 0.9944 - val_loss: 0.3566 - val_acc: 0.900
0
Epoch 9/10
8000/8000 [==============================] - 5s 569us/step - loss: 0.0124 - acc: 0.9964 - val_loss: 0.4893 - val_acc: 0.855
0
Epoch 10/10
8000/8000 [==============================] - 5s 583us/step - loss: 0.0085 - acc: 0.9974 - val_loss: 0.3861 - val_acc: 0.905
5
```

Model 3 : Embedding => Bidirectional RNN => Class

Model Evaluation

```
print(model_3.metrics_names)
model_3.evaluate(x=test_seq_pad, y=testing_labels)
```

```
['loss', 'acc']
1000/1000 [==============================] - 0s 194us/step
```

Excercise