

# COMP3420 Lesson 10

Greg Baker

2023-05-08

## Contents

<b>1</b>	<b>Bag-of-words problems</b>	<b>2</b>
<b>2</b>	<b>Wordnet</b>	<b>5</b>
<b>3</b>	<b>What relu does</b>	<b>10</b>
<b>4</b>	<b>Word Embeddings</b>	<b>13</b>
<b>5</b>	<b>Using embeddings</b>	<b>18</b>
<b>6</b>	<b>Drift</b>	<b>21</b>
<b>7</b>	<b>Wrap-up</b>	<b>22</b>

## Today's lesson

- Bag-of-words review and its problems.
- Wordnet
- Embeddings
- Example of using embeddings for colours
- Using embeddings without sequences
- Context drift

## Reading

- Chollet: Section 11.3.3
- Jurafsky and Martin, Chapter 9 (optional)

## Other administrative matters

Student survey for COMP3420 is now open.

We don't see it until after your exams, so be honest!

This is my first time teaching at Macquarie, so I'm very interested to hear feedback.

If you aren't happy with something in the course that you want fixed urgently, talk to Abid or email me ([greg.baker@mq.edu.au](mailto:greg.baker@mq.edu.au))

## 1 Bag-of-words problems

### Homographic Ambiguities Everywhere

Language features ambiguity at multiple levels.

#### Lexical Ambiguity

Example from Google's dictionary:

- bank (n): the land alongside or sloping down a river or lake.
- bank (n): financial establishment that uses money deposited by customers for investment, ...
- bank (v): form in to a mass or mound.
- bank (v): build (a road, railway, or sports track) higher at the outer edge of a bend to facilitate fast cornering.
- ...

### Long-distance Dependencies

- Sentences are sequences of words.
- Words close in the sentence are often related.
- But, sometimes, there are relations between words far apart.

**grammatical:** “The man living upstairs is very cheerful” “The people living upstairs are very cheerful”

**knowledge:** “I was born in France and I speak fluent French”

**reference:** “I bought a book from the shopkeeper and I liked it”

From the two examples above we can see that words that may be far from the gap determine what is the best word that fills the gap.

The third example shows that the reference of a pronoun can be fairly far from the pronoun itself.

### Cross-culture concepts

A word might-or-might-not be translatable across different cultures or languages.



Dinner  
Dinner

### Review of bag-of-words model for embedding text

- ## 1 BAG-OF-WORDS PROBLEMS

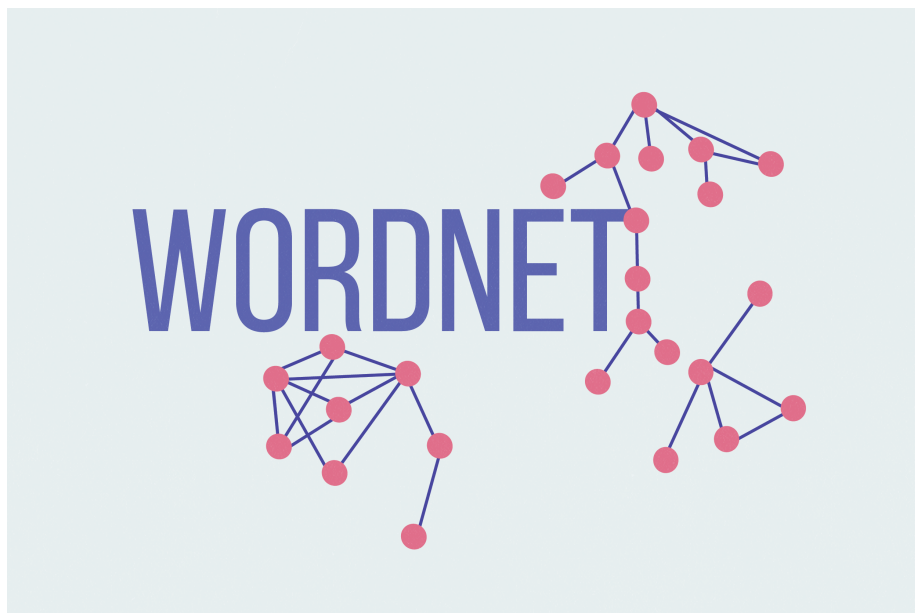
– *Commence*

- Word order is ignored (unless using bigrams)
- No way of handling long-distance dependencies
- Context ignored

## 2 Wordnet

### Introduction to WordNet

- Developed by Princeton University
- Large lexical database of English words
- Organizes words into sets of synonyms called synsets
- Word relationships captured through hypernyms and hyponyms
- Only contains “open-class words”: nouns, verbs, adjectives, and adverbs.
- (Doesn’t have determiners, prepositions, pronouns, conjunctions, and particles)
- WordNets exist for many languages (we’ll just look at English) <http://globalwordnet.org/resources/wordnets-in-the-world/>



## Grabbing dependencies

```
#!/usr/bin/env python3
import nltk
nltk.download('wordnet')
nltk.download('omw-1.4')
```

## Wordnet synsets

```
#!/usr/bin/env python3

from nltk.corpus import wordnet as wn
synsets = wn.synsets('car')
print(synsets)
print(synsets[0].definition())
print(synsets[0].examples())
print(synsets[0].lemmas())
print(synsets[0].lemmas(lang='spa'))

[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'), Synset('cable_car.n.01')]
a motor vehicle with four wheels; usually propelled by an internal combustion engine
['he needs a car to get to work']
[Lemma('car.n.01.car'), Lemma('car.n.01.auto'), Lemma('car.n.01.automobile'), Lemma('car.n.01.machine'), Lemma('car.n.01.motorcar')]
[Lemma('car.n.01.auto'), Lemma('car.n.01.automóvil'), Lemma('car.n.01.carro'), Lemma('car.n.01.coche'), Lemma('car.n.01.máquina'), Lemma('car.n.01.turismo'), Lemma('car.n.01.vehículo')]
```

## wordnet2.py (hypernyms)

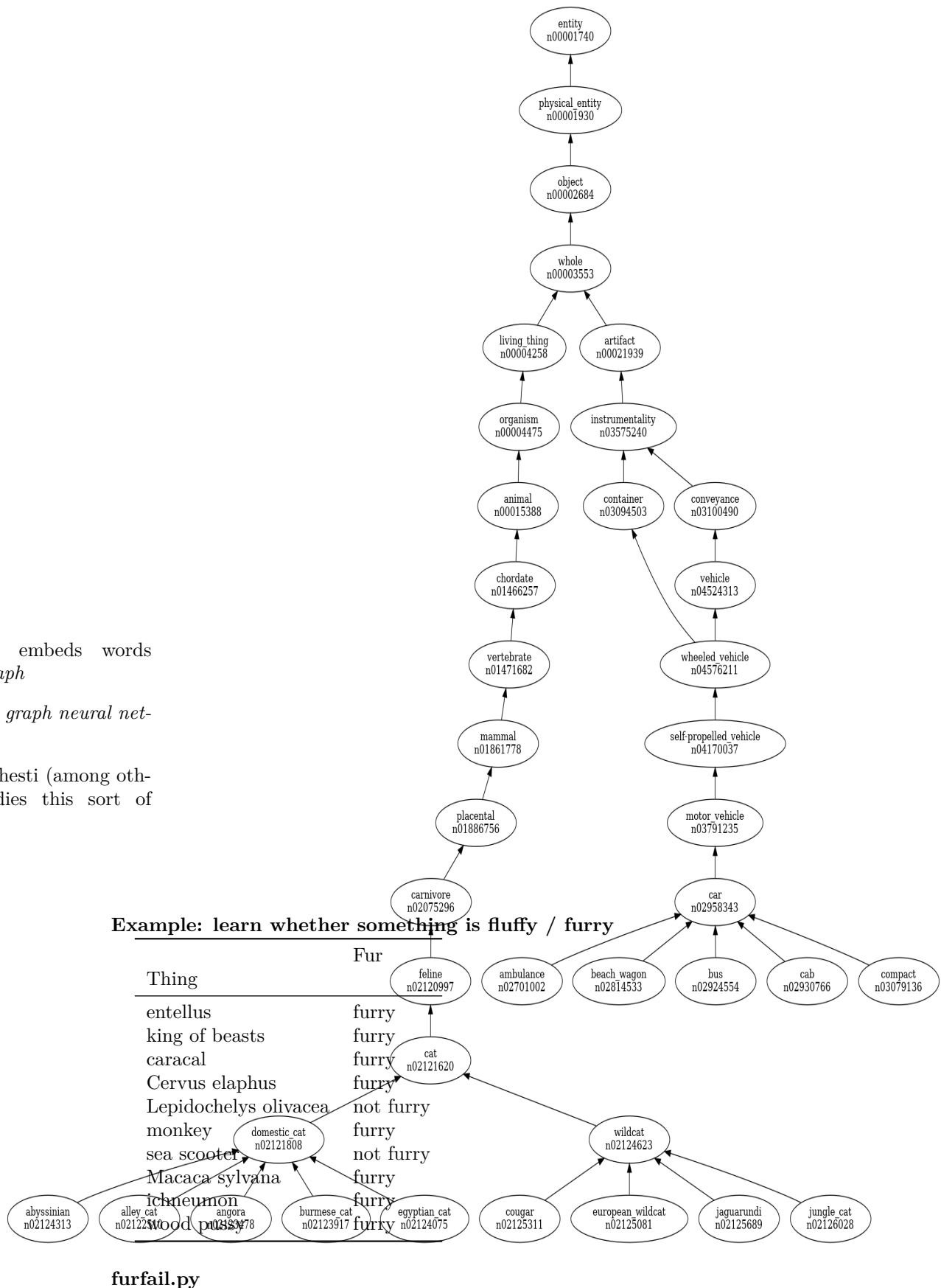
```
#!/usr/bin/env python3

from nltk.corpus import wordnet as wn
synsets = wn.synsets('car')
print(synsets[0])
print(synsets[0].hypernyms())
x = synsets[0].hypernyms()[0]
while True:
    print(x.hypernyms())
    if len(x.hypernyms()) > 0:
        x = x.hypernyms()[0]
    else:
        break

Synset('car.n.01')
[Synset('motor_vehicle.n.01')]
[Synset('self-propelled_vehicle.n.01')]
[Synset('wheeled_vehicle.n.01')]
[Synset('container.n.01'), Synset('vehicle.n.01')]
[Synset('instrumentality.n.03')]
[Synset('artifact.n.01')]
[Synset('whole.n.02')]
[Synset('object.n.01')]
[Synset('physical_entity.n.01')]
[Synset('entity.n.01')]
[]
```

## How to find hyponyms — Hearst Patterns

- WordNet embeds words into a *graph*
- Good for *graph neural networks*
- Amin Behesti (among others) studies this sort of learning



```

vectorizer = keras.layers.StringLookup()
vectorizer.adapt(train.Thing)
train_data = vectorizer(train.Thing)
validation_data = vectorizer(validation.Thing)

inputs = keras.Input(shape=(1,))
output = keras.layers.Dense(1,
    activation="sigmoid")(inputs)
model = keras.Model(inputs=[inputs], outputs=[output])
model.compile(
    loss='binary_crossentropy',
    metrics=['accuracy'])

history = model.fit(x=train_data, y=train.target,
    validation_data=(
        validation_data,
        validation.target),
    verbose=2,

```

### Data irrelevancy — it is only as good as chance

```

21/21 - 1s - loss: 14.4917 - accuracy: 0.5607 - val_loss: 0.6959 - val_accuracy: 0.4093 - 797ms/epoch - 38ms/step
21/21 - 0s - loss: 11.5815 - accuracy: 0.5607 - val_loss: 0.6979 - val_accuracy: 0.4093 - 108ms/epoch - 5ms/step
21/21 - 0s - loss: 9.1865 - accuracy: 0.5607 - val_loss: 0.7000 - val_accuracy: 0.4093 - 114ms/epoch - 5ms/step
21/21 - 0s - loss: 6.6852 - accuracy: 0.5623 - val_loss: 0.7022 - val_accuracy: 0.4093 - 105ms/epoch - 5ms/step
21/21 - 0s - loss: 4.1274 - accuracy: 0.5623 - val_loss: 0.7047 - val_accuracy: 0.4093 - 108ms/epoch - 5ms/step
21/21 - 0s - loss: 1.6590 - accuracy: 0.5561 - val_loss: 0.7069 - val_accuracy: 0.4093 - 106ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6782 - accuracy: 0.5717 - val_loss: 0.7074 - val_accuracy: 0.4093 - 113ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6855 - accuracy: 0.5530 - val_loss: 0.7074 - val_accuracy: 0.4093 - 107ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6878 - accuracy: 0.5452 - val_loss: 0.7076 - val_accuracy: 0.4093 - 99ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6843 - accuracy: 0.5623 - val_loss: 0.7076 - val_accuracy: 0.4093 - 99ms/epoch - 5ms/step

```



### fur-success.py 14–31 — Enrich with hypernym information

```

def hypernym_line(synset):
    s = set()
    hypernyms = synset.hypernyms()
    for h in hypernyms:
        s.update([h])
        parents = hypernym_line(h)
        s.update(parents)
    return s

def enrich_with_hypernyms(x):
    synsets = wn.synsets(x)
    if len(synsets) == 0: return x
    first_synset = synsets[0]
    hypernyms = hypernym_line(first_synset)
    h = " ".join([x.lemma_names()[0] for x in hypernyms])
    return x + " " + h

df['enriched'] = df.Thing.map(enrich_with_hypernyms)

train, validation = train_test_split(df)

```

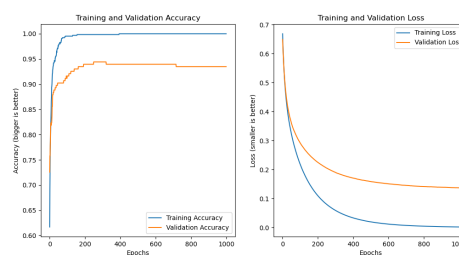


## Concept similarity is now encoded

```

21/21 - 1s - loss: 0.6685 - accuracy: 0.6168 - val_loss: 0.6489 - val_accuracy: 0.7256 - 547ms/epoch - 26ms/step
21/21 - 0s - loss: 0.6454 - accuracy: 0.6900 - val_loss: 0.6317 - val_accuracy: 0.7581 - 104ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6263 - accuracy: 0.7305 - val_loss: 0.6164 - val_accuracy: 0.7581 - 105ms/epoch - 5ms/step
21/21 - 0s - loss: 0.6093 - accuracy: 0.7664 - val_loss: 0.6019 - val_accuracy: 0.7767 - 103ms/epoch - 5ms/step
21/21 - 0s - loss: 0.5936 - accuracy: 0.7850 - val_loss: 0.5890 - val_accuracy: 0.7860 - 105ms/epoch - 5ms/step
...
21/21 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.1374 - val_accuracy: 0.9349 - 100ms/epoch - 5ms/step
21/21 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.1374 - val_accuracy: 0.9349 - 100ms/epoch - 5ms/step
21/21 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.1374 - val_accuracy: 0.9349 - 98ms/epoch - 5ms/step
21/21 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.1374 - val_accuracy: 0.9349 - 100ms/epoch - 5ms/step
21/21 - 0s - loss: 0.0023 - accuracy: 1.0000 - val_loss: 0.1374 - val_accuracy: 0.9349 - 103ms/epoch - 5ms/step

```



Wait? What?

**Me: Adds vocabulary.**

**Model works better now**

**Me:**



- Don't we need to *reduce* the dimensionality to improve a model?

### What happened

We changed  $\beta$  from 1.0 down to almost 0.0 (and increased  $k$ ).  
We made *some* vocabulary terms very effective.

$$V = kN^\beta$$

where:

- $V$  is the size of the vocabulary
- $N$  is the size of the corpus
- $k$  and  $\beta$  are constants that depend on the language and the type of text
- Usually  $.67 < \beta < .75$  (Jane Austen is verbose, so very low  $\beta$ ; Shakespeare is concise, so very high  $\beta$ ; 1.0 would be noise)

### Explainer

	0		0
mammal	9.873138	snake	-9.926085
dog	8.689872	car	-9.148499
genus	7.822046	lizard	-8.403874
whale	7.587407	buggy	-8.201899
american	7.585626	cart	-7.305783

## 3 What relu does

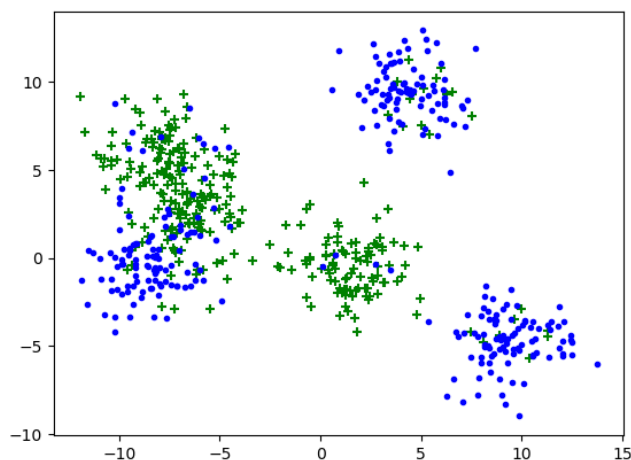
### ReLU review

ReLU = **r**ectified **l**inear **u**nit

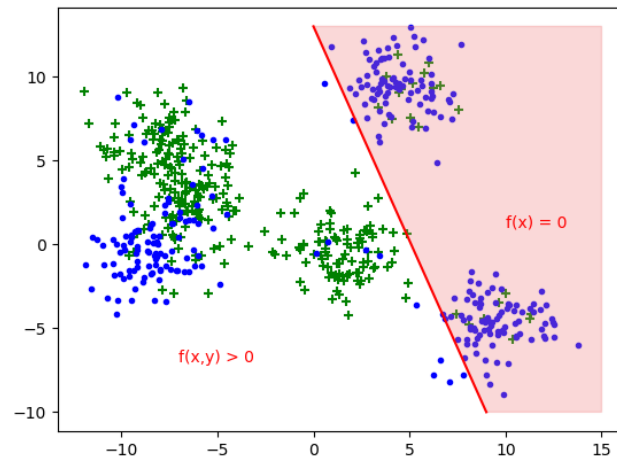
$$f(x) = \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases}$$

```
keras.layers.Dense(1, activation='relu')
```

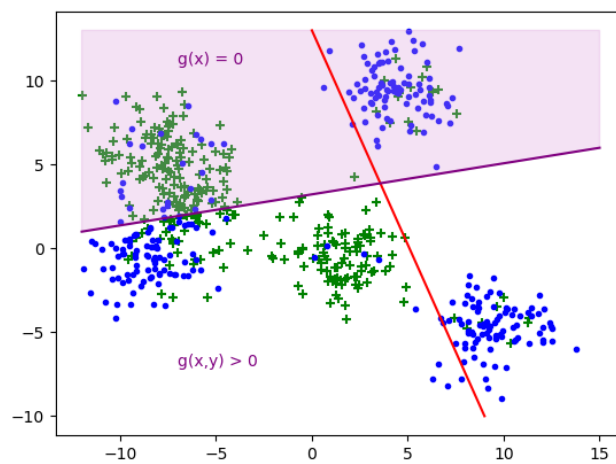
A 2D dataset (not necessarily language based)



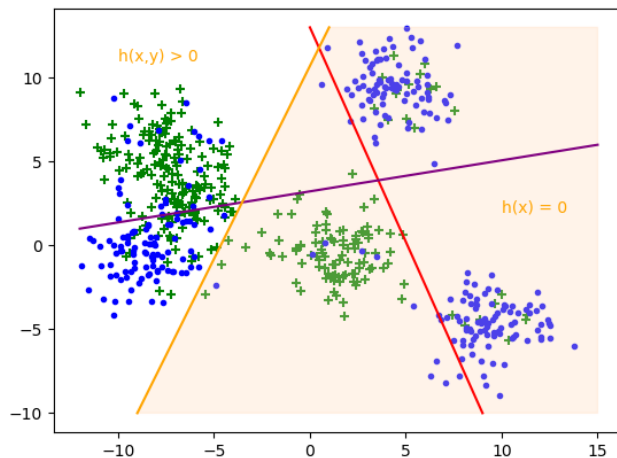
ReLU divides the data into a zero-region and a non-zero region



Adding another relu gives some more regions

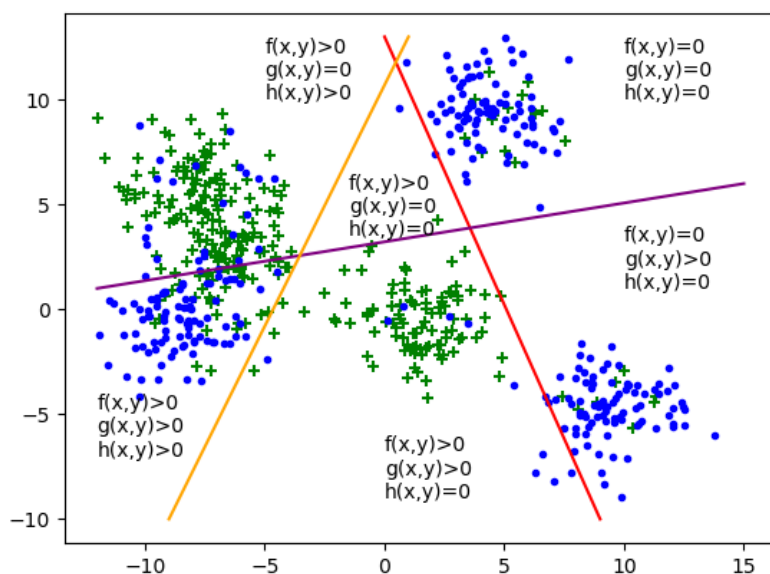


And so on with a third ReLU — not quite 8 regions



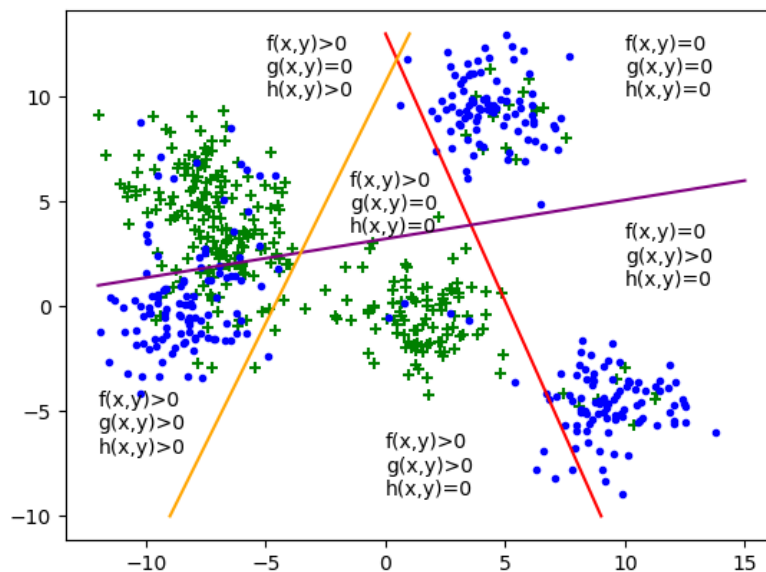
### Observations about the null region

- All the  $f(x)=g(x)=h(x)=0$  points are indistinguishable
- Unless the optimizer allows a big enough “jump” to “capture” another point, the loss function from that region is constant.
- A line could only move if it alters the loss on the *other* sides of lines.
- The derivative of the loss function near the lines in the area is 0.
- If a word vector is in the null region, often no improvement is possible.



### More observations

- We took a 2D space  $(x, y)$  and turned it into a 3D space  $(f(x, y), g(x, y), h(x, y))$
- Points where  $f(x, y) = 0$  are orthogonal to points where  $g(x, y) = h(x, y) = 0$ 
  - Many other sets of orthogonal vectors too
- With enough relu lines, we could give almost every point its own region



## 4 Word Embeddings

### Key ideas behind word embeddings

- **The Curse of Dimensionality** means that the dot product of two vectors with random values is going to be close to zero
- We can assign a random vector to each word
- Adjust the vectors to make similar words have a dot product bigger than zero
- Any way of adjusting those vectors is reasonable

### One-hot vs. word embeddings

#### One-hot

- Sparse
- Binary values (typically)

- High-dimensional
- Hard-coded

### Word embeddings

- Dense
- Continuous values
- Lower-dimensional
- Learned from data

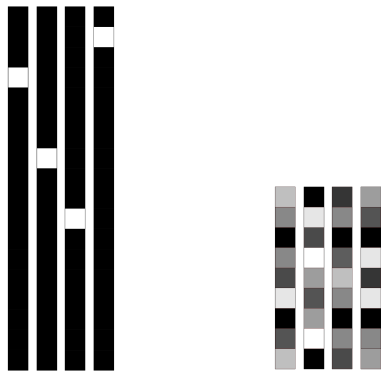
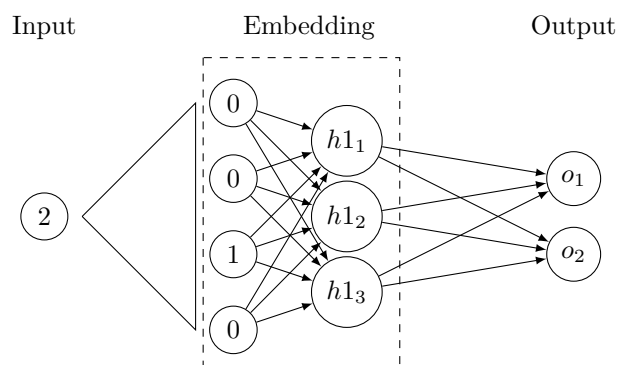


Image from Chollet (2018) "Deep Learning with Python:", Manning. Figure 6.2, page 184.

### Embedding Layer in Keras

The input of a Keras embedding layer is a sequence of word indices which will be internally converted into their one-hot representations.

#### A Keras Embedding Layer (for one word)



## Processing Sequences of Words in Keras

- The input of a Keras embedding layers is a sequence of words. It might be one or it might be a million words.
- The output is a sequence of word embeddings, perhaps 300 dimensions each.
- So the next layer up may have to deal with input that is shaped  $1 \times 300$  or  $100000 \times 300$ . This causes a problem. Here are potential solutions:
  - Choose a problem where you always have the same number of words in each document, like the example we'll do in a moment
  - Add a pooling layer (e.g. take the average) of all the words' embeddings.
  - Trim and pad sequences so that they are the same length. `keras.utils.pad_sequences` can help with this
  - Use a layer that works on sequences (we'll do this next week)

## A fun example with colours

- [https://cosmiccoding.com.au/tutorials/encoding\\_colours/](https://cosmiccoding.com.au/tutorials/encoding_colours/)
- Start with a data set of similar and dissimilar colours
- Assign a random 2D vector to each colour name. This is an **embedding** of colours into  $R^2$ . We do this using an Embedding layer.
  - Each 2D vector value is an *updateable* parameter.
  - Gradient descent will improve those vector values.
- Train a classifier to predict whether two colours are similar or not

### The data set

Colour_x	Colour_y	colours_are_similar	colour_number	Colour	hex
Chocolate	GoldOchre	1.0	313	Aquamarine4	#458
SlateBlueMedium	Gray92	0.0	119	BrownOchre	#874
SpringGreen	YellowLight	0.0	477	MediumPurple1	#ab8
Pink2	Tomato2	0.0	46	PaleGreen	#98f
GeraniumLake	GeraniumLake	1.0	478	MediumPurple2	#9f7

### colour-embedding.py 19–34

```
def euclidean_distance(tensor):
    x1, y1, x2, y2 = tensor[:, 0], tensor[:, 1], tensor[:, 2],
        tensor[:, 3]
    distance = (x1 - x2)**2 + (y1 - y2)**2
    return tf.expand_dims(distance, axis=-1)

inputs = keras.Input(shape=(2,))
embedding_layer = Embedding(
    input_dim=len(colour_lookup),
```

```

        output_dim=2,
        input_length=2
    )(inputs)
    flattening_layer = Flatten()(embedding_layer)
    distance_layer = Lambda(euclidean_distance)(flattening_layer)
    output = Dense(1, activation="sigmoid")(distance_layer)
    model = keras.Model(inputs=[inputs], outputs=[output])
    model.compile(loss='binary_crossentropy')

```

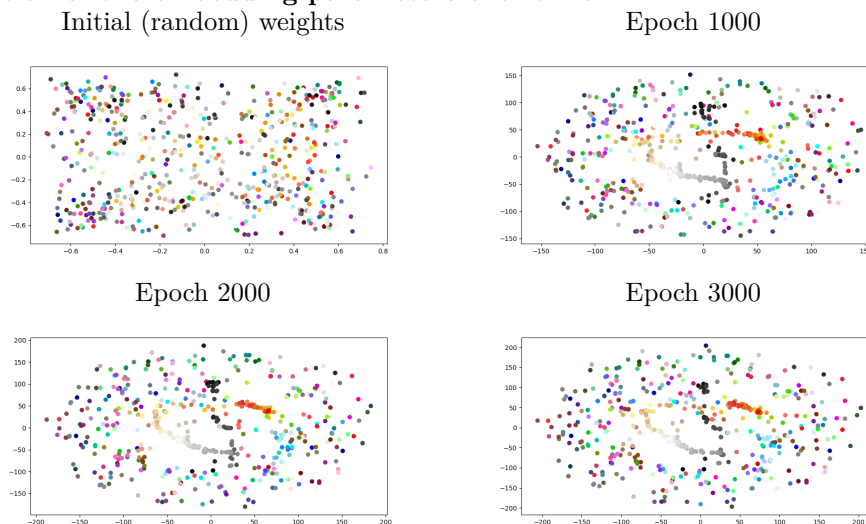
### Notes on colour-embedding.py

A *sigmoid* activation can't say whether two points are close to each other, so we need a layer that provides a distance measure.

That's the *Lambda* layer.

We didn't care about validation loss, or a training set: we just wanted the values of the embedding layer.

### Evolution of the embedding parameters over time



I made a video `colour-animation.mp4`

### Four Ways to Obtain Word Embeddings

**Manually** e.g. WordNet

**Naively** e.g. bag-of-words

**Jointly** Learn the word embeddings jointly with the task you care about (e.g. document classification) — what we just did with colours

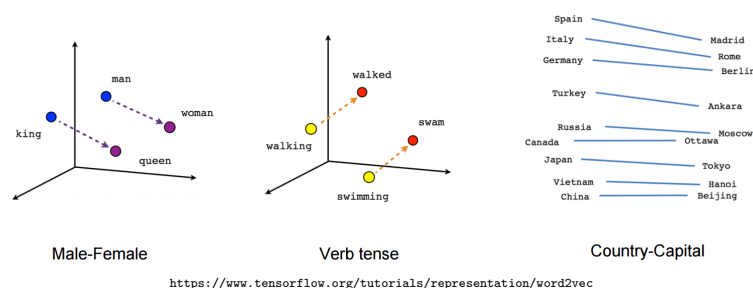
**Pre-trained** Use pre-trained word embeddings — we'll do this with Enron



## Word2vec was the first pre-trained word embedding

If you don't count manual approaches as being pre-training

- First introduced in 2013 by a team of researchers led by Tomas Mikolov at Google. Word2vec was trained on a large-scale dataset, the Google News corpus.
- Comprised of about 100 billion words from news articles.
- Resulted in a vocabulary size of around 3 million unique words.
- The pretrained word vectors provided by the authors used a 300-dimensional vector space.



## Methods

If the training set contains “Mary sat at the bank” ...

**CBOW: Continuous Bag-of-Words** CBOW attempts to predict “sat” given “Mary”, “at”, [blank] “the”, “bank”.

**Continuous skip-gram** Skip-gram attempts to predict “Mary”, “at”, “the”, “bank” given “sat”

## FastText: An Extension of Word2vec

- **Overview:** FastText is a word embedding technique and an extension of Word2vec, developed by Facebook’s AI Research (FAIR) lab in 2016.
- **Key Improvement:** Considers subword information (character-by-character sequences) to create embeddings.
- Helps solve out-of-vocabulary (OOV) problems
- Works well in morphologically-rich languages like Turkish, Arabic, Russian, Finnish (where words change form a lot)

## ELMo / BERT — Contextual Embeddings

Not a simple lookup; needs context to create embeddings. Google search uses BERT.



<http://jalammar.github.io/illustrated-bert/>

## 5 Using embeddings

### enron-with-embeddings.py 25–32

```
sequence_length = 200
vectorizer = TextVectorization(
    output_sequence_length=sequence_length,
    output_mode='int')
vectorizer.adapt(train_data.email_text)
train_vectors = vectorizer(train_data.email_text)
# Display a few training vectors
print(train_vectors[:2])
```

```
tf.Tensor(
[[ 9  48  124  6  439  215  21  89  99  165 1040 4635
 117 1040 4635 117  0  0  0  0  0  0  0 ...
...
 0  0  0  0  0  0  0  0  0]
[ 9  76 4651 321 123  15  2 162 834  7  2 7987
 87  29 443  19  29  38 11  2 407 942  6  76
100  29  14 6195 1665 781  7 195  29  82 443  5
...
1233 36061 16228  446 7131 6389  62 36100])
```

### enron-with-embeddings.py 37–41

```
vocab_size = vectorizer.vocabulary_size()
inputs = keras.Input(shape=(sequence_length,))
embedding_layer = keras.layers.Embedding(
    input_dim=sequence_length,
    output_dim=16)(inputs)
```

Embed the first `sequence_length` (200) words of each Enron email as a vector with 16 dimensions

### enron-with-embeddings.py 42–47

```
averaging_layer =
    keras.layers.GlobalAveragePooling1D()(embedding_layer)
thinking_layer = keras.layers.Dense(8,
    activation="relu")(averaging_layer)
output = Dense(1, activation="sigmoid")(thinking_layer)
model = keras.Model(
    inputs=[inputs],
    outputs=[output])
```

**AveragePooling:** If there's a significant word, it will have a strong signal in some direction (in 16 dimensional space), and even averaged over 200 words we can still slice it apart from the rest of the data

8 `thinking_layer` neurons can slice the space into hundreds of regions.

## Model summary

Model: "model"

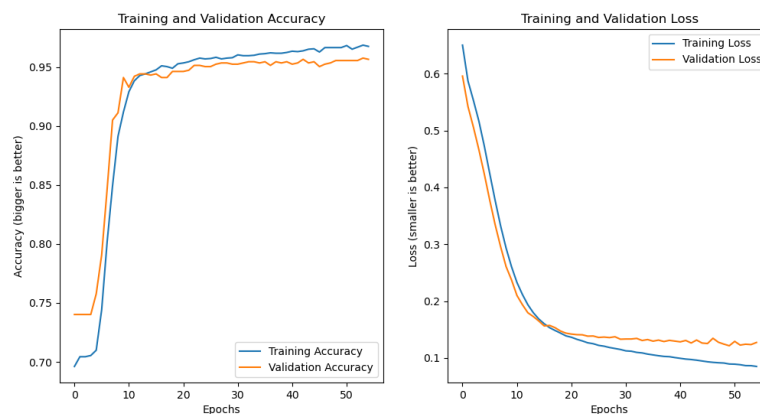
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 200)]	0
embedding (Embedding)	(None, 200, 16)	3200
global_average_pooling1d (G	(None, 16)	0
lobalAveragePooling1D)		
dense (Dense)	(None, 8)	136
dense_1 (Dense)	(None, 1)	9

Total params: 3,345

Trainable params: 3,345

Non-trainable params: 0

## Slightly disappointing results



## GloVe embeddings

- GloVe takes into account both local and global context in its training process
- Sometimes leads to more accurate word representations
- GloVe's matrix factorization-based approach can handle sparse data better than Word2Vec
- Sometimes works better on smaller corpora.

In reality:

- On small problems, with small documents bag-of-words wins

- On larger documents word2vec and GloVe are about equivalent
- On large documents BERT does better than either, but embedding takes much longer

#### enron-pretrained.py 35–40 Read in the GloVe vectors

```
embeddings_index = {}
with open('glove.6B.50d.txt') as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

the 0.418000 0.24968 -0.41242 ... -0.78581
, 0.013441 0.23682 -0.16899 ... 0.30392
. 0.151640 0.30177 -0.16763 ... 0.10216
of 0.708530 0.57088 -0.4716 ... -0.80375
to 0.680470 -0.039263 0.30186 ... -0.26044
```

#### enron-pretrained.py 42–48 Match GloVe up with our vocabulary

```
voc = vectorizer.get_vocabulary()
word_index = dict(zip(voc, range(len(voc))))
embedding_matrix = np.zeros((len(voc)+2, 50))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

- There are two extra words of vocabulary: “padding” and “OOV”, hence `len(voc)+2`
- Words not found in embedding index will be all-zeros.
- This includes the representation for “padding” and “OOV”

#### enron-pretrained.py 51–61 Use that data in our embedding layer

```
from keras.initializers import Constant
embedding_layer = keras.layers.Embedding(
    input_dim=len(voc)+2,
    output_dim=50,
    embeddings_initializer=Constant(embedding_matrix),
    trainable=False)

vocab_size = vectorizer.vocabulary_size()
inputs = keras.Input(shape=(sequence_length,))
embedding = embedding_layer(inputs)
averaging_layer = keras.layers.GlobalAveragePooling1D()(embedding)
```

- Everything afterwards is the same as the for `enron-with-embeddings.py`
- But far fewer modifiable parameters, so trains faster.

## Model summary

Model: "model"

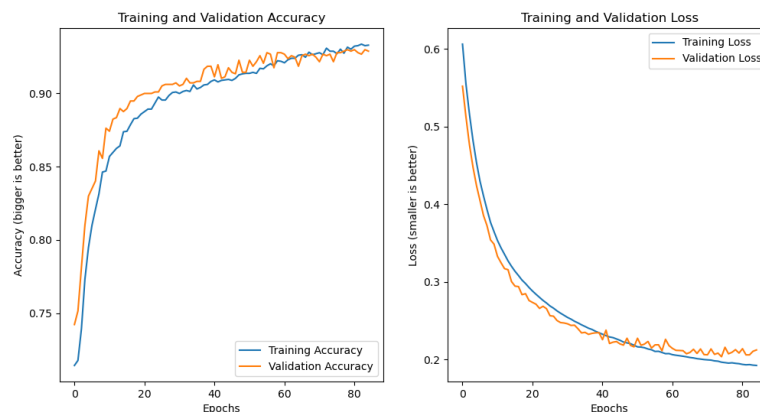
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 200)]	0
embedding (Embedding)	(None, 200, 50)	1888600
global_average_pooling1d (GlobalAveragePooling1D)	(None, 50)	0
dense (Dense)	(None, 8)	408
dense_1 (Dense)	(None, 1)	9

Total params: 1,889,017

Trainable params: 417

Non-trainable params: 1,888,600

## Even more disappointing results



## 6 Drift

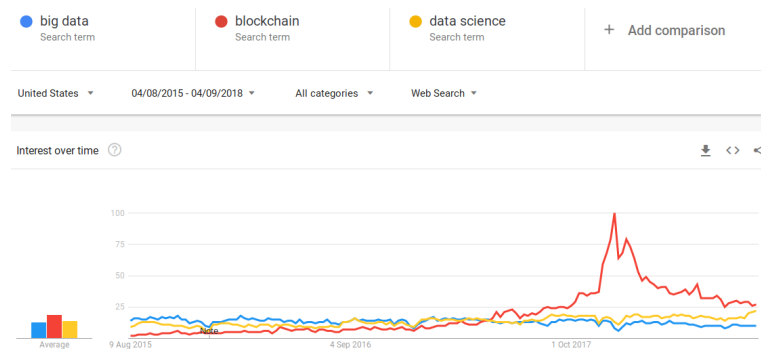
We use “Context” in different ways

The words surrounding a target word give meaning to a word ELMo, BERT are contextual embeddings.

The universe around your language model Context drift makes your model worse over time

So many words!

- Any language features a large number of distinct words.
- New words are coined.
- Words change their use in time.
- There are also names, numbers, dates ...an infinite number.



<https://trends.google.com>

### Context drift

- You train a language model today
- In the future there will be more and more new words
- Existing vocabulary will be used less, and get used differently
- The task may change: should a very relevant GPT-generated email be labelled as “spam”?
- **Your model will perform worse over time**

## 7 Wrap-up

### Take-home Messages 1

We have gone in-depth into three different word embedding approaches:

**Bag-of-words** Non-contextual: Homographs and synonyms handled poorly, loss of word order information

**WordNet** Pseudo-contextual: you have to work out which synset is right for the context; homographs and synonyms handled well; hard to use in deep learning

**Vector embeddings** A word is turned into a vector

### Take-home Messages 2

There are many ways of generating vector embeddings. We have worked with two of them, and mentioned a third

**Jointly with the task** Let gradient descent find the correct answer

**Word2vec / Glove** Non-contextual, homographs and synonyms handled poorly

**ELMo / BERT** Contextual, generating an embedding based on surrounding words

Any method that puts synonyms close to each other and keeps non-synonyms apart is good.

### Take-home Messages 3

- Looked at a geometric interpretation of ReLU activation
- Context drift is a major challenge for text classification problems

### Not-in-the-exam research idea: James-Stein estimator

- Let's say that there is a “best” vector for some word (contextual, non-contextual, whatever)
- Gradient descent methods for finding that vector is like an experiment with some noise.
- We assume that the result we get from the experiment is the best estimator for the real value.
- Unless we are embedding in one or two dimensions, that last dot point isn't true. The James-Stein estimator gives a better result.

### What's Next

#### Week 11

- Processing text sequences.
- Reading: Chollet, Section 11.4
- Reading: Jurafsky & Martin, Chapters 9 and 10.