Entity Embedding

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub

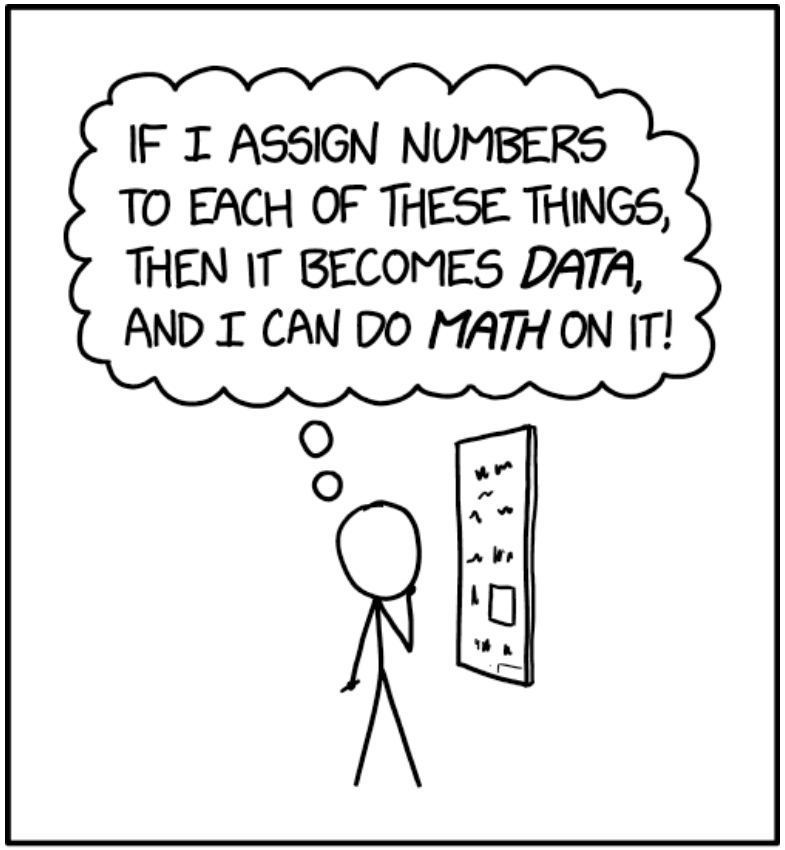
Lecture Outline

- **Word Embeddings**
- Word Embeddings II
- Car Crash NLP Part II
- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure

Overview

Popular methods for converting text into numbers include:

- One-hot encoding
- Bag of words
- TF-IDF
- Word vectors (*transfer learning*)

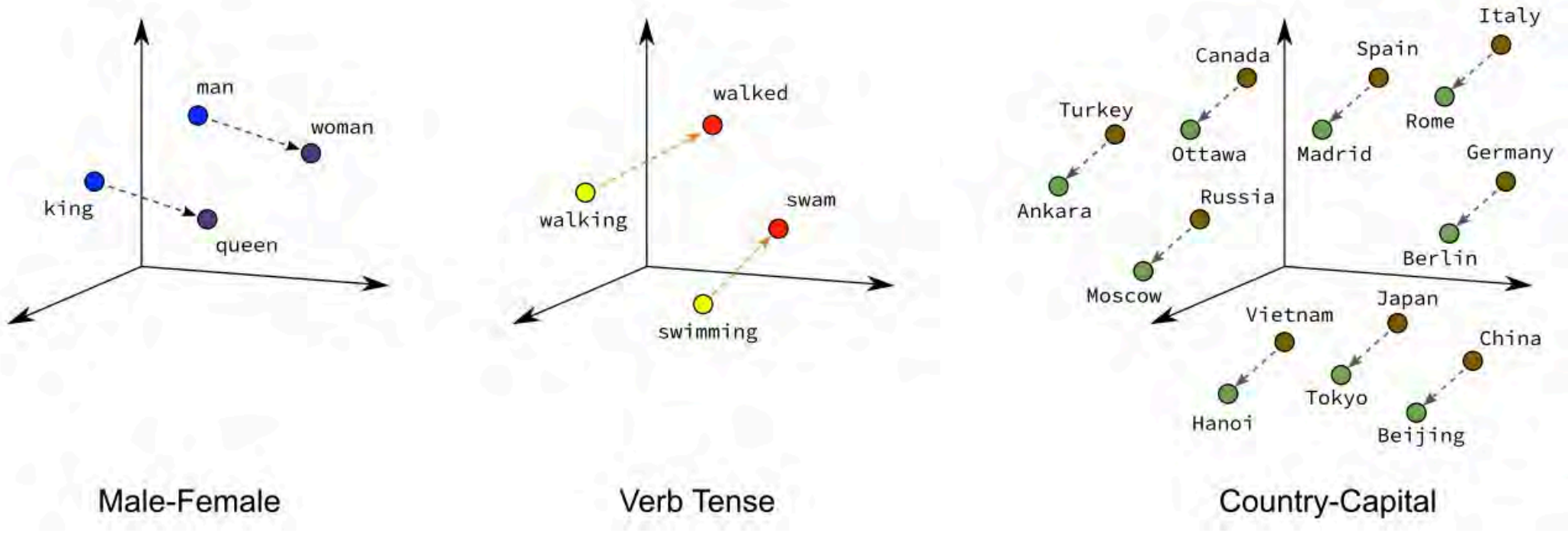


Source: Randall Munroe (2022), xkcd #2610: Assigning Numbers.

Word Vectors

- One-hot representations capture word ‘existence’ only, whereas word vectors capture information about word meaning as well as location.
- This enables deep learning NLP models to automatically learn linguistic features.
- **Word2Vec** & **GloVe** are popular algorithms for generating word embeddings (i.e. word vectors).

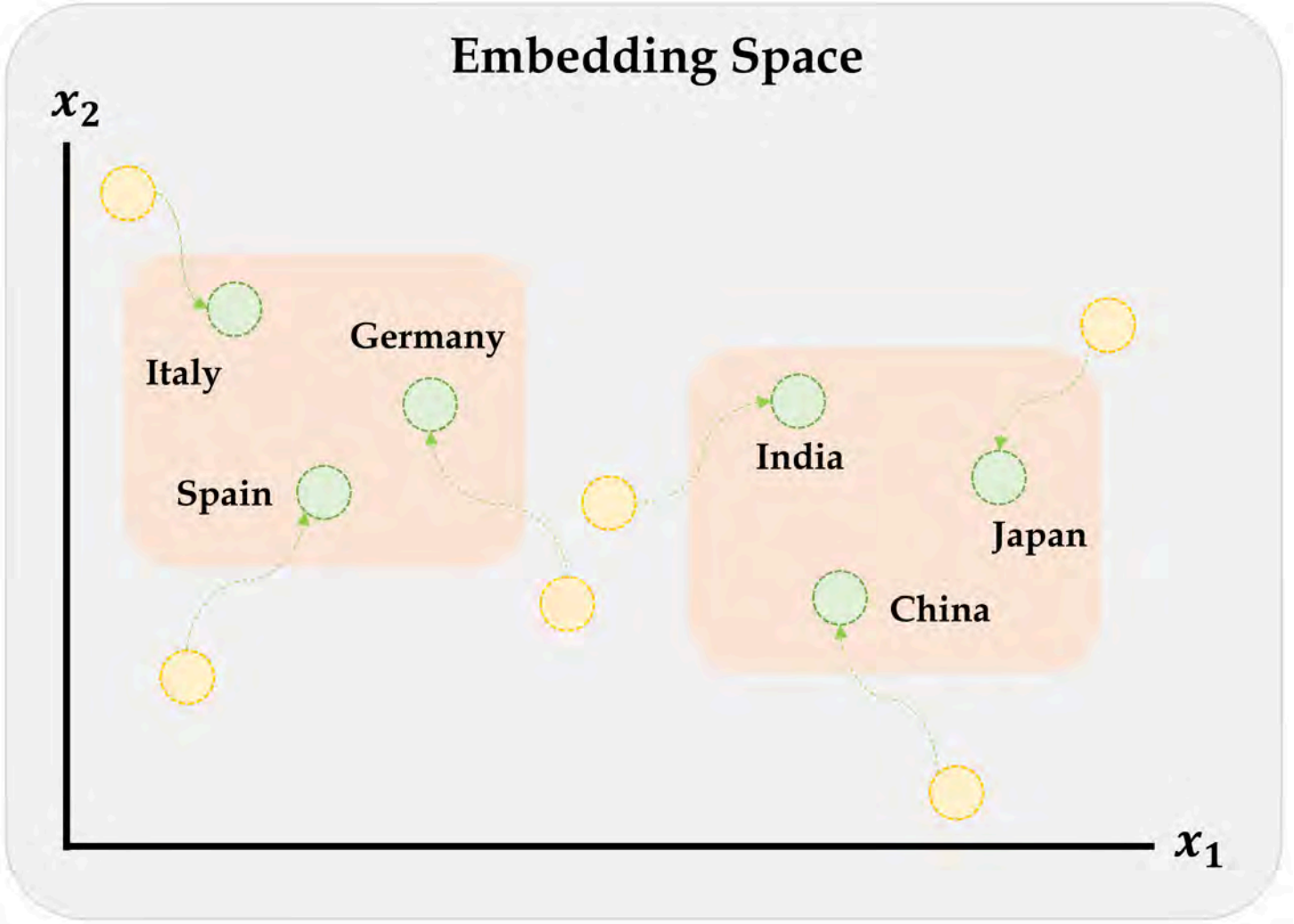
Word Vectors II



Illustrative word vectors.

Bogacz (2023), *Text Vectorization, an Introduction*, The Information Lab.

Remember this diagram?



Embeddings will gradually improve during training.

Source: Marcus Lautier (2022) !!!CHECK WITH PL

Word2Vec

Key idea: You're known by the company you keep.

Two algorithms are used to calculate embeddings:

- *Continuous bag of words*: uses the context words to predict the target word
- *Skip-gram*: uses the target word to predict the context words

Predictions are made using a neural network with one hidden layer. Through backpropagation, we update a set of "weights" which become the word vectors.

Paper: Mikolov et al. (2013), *Efficient estimation of word representations in vector space*, arXiv:1301.3781.

Word2Vec training methods

Continuous bag of words is a *centre word prediction* task

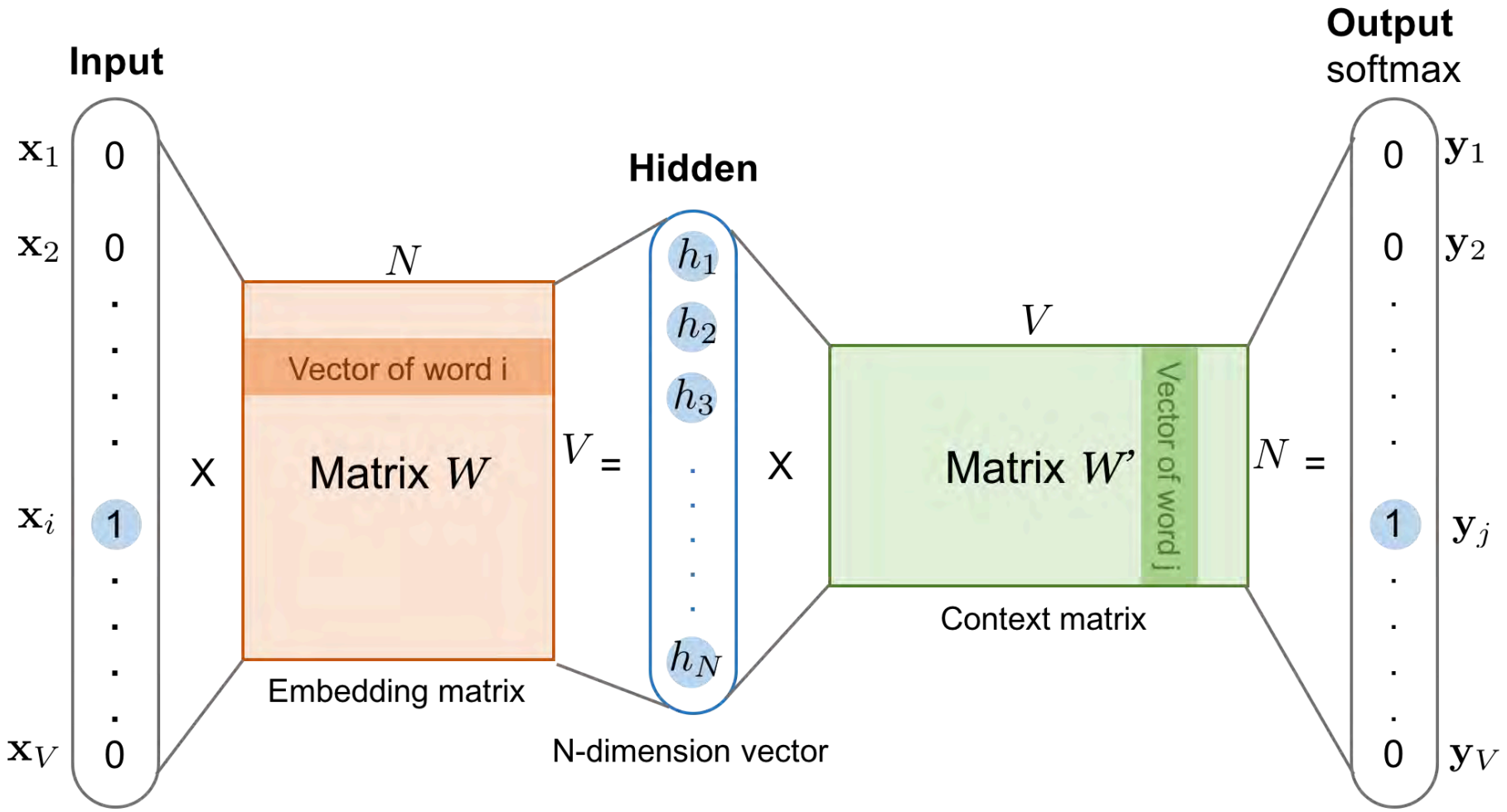
Skip-gram is a *neighbour word prediction* task

Suggested viewing

Computerphile (2019), Vectoring Words (Word Embeddings), YouTube (16 mins).

Source: Amit Chaudhary (2020), Self Supervised Representation Learning in NLP.

The skip-gram network

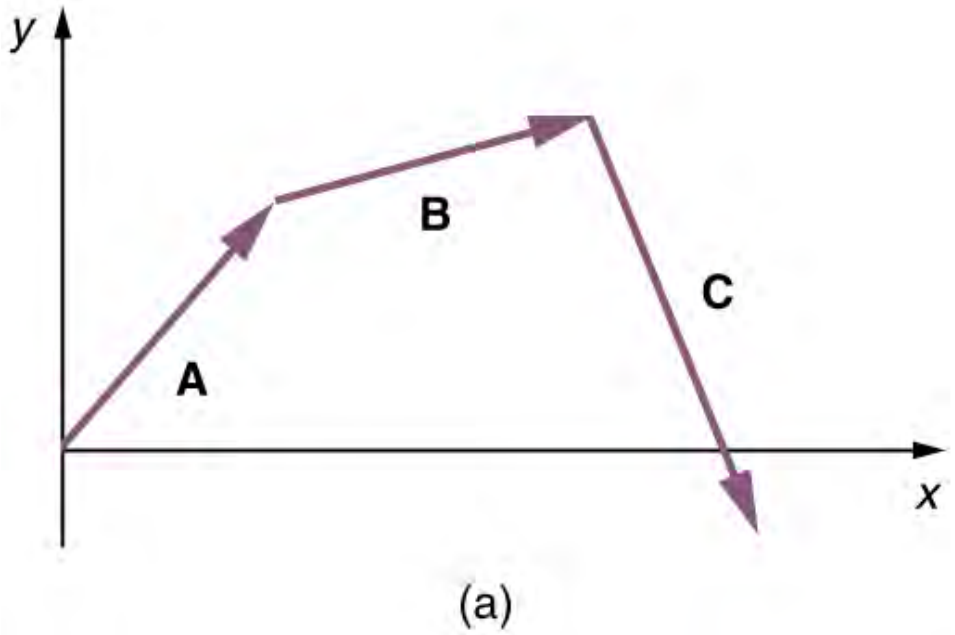


The skip-gram model. Both the input vector $\boldsymbol{x}$ and the output $\boldsymbol{y}$ are one-hot encoded word representations. The hidden layer is the word embedding of size N.

Source: Lilian Weng (2017), Learning Word Embedding, Blog post, Figure 1.

Word Vector Arithmetic

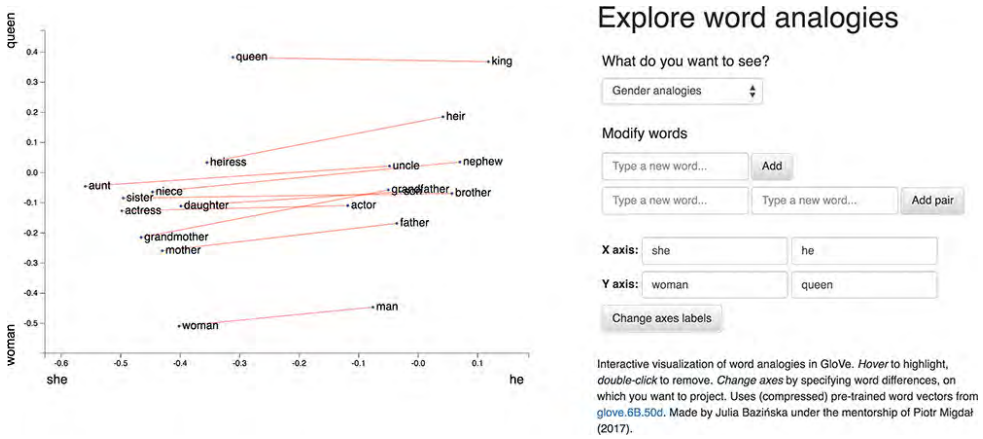
Relationships between words becomes vector math.



You remember vectors, right?

$$v_{king} - v_{man} + v_{woman} = v_{queen}$$
$$v_{bezos} - v_{amazon} + v_{tesla} = v_{musk}$$
$$v_{windows} - v_{microsoft} + v_{google} = v_{android}$$

Illustrative word vector arithmetic



Screenshot from Word2viz

Sources: PressBooks, College Physics: OpenStax, Chapter 17 Figure 9, and Krohn (2019), *Deep Learning Illustrated*, Figures 2-7 & 2-8.

UNSW SYDNEY

Lecture Outline

- Word Embeddings
- **Word Embeddings II**
- Car Crash NLP Part II
- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure

Pretrained word embeddings

GloVe (**Gl**obal **Ve**ctors) are pre-trained word embeddings from Stanford, trained on Wikipedia + Gigaword (6 billion tokens, ~400,000 word vocabulary).

```
1  f"The size of the vocabulary is {len(words)}"
```

```
'The size of the vocabulary is 400000'
```

Look up a word vector

```
1 vec("pizza")
```

```
array([ 0.04,  0.07,  0.06, -0.  , -0.03,  0.03, -0.01, -0.04, -0.07,
        0.01, -0.04, -0.1 , -0.03,  0.09, -0.05,  0.  , -0.02, -0.02,
       -0.05,  0.06,  0.05,  0.11, -0.01, -0.02,  0.06, -0.01,  0.04,
        0.  ,  0.06, -0.18, -0.08,  0.07,  0.05, -0.04, -0.08,  0.05,
       -0.06, -0.05, -0.07,  0.04,  0.02,  0.01,  0.  ,  0.08, -0.02,
        0.05,  0.15,  0.02,  0.01, -0.03, -0.01, -0.1 ,  0.05,  0.08,
       -0.03, -0.04, -0.01,  0.05,  0.02, -0.04,  0.12, -0.04, -0.  ,
       -0.07, -0.02, -0.05, -0.03,  0.07, -0.04,  0.04,  0.08,  0.02,
       -0.01, -0.06,  0.02, -0.03, -0.02, -0.01, -0.01, -0.05, -0.02,
        0.06,  0.01, -0.06, -0.02, -0.07,  0.04,  0.04, -0.05,  0.01,
        0.04,  0.02, -0.02, -0.02, -0.  ,  0.01, -0.04,  0.03, -0.06,
        0.01,  0.05, -0.01,  0.  , -0.07, -0.01, -0.06,  0.02,  0.11,
       -0.03,  0.02,  0.05,  0.  ,  0.04, -0.09,  0.06, -0.09, -0.09,
        0.04, -0.05, -0.01, -0.03,  0.02,  0.12, -0.02, -0.04,  0.03,
        0.01,  0.02, -0.05, -0.02,  0.01,  0.06, -0.03, -0.  ,  0.03,
       -0.03,  0.  ,  0.03, -0.02,  0.06,  0.02,  0.01, -0.04, -0.06,
```

```
1 len(vec("pizza"))
```

```
300
```

UNSW SYDNEY

Find nearby word vectors

```
1 nearest(vec("python"))
```

```
[('python', 1.0),
 ('monty', 0.6837381720542908),
 ('perl', 0.519283652305603),
 ('cleese', 0.5092198848724365),
 ('pythons', 0.5007115006446838),
 ('php', 0.49423137307167053),
 ('grail', 0.4683017432689667),
 ('scripting', 0.467612624168396),
 ('skit', 0.4474538564682007),
 ('javascript', 0.4312553107738495)]
```

```
1 similarity("python", "java")
```

```
0.35804617404937744
```

```
1 similarity("python", "sport")
```

```
0.005185101181268692
```

```
1 similarity("python", "r")
```

```
0.060789704322814 94
```

What does ‘similarity’ mean?

The ‘similarity’ scores

```
1 similarity("sydney", "melbourne")
```

```
0.7951619625091553
```

are normally based on cosine distance.

```
1 x = vec("sydney")
2 y = vec("melbourne")
3 x.dot(y) / (np.linalg.norm(x) * np.linalg.norm(y))
```

```
np.float32(0.795162)
```

```
1 similarity("sydney", "aarhus")
```

```
0.14399726688861847
```

Weng's GoT Word2Vec

In the Game of Thrones (GoT) word embedding space, the top similar words to "king" and "queen" are:

```
1 model.most_similar("king")
```

```
('kings', 0.897245)
('baratheon', 0.809675)
('son', 0.763614)
('robert', 0.708522)
('lords', 0.698684)
('joffrey', 0.696455)
('prince', 0.695699)
('brother', 0.685239)
('aerys', 0.684527)
('stannis', 0.682932)
```

```
1 model.most_similar("queen")
```

```
('cersei', 0.942618)
('joffrey', 0.933756)
('margaery', 0.931099)
('sister', 0.928902)
('prince', 0.927364)
('uncle', 0.922507)
('varys', 0.918421)
('ned', 0.917492)
('melisandre', 0.915403)
('robb', 0.915272)
```

Source: Lilian Weng (2017), Learning Word Embedding, Blog post.

Combining word vectors

You can summarise a sentence by averaging the individual word vectors.

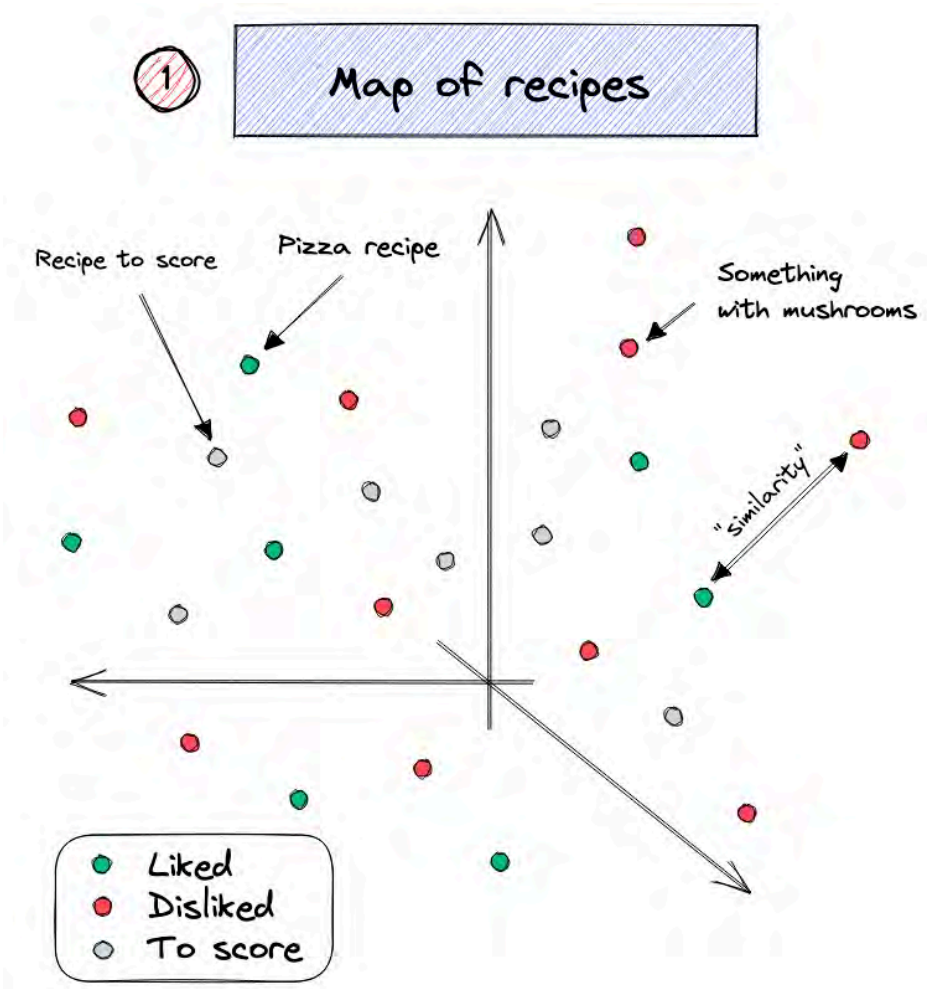
```
1 sv = (vec("melbourne") + vec("has") + vec("better") + vec("coffee")) / 4
2 len(sv), sv[:5]
```

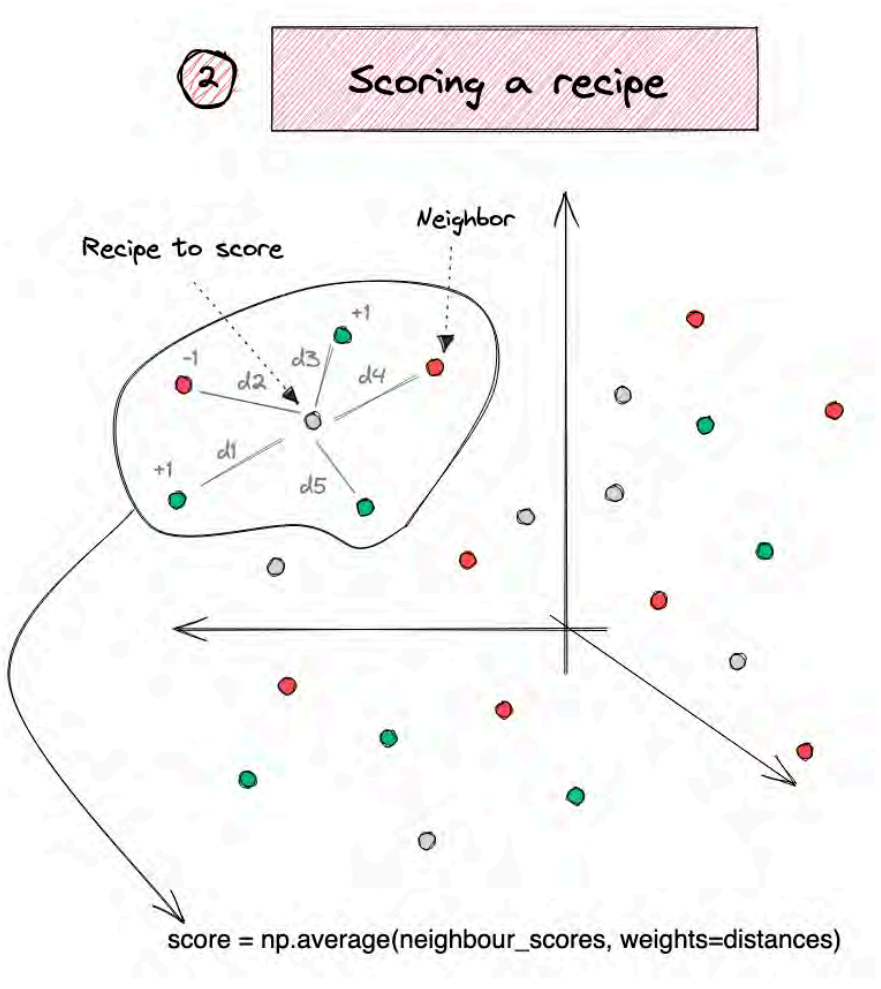
```
(300, array([-0.03,  0.03,  0.01,  0.01, -0.01], dtype=float32))
```

> As it turns out, averaging word embeddings is a surprisingly effective way to create word embeddings. It's not perfect (as you'll see), but it does a strong job of capturing what you might perceive to be complex relationships between words.

Source: Trask (2019), Grokking Deep Learning, Chapter 12.

Recipe recommender





Recipes are the average of the word vectors of the ingredients.

Nearest neighbours used to classify new recipes as potentially delicious.

Source: Duarte O.Carmo (2022), A recipe recommendation system, Blog post.

Analogies with word vectors

Obama is to America as ___ is to Australia.

$$\text{Obama} - \text{America} + \text{Australia} = ?$$

```
1 analogy("america", "obama", "australia")
```

```
[('barack', 0.5462056994438171),
 ('canberra', 0.480049580335617O7),
 ('australian', 0.4731597900390625),
 ('rudd', 0.46835529804229736),
 ('bush', 0.431667685508728),
 ('mccain', 0.42993825674057007),
 ('gillard', 0.42073196172714233),
 ('australians', 0.4089258909225464),
 ('sydney', 0.40701723098754883),
 ('zealand', 0.3963095545768738)]
```

Testing more associations

```
1 analogy("paris", "france", "london")
```

```
[('britain', 0.7673852443695068),
 ('england', 0.62794429063797),
 ('uk', 0.6197735071182251),
 ('british', 0.6013830900192261),
 ('ireland', 0.5365166664123535),
 ('u.k.', 0.5294837355613708),
 ('scotland', 0.5195812582969666),
 ('australia', 0.5150107145309448),
 ('wales', 0.5144591927528381),
 ('europe', 0.5047693848609924)]
```

Quickly get to bad associations

```
1 analogy("man", "king", "woman")
```

```
[('queen', 0.6713277101516724),
 ('princess', 0.5432624816894531),
 ('throne', 0.538610577583313),
 ('monarch', 0.5347574949264526),
 ('daughter', 0.4980251491069793 7),
 ('mother', 0.49564433097839355),
 ('elizabeth', 0.4832652807235718),
 ('kingdom', 0.47747093439102173),
 ('prince', 0.4668240547180176),
 ('wife', 0.4647327661514282)]
```

```
1 analogy("man", "programmer", "woman")
```

```
[('programmers', 0.4976009428501129),
 ('freelance', 0.41725867986679077),
 ('educator', 0.40316858887672424),
 ('businesswoman', 0.3929097056388855),
 ('designer', 0.39289426803588867),
 ('translator', 0.38584306836128235),
 ('technician', 0.3751075267791748),
 ('computer', 0.3749142289161682),
 ('animator', 0.3677002191543579),
 ('homemaker', 0.3675471544265747)]
```

Bias in NLP models

The Verge (2016), Twitter taught Microsoft's AI chatbot to be a racist a****** in less than a day.

> ... there are serious questions to answer, like how are we going to teach AI using public data without incorporating the worst traits of humanity? If we create bots that mirror their users, do we care if their users are human trash? There are plenty of examples of technology embodying — either accidentally or on purpose — the prejudices of society, and Tay's adventures on Twitter show that even big corporations like Microsoft forget to take any preventative measures against these problems.

The `analogy` function cheats a little bit

```
nearest(vec("programmer") - vec("man") + vec("woman"))
```

```
[('programmer', 0.7800661325454712),
 ('programmers', 0.4976009428501129),
 ('freelance', 0.41725867986679077),
 ('educator', 0.40316858887672424),
 ('businesswoman', 0.3929097056388855),
 ('designer', 0.39289426803588867),
 ('translator', 0.38584306836128235),
 ('technician', 0.3751075267791748),
 ('computer', 0.3749142289161682),
 ('animator', 0.3677002191543579)]
```

To get the 'nice' analogies, `analogy` ignores the input words as possible answers.

```
# ignore (don't return) keys from the input
if w not in exclude:
    results.append((w, float(sims[i])))
```

Lecture Outline

- Word Embeddings
- Word Embeddings II
- **Car Crash NLP Part II**
- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure

Dataset source: Dr Jürg Schelldorfer's GitHub.

Predict injury severity

```
features = df["SUMMARY_EN"]
target = LabelEncoder().fit_transform(df["INJSEVB"])

X_main, X_test, y_main, y_test = \
    train_test_split(features, target, test_size=0.2, random_state=1)
X_train, X_val, y_train, y_val = \
    train_test_split(X_main, y_main, test_size=0.25, random_state=1)
X_train.shape, X_val.shape, X_test.shape
```

```
((4169,), (1390,), (1390,))
```

Using TF-IDF Vectorization

```python
from sklearn.feature_extraction.text import TfidfVectorizer

max_tokens = 1_000
vect = TfidfVectorizer(
    max_features=max_tokens,
    lowercase=True,
    token_pattern=r"(?u)\b\w+\b",  # Similar to "lower_and_strip_punctuation"
)

X_train_txt = vect.fit_transform(X_train).toarray()
X_val_txt = vect.transform(X_val).toarray()
X_test_txt = vect.transform(X_test).toarray()

vocab = vect.get_feature_names_out()
print(list(vocab[:50]))
```

```
['0', '1', '10', '105', '12', '15', '150', '16', '17', '18', '19', '1990', '1991', '1992',
'1993', '1994', '1995', '1996', '1997', '1998', '1999', '2', '20', '2000', '2001', '2002',
'2003', '2004', '2005', '2006', '2007', '21', '22', '23', '24', '25', '26', '27', '28', '29',
'3', '30', '30mph', '31', '32', '33', '34', '35', '35mph', '36']
```

The TF-IDF vectors

```
1 pd.DataFrame(X_train_txt, columns=vocab, index=X_train.index)
```

	0	1	10	105	12	15	150	16	17	18	...	worked	working	w
2532	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0
6209	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
206	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0
6356	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0

4169 rows × 1000 columns

Feed TF-IDF into an ANN

```
random.seed(42)
tfidf_model = keras.models.Sequential([
    layers.Input((X_train_txt.shape[1],)),
    layers.Dense(250, "relu"),
    layers.Dense(1, "sigmoid")
])

tfidf_model.compile("adam", "binary_crossentropy", metrics=["accuracy"])
tfidf_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 250)	250,250
dense_1 (Dense)	(None, 1)	251

Total params: 250,501 (978.52 KB)
Trainable params: 250,501 (978.52 KB)
Non-trainable params: 0 (0.00 B)

Fit & evaluate

```
es = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True,
    monitor="val_accuracy", verbose=2)

if not Path("models/tfidf-model.keras").exists():
    tfidf_model.fit(X_train_txt, y_train, epochs=1_000, callbacks=[es],
        validation_data=(X_val_txt, y_val), verbose=0)
    tfidf_model.save("models/tfidf-model.keras")
else:
    tfidf_model = keras.models.load_model("models/tfidf-model.keras")
```

```
tfidf_model.evaluate(X_train_txt, y_train, verbose=0, batch_size=1_000)
```

```
[0.131618469953537, 0.9589829444885254]
```

```
tfidf_model.evaluate(X_val_txt, y_val, verbose=0, batch_size=1_000)
```

```
[0.26651373505592346, 0.8949640393257141]
```

Keep text as sequence of tokens

```python
from sklearn.feature_extraction.text import CountVectorizer
import re

max_length = 500
max_tokens = 1_000

# Build vocabulary using CountVectorizer
count_vect = CountVectorizer(max_features=max_tokens-1, lowercase=True,
    token_pattern=r"(?u)\b\w+\b")
count_vect.fit(X_train)
vocab = [""] + list(count_vect.get_feature_names_out())  # Index 0 reserved for padding
word_to_idx = {word: idx for idx, word in enumerate(vocab)}

def texts_to_sequences(texts, word_to_idx, max_length):
    sequences = []
    for text in texts:
        tokens = re.findall(r"(?u)\b\w+\b", text.lower())
        seq = [word_to_idx.get(t, 0) for t in tokens][:max_length]
        seq = seq + [0] * (max_length - len(seq))  # Pad with zeros
        sequences.append(seq)
    return np.array(sequences)

X_train_txt = texts_to_sequences(X_train, word_to_idx, max_length)
X_val_txt = texts_to_sequences(X_val, word_to_idx, max_length)
```

```
['', '0', '1', '10', '105', '12', '15', '150', '16', '17', '18', '19', '1990', '1991', '1992',
'1993', '1994', '1995', '1996', '1997', '1998', '1999', '2', '20', '2000', '2001', '2002',
```

A sequence of integers

```
1 X_train_txt[0]
```

```
array([880, 249, 619, 469, 870, 319,  96, 620,  77, 963, 469, 870, 568,
       620,  77, 392, 958, 849, 493, 870, 493, 224, 620,  77, 605, 818,
        61, 513, 924, 514, 317, 284, 191, 919, 513, 741, 491, 178, 108,
       320, 969,  61, 513, 924, 514, 317, 284, 191, 919, 513, 741, 235,
       178,  77, 691,   0, 900, 788, 870, 161, 605, 818, 741, 957, 313,
       109, 843, 984,  77, 807, 672, 809, 870, 675, 823, 957,  64, 507,
        48, 587, 870, 900, 382, 957, 604, 109, 874, 968, 601,  93, 134,
       219, 133, 870, 885, 620, 870, 249, 943,  77,  17, 179,   0,   0,
       957, 840, 133, 870, 493, 355, 818, 469, 513, 883, 954, 387, 870,
       788, 889, 193, 815, 501, 244, 919, 521, 944,  77,  24, 297,   0,
       957, 606, 469, 513, 924, 118, 870, 759, 493, 976, 501, 486, 889,
       411, 843, 975, 870, 529, 920, 420, 943, 154, 502, 521, 125, 943,
       231, 870, 919, 502, 306, 761,  77, 667, 949, 984,   0, 531,   0,
       118, 870, 493, 397, 870, 320, 943, 840, 469, 870, 568, 620, 870,
       493, 469,   0, 889, 870, 118, 667, 949, 944, 334, 870, 493, 109,
       848, 870, 398, 620, 943, 984, 502, 398, 521, 239, 166, 950, 180,
```

Feed LSTM a sequence of one-hots

```
from keras.layers import CategoryEncoding, Bidirectional, LSTM
random.seed(42)
one_hot_model = Sequential([Input(shape=(max_length,), dtype="int64"),
    CategoryEncoding(num_tokens=max_tokens, output_mode="one_hot"),
    Bidirectional(LSTM(24)),
    Dense(1, activation="sigmoid")])
one_hot_model.compile(optimizer="adam",
    loss="binary_crossentropy", metrics=["accuracy"])
one_hot_model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
category_encoding (CategoryEncoding)	(None, 500, 1000)	0
bidirectional (Bidirectional)	(None, 48)	196,800
dense_2 (Dense)	(None, 1)	49

Total params: 196,849 (768.94 KB)
Trainable params: 196,849 (768.94 KB)
Non-trainable params: 0 (0.00 B)

Fit & evaluate

```
es = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True,
    monitor="val_accuracy", verbose=2)

if not Path("models/one-hot-model.keras").exists():
    one_hot_model.fit(X_train_txt, y_train, epochs=1_000, callbacks=[es],
        validation_data=(X_val_txt, y_val), verbose=0);
    one_hot_model.save("models/one-hot-model.keras")
else:
    one_hot_model = keras.models.load_model("models/one-hot-model.keras")
```

```
one_hot_model.evaluate(X_train_txt, y_train, verbose=0, batch_size=1_000)
```

```
[0.6978467702865601, 0.6217318177223206]
```

```
one_hot_model.evaluate(X_val_txt, y_val, verbose=0, batch_size=1_000)
```

```
[0.6978550553321838, 0.6158273220062256]
```

Custom embeddings

```
from keras.layers import Embedding
embed_lstm = Sequential([Input(shape=(max_length,), dtype="int64"),
    Embedding(input_dim=max_tokens, output_dim=32, mask_zero=True),
    Bidirectional(LSTM(24)),
    Dense(1, activation="sigmoid")])
embed_lstm.compile("adam", "binary_crossentropy", metrics=["accuracy"])
embed_lstm.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 32)	32,000
bidirectional_1 (Bidirectional)	(None, 48)	10,944
dense_3 (Dense)	(None, 1)	49

Total params: 42,993 (167.94 KB)
Trainable params: 42,993 (167.94 KB)
Non-trainable params: 0 (0.00 B)

Fit & evaluate

```
es = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True,
    monitor="val_accuracy", verbose=2)

if not Path("models/embed-lstm.keras").exists():
    embed_lstm.fit(X_train_txt, y_train, epochs=1_000, callbacks=[es],
        validation_data=(X_val_txt, y_val), verbose=0);
    embed_lstm.save("models/embed-lstm.keras")
else:
    embed_lstm = keras.models.load_model("models/embed-lstm.keras")
```

```
embed_lstm.evaluate(X_train_txt, y_train, verbose=0, batch_size=1_000)
```

```
[0.7265174984931946, 0.5706404447555542]
```

```
embed_lstm.evaluate(X_val_txt, y_val, verbose=0, batch_size=1_000)
```

```
[0.7588797807693481, 0.5446043014526367]
```

```
embed_lstm.evaluate(X_test_txt, y_test, verbose=0, batch_size=1_000)
```

```
[0.7490332126617432, 0.5482014417648315]
```

Lecture Outline

- Word Embeddings
- Word Embeddings II
- Car Crash NLP Part II
- **Entity Embedding**
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure

Revisit the French motor dataset

	IDpol	ClaimNb	Exposure	Area	VehPower	VehAge	DrivAge	Bon
0	1.0	1.0	0.10000	D	5.0	0.0	55.0	50.0
1	3.0	1.0	0.77000	D	5.0	0.0	55.0	50.0
...	...	...	...	...	...	...	...	...
678011	6114329.0	0.0	0.00274	B	4.0	0.0	60.0	50.0
678012	6114330.0	0.0	0.00274	B	7.0	6.0	29.0	54.0

678013 rows × 12 columns

Source: Nell et al. (2020), Case Study: French Motor Third-Party Liability Claims, SSRN.

Data dictionary

Variable	Description	Preprocessing
`IDpol`	Policy number (unique identifier)	Dropped
`ClaimNb`	Number of claims on the given policy	**Target**
`Exposure`*	Total exposure in yearly units	Normalised
`Area`	Area code (ordinal)	Ordinal Encode
`VehPower`	Power of the car (ordinal encoded)	Normalised
`VehAge`	Age of the car in years	Normalised
`DrivAge`	Age of the (most common) driver in years	Normalised
`BonusMalus`	Bonus–malus level between 50 and 230 (with reference level 100)	Normalised
`VehBrand`*	Car brand (nominal)	One-hot
`VehGas`	Diesel or regular fuel car (binary)	One-hot
`Density`	Density of inhabitants per km² in the city of the living place of the driver	Normalised
`Region`*	Regions in France (prior to 2016)	One-hot

Source: Nell et al. (2020), Case Study: French Motor Third-Party Liability Claims, SSRN.

The model

Have $\{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,n}$ for $\mathbf{x}_i \in \mathbb{R}^{47}$ and $y_i \in \mathbb{N}_0$.

Assume the distribution

$$Y_i \sim \mathsf{Poisson}(\lambda(\mathbf{x}_i))$$

We have $\mathbb{E}Y_i = \lambda(\mathbf{x}_i)$. The NN takes $\mathbf{x}_i$ & predicts $\mathbb{E}Y_i$.

Note

For insurance, *this is a bit weird*. The exposures are different for each policy.

$\lambda(\mathbf{x}_i)$ is the expected number of claims for the duration of policy i's contract.

Normally, $\text{Exposure}_i \notin \mathbf{x}_i$, and $\lambda(\mathbf{x}_i)$ is the expected rate *per year*, then

$$Y_i \sim \mathsf{Poisson}(\text{Exposure}_i \times \lambda(\mathbf{x}_i)).$$

What values do we see in the data?

```
1 X_train_raw["Area"].value_counts()
2 X_train_raw["VehBrand"].value_counts()
3 X_train_raw["VehGas"].value_counts()
4 X_train_raw["Region"].value_counts()
```

```
Area
C    5514
D    4116
     ...
B    2387
F     444
Name: count, Length: 6, dtype: int64
```

```
VehGas
Regular    10658
Diesel      8092
Name: count, dtype: int64
```

```
VehBrand
B1     4998
B2     4906
       ...
B11     283
B14     140
Name: count, Length: 11, dtype: int64
```

```
Region
R24    6493
R82    2112
       ...
R42      48
R43      26
Name: count, Length: 22, dtype: int64
```

How we preprocessed last time

```
from sklearn.compose import make_column_transformer

ct = make_column_transformer(
  (OneHotEncoder(sparse_output=False, drop="first"), ["VehGas", "VehBrand", "Region"]),
  (OrdinalEncoder(), ["Area"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train = ct.fit_transform(X_train_raw)
```

```
X_train_raw.head(3)
```

	Exposure	Area	VehPower	VehAge
0	1.00	A	7.0	8.0
1	0.79	B	7.0	7.0
2	1.00	C	6.0	13.0

```
X_train.head(3)
```

	VehGas_Regular	VehBrand_B10	Ve
0	0.0	0.0	0.(
1	0.0	0.0	0.(
2	1.0	0.0	0.(

3 rows × 39 columns

Lecture Outline

- Word Embeddings
- Word Embeddings II
- Car Crash NLP Part II
- Entity Embedding
- **Categorical Variables & Entity Embeddings**
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure

Region column

French Administrative Regions

Source: Wikimedia

One-hot encoding

```
1 oh = OneHotEncoder(sparse_output=False)
2 X_train_oh = oh.fit_transform(X_train_raw[["Region"]])
3 X_test_oh = oh.transform(X_test_raw[["Region"]])
4 print(list(X_train_raw["Region"][:5]))
5 X_train_oh.head()
```

```
['R24', 'R21', 'R53', 'R24', 'R82']
```

	Region_R11	Region_R21	Region_R22	Region_R23	Region_R24	Region_R2
0	0.0	0.0	0.0	0.0	1.0	0.0
1	0.0	1.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...
3	0.0	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 22 columns

Train on one-hot inputs

```
num_regions = len(oh.categories_[0])

random.seed(12)
model = Sequential([
  Dense(2, input_dim=num_regions),
  Dense(1, activation="exponential")
])

model.compile(optimizer="adam", loss="poisson")

es = EarlyStopping(verbose=True)
hist = model.fit(X_train_oh, y_train, epochs=100, verbose=0,
    validation_split=0.2, callbacks=[es])
hist.history["val_loss"][-1]
```

```
Epoch 7: early stopping
```

```
0.7678699493408203
```

Make a fake batch of data

```
1 X = np.eye(num_regions)
2 pd.DataFrame(X, columns=oh.categories_[0]
```

	R11	R21	R22	R23	R24	R25	R2
0	1.0	0.0	0.0	0.0	0.0	0.0	0.(
1	0.0	1.0	0.0	0.0	0.0	0.0	0.(
...	...	...	...	...	...	...	...
20	0.0	0.0	0.0	0.0	0.0	0.0	0.(
21	0.0	0.0	0.0	0.0	0.0	0.0	0.(

22 rows × 22 columns

```
1 model.layers[0](X)
```

```
tensor([[-0.0600, -0.2381],
        [ 0.0959,  0.0169],
        [-0.3198, -0.2758],
        [ 0.0217, -0.3132],
        [ 0.3903, -0.2820],
        [ 0.0979, -0.4124],
        [ 0.2332, -0.2582],
        [ 0.1300,  0.3135],
        [ 0.6294, -0.4380],
        [ 0.1788, -0.0082],
        [-0.2035, -0.3797],
        [ 0.7685, -0.2326],
        [ 0.5165, -0.0520],
        [ 0.7843, -0.4288],
        [-0.1971, -0.1840],
        [-0.2288, -0.1071],
```

The first layer

```
1 layer = model.layers[0]
2 W, b = layer.get_weights()
3 X.shape, W.shape, b.shape
```

```
((22, 22), (22, 2), (2,))
```

```
1 X @ W + b
```

```
array([[-0.06, -0.24],
       [ 0.1 ,  0.02],
       [-0.32, -0.28],
       [ 0.02, -0.31],
       [ 0.39, -0.28],
       [ 0.1 , -0.41],
       [ 0.23, -0.26],
       [ 0.13,  0.31],
       [ 0.63, -0.44],
       [ 0.18, -0.01],
       [-0.2 , -0.38],
       [ 0.77, -0.23],
       [ 0.52, -0.05],
       [ 0.78, -0.43],
       [-0.2 , -0.18],
       [-0.23, -0.11],
```

```
1 W + b
```

```
array([[-0.06, -0.24],
       [ 0.1 ,  0.02],
       [-0.32, -0.28],
       [ 0.02, -0.31],
       [ 0.39, -0.28],
       [ 0.1 , -0.41],
       [ 0.23, -0.26],
       [ 0.13,  0.31],
       [ 0.63, -0.44],
       [ 0.18, -0.01],
       [-0.2 , -0.38],
       [ 0.77, -0.23],
       [ 0.52, -0.05],
       [ 0.78, -0.43],
       [-0.2 , -0.18],
       [-0.23, -0.11],
```

Just a look-up operation

```
1 display(list(oh.categories_[0]))
```

```
['R11',
 'R21',
 'R22',
 'R23',
 'R24',
 'R25',
 'R26',
 'R31',
 'R41',
 'R42',
 'R43',
 'R52',
 'R53',
 'R54',
 'R72',
 'R73',
```

```
1 W + b
```

```
array([[-0.06, -0.24],
       [ 0.1 ,  0.02],
       [-0.32, -0.28],
       [ 0.02, -0.31],
       [ 0.39, -0.28],
       [ 0.1 , -0.41],
       [ 0.23, -0.26],
       [ 0.13,  0.31],
       [ 0.63, -0.44],
       [ 0.18, -0.01],
       [-0.2 , -0.38],
       [ 0.77, -0.23],
       [ 0.52, -0.05],
       [ 0.78, -0.43],
       [-0.2 , -0.18],
       [-0.23, -0.11],
```

Turn the region into an index

```
oe = OrdinalEncoder()
X_train_reg = oe.fit_transform(X_train_raw[["Region"]])
X_test_reg = oe.transform(X_test_raw[["Region"]])

for i, reg in enumerate(oe.categories_[0][:3]):
  print(f"The Region value {reg} gets turned into {i}.")
```

```
The Region value R11 gets turned into 0.
The Region value R21 gets turned into 1.
The Region value R22 gets turned into 2.
```

Use an Embedding layer

```
from keras.layers import Embedding
num_regions = X_train_raw["Region"].nunique()

random.seed(12)
model = Sequential([
  Embedding(input_dim=num_regions, output_dim=2),
  Dense(1, activation="exponential")
])

model.compile(optimizer="adam", loss="poisson")
```

```
es = EarlyStopping(verbose=True)
hist = model.fit(X_train_reg, y_train, epochs=100, verbose=0,
    validation_split=0.2, callbacks=[es])
hist.history["val_loss"][-1]
```

```
Epoch 5: early stopping
```

```
0.7681587934494019
```

```
model.layers
```

```
[<Embedding name=embedding_1, built=True>, <Dense name=dense_6, built=True>]
```

Keras' Embedding Layer

```
1 model.layers[0].get_weights()[0]
```

```
array([[ 0.06, -0.09],
       [-0.02,  0.03],
       [-0.04, -0.02],
       [ 0.1 , -0.12],
       [ 0.24, -0.22],
       [ 0.17, -0.2 ],
       [ 0.18, -0.18],
       [-0.05,  0.09],
       [ 0.37, -0.34],
       [ 0.03, -0.01],
       [ 0.02, -0.08],
       [ 0.37, -0.31],
       [ 0.21, -0.16],
       [ 0.43, -0.38],
       [-0.03, -0.01],
       [-0.06,  0.02],
```

```
1 X_train_raw["Region"].head(4)
```

```
0    R24
1    R21
2    R53
3    R24
Name: Region, dtype: str
```

```
1 X_sample = X_train_reg[:4].to_numpy()
2 X_sample
```

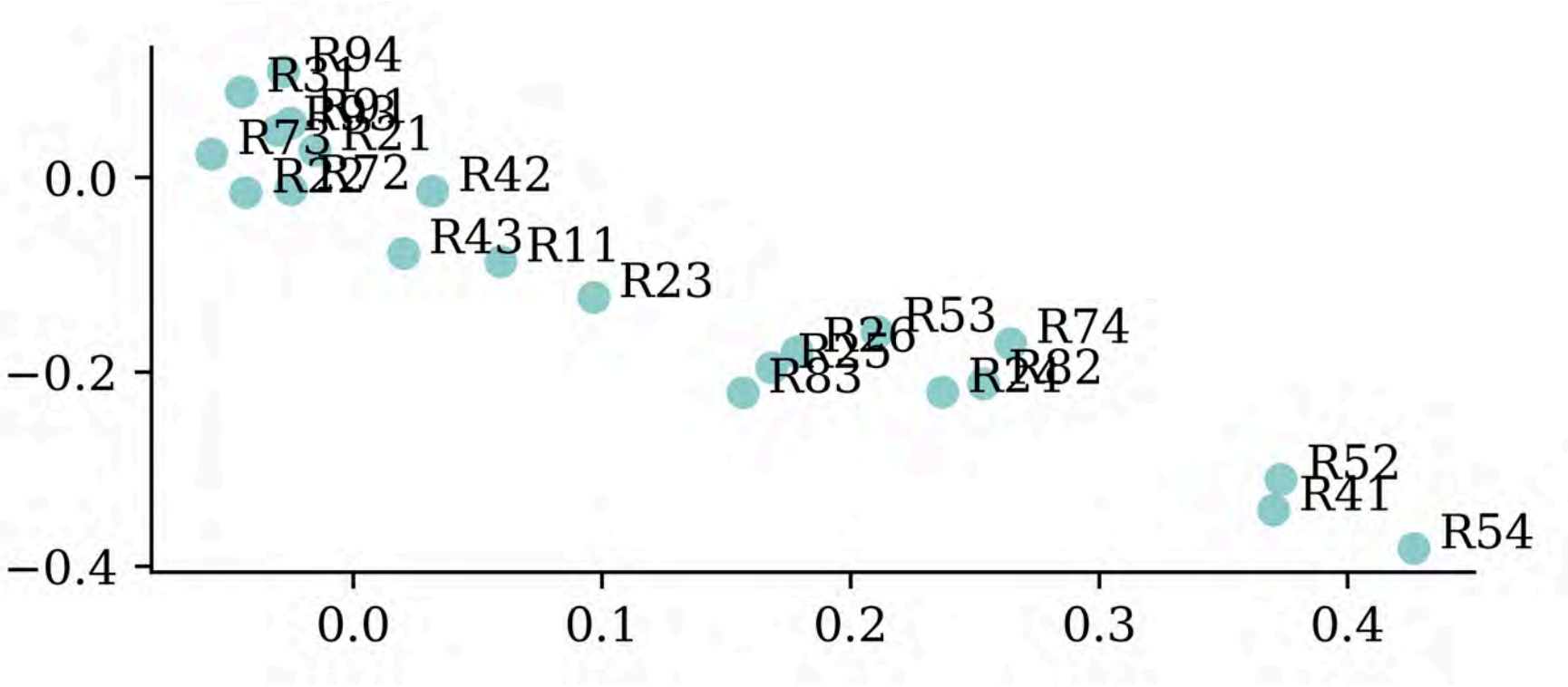
```
array([[ 4.],
       [ 1.],
       [12.],
       [ 4.]])
```

```
1 enc_tensor = model.layers[0](X_sample)
2 keras.ops.convert_to_numpy(enc_tensor).sc
```

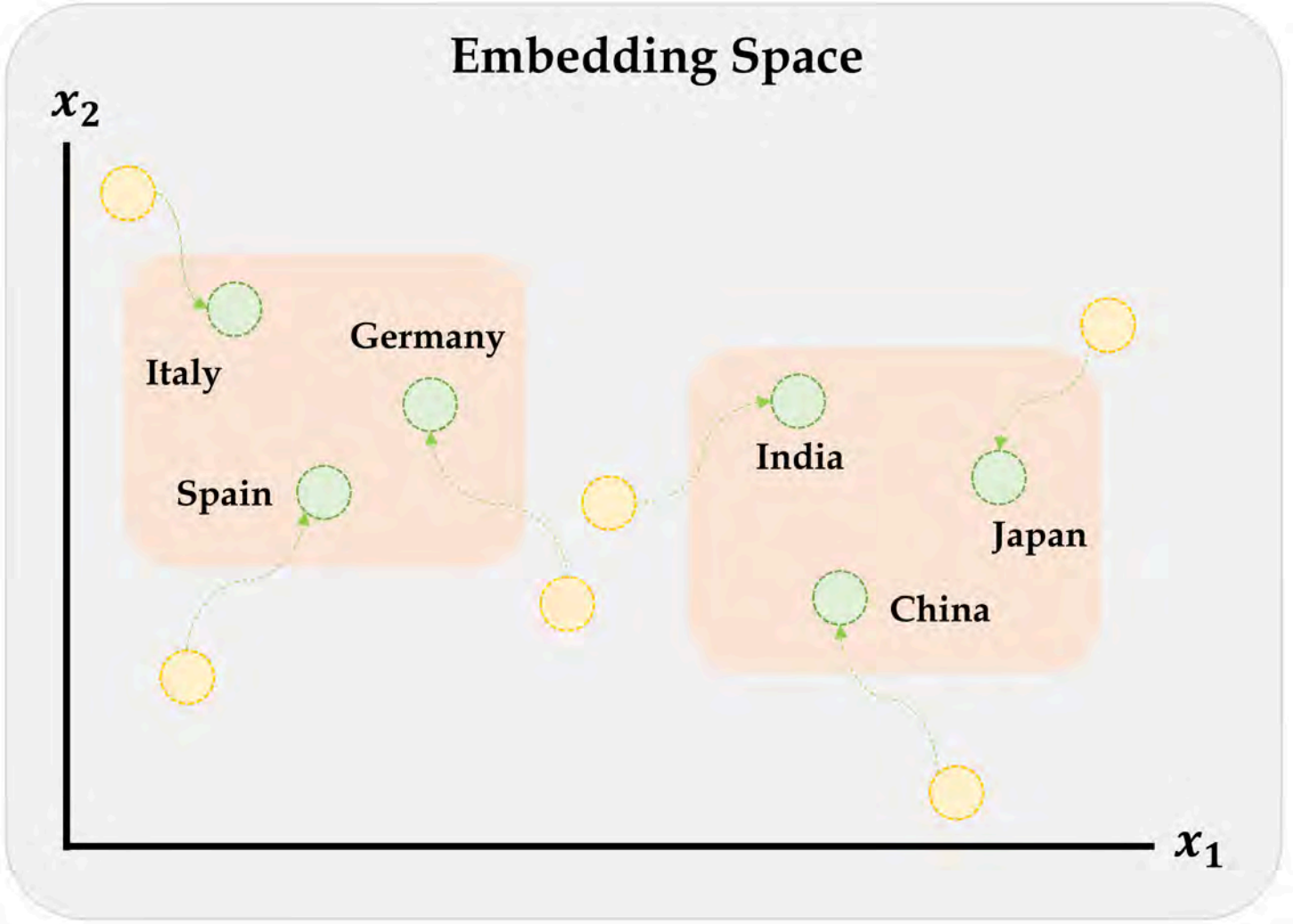
```
array([[ 0.24, -0.22],
       [-0.02,  0.03],
       [ 0.21, -0.16],
       [ 0.24, -0.22]], dtype=float32)
```

The learned embeddings

```
1 points = model.layers[0].get_weights()[0]
2 plt.scatter(points[:,0], points[:,1])
3 for i in range(num_regions):
4   plt.text(points[i,0]+0.01, points[i,1] , s=oe.categories_[0][i])
```



Entity embeddings



Embeddings will gradually improve during training.

Source: Marcus Lautier (2022).

Embeddings & other inputs

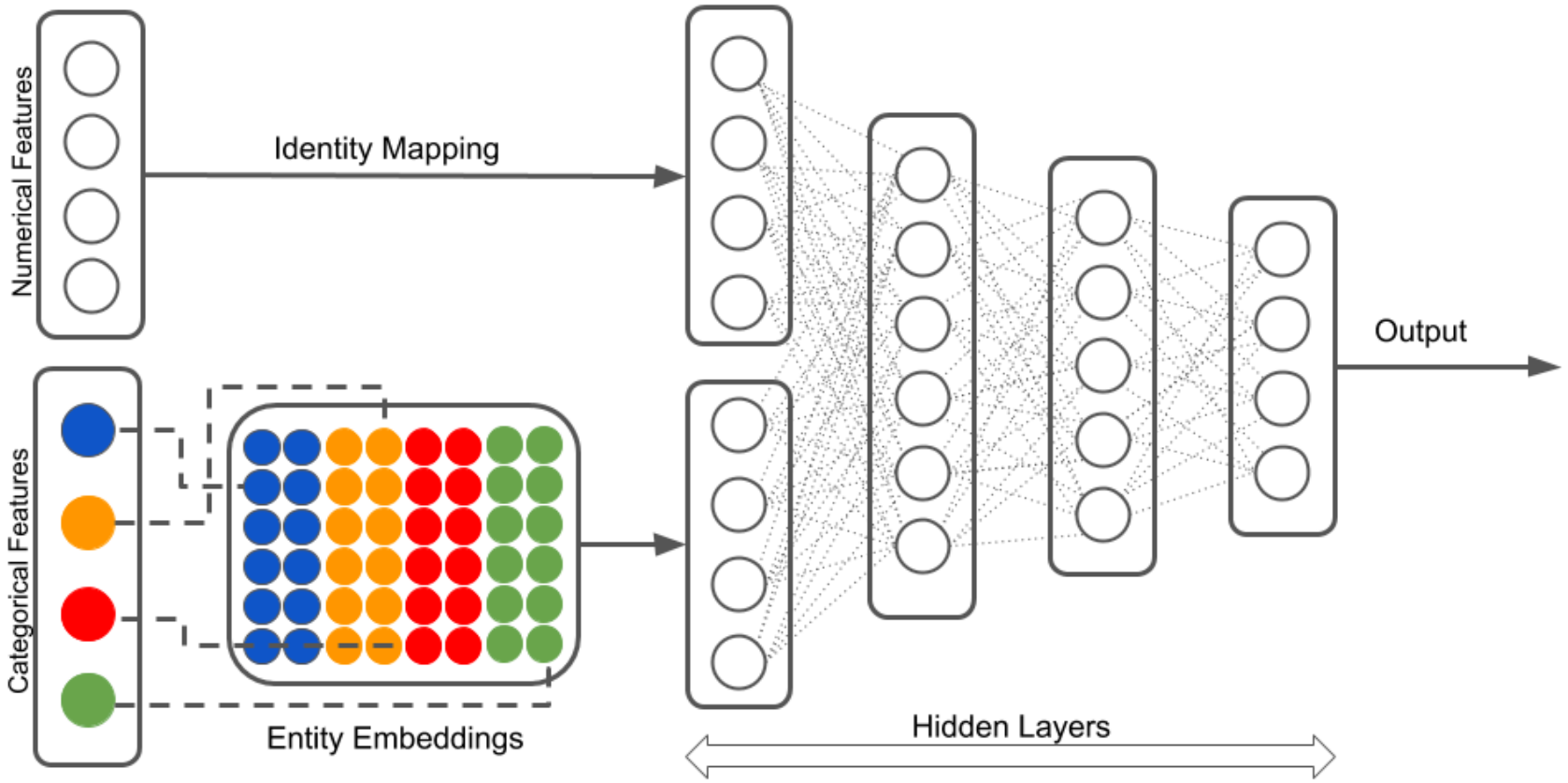


Illustration of a neural network with both continuous and categorical inputs.

We can't do this with Sequential models...

Source: LotusLabs Blog, Accurate insurance claims prediction with Deep Learning.

Lecture Outline

- Word Embeddings
- Word Embeddings II
- Car Crash NLP Part II
- Entity Embedding
- Categorical Variables & Entity Embeddings
- **Keras' Functional API**
- French Motor Dataset with Embeddings
- Scale By Exposure

Converting Sequential models

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

```
random.seed(12)

model = Sequential([
  Dense(30, "leaky_relu"),
  Dense(1, "exponential")
])

model.compile(
  optimizer="adam",
  loss="poisson")

hist = model.fit(
  X_train_oh, y_train,
  epochs=1, verbose=0,
  validation_split=0.2)
hist.history["val_loss"][-1]
```

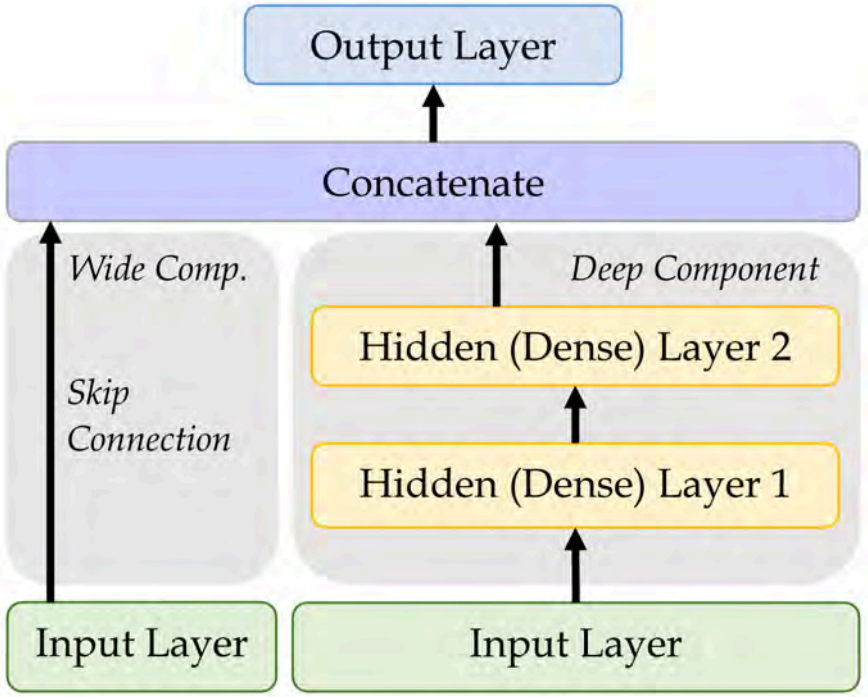
```
0.7700952887535095
```

```
random.seed(12)

inputs = Input(shape=(X_train_oh.shape[1]
x = Dense(30, "leaky_relu")(inputs)
out = Dense(1, "exponential")(x)
model = Model(inputs, out)

model.compile(
  optimizer="adam",
  loss="poisson")

hist = model.fit(
  X_train_oh, y_train,
  epochs=1, verbose=0,
  validation_split=0.2)
hist.history["val_loss"][-1]
```

```
0.7697821855545044
```

See one-length tuples.

Wide & Deep network



An illustration of the wide & deep network architecture.

Add a *skip connection* from input to output layers.

```
from keras.layers \
    import Concatenate

inp = Input(shape=X_train.shape[1:])
hidden1 = Dense(30, "leaky_relu")(inp)
hidden2 = Dense(30, "leaky_relu")(hidden1)
concat = Concatenate()(
  [inp, hidden2])
output = Dense(1)(concat)
model = Model(
    inputs=[inp],
    outputs=[output])
```

Sources: Marcus Lautier (2022) & Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, Chapter 10 code snippet.

Naming the layers

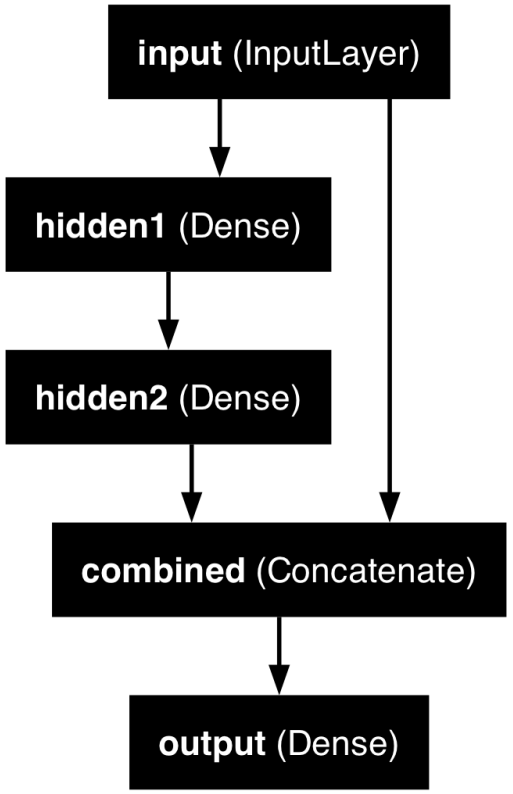
For complex networks, it is often useful to give meaningful names to the layers.

```
input_ = Input(shape=X_train.shape[1:], name="input")
hidden1 = Dense(30, activation="leaky_relu", name="hidden1")(input_)
hidden2 = Dense(30, activation="leaky_relu", name="hidden2")(hidden1)
concat = Concatenate(name="combined")([input_, hidden2])
output = Dense(1, name="output")(concat)
model = Model(inputs=[input_], outputs=[output])
```

Inspecting a complex model

```
1 from keras.utils import plot_model
```

```
1 plot_model(model, show
```




```
1 model.summary(line_length=75)
```

```
Model: "functional_18"
```

Layer (type)	Output Shape	Param #	Connected
input (InputLayer)	(None, 39)	0	–
hidden1 (Dense)	(None, 30)	1,200	input[0][0
hidden2 (Dense)	(None, 30)	930	hidden1[0]
combined (Concatenate)	(None, 69)	0	input[0][0 hidden2[0]
output (Dense)	(None, 1)	70	combined[0

```
Total params: 2,200 (8.59 KB)
Trainable params: 2,200 (8.59 KB)
Non-trainable params: 0 (0.00 B)
```

UNSW SYDNEY

Lecture Outline

- Word Embeddings
- Word Embeddings II
- Car Crash NLP Part II
- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- **French Motor Dataset with Embeddings**
- Scale By Exposure

The desired architecture

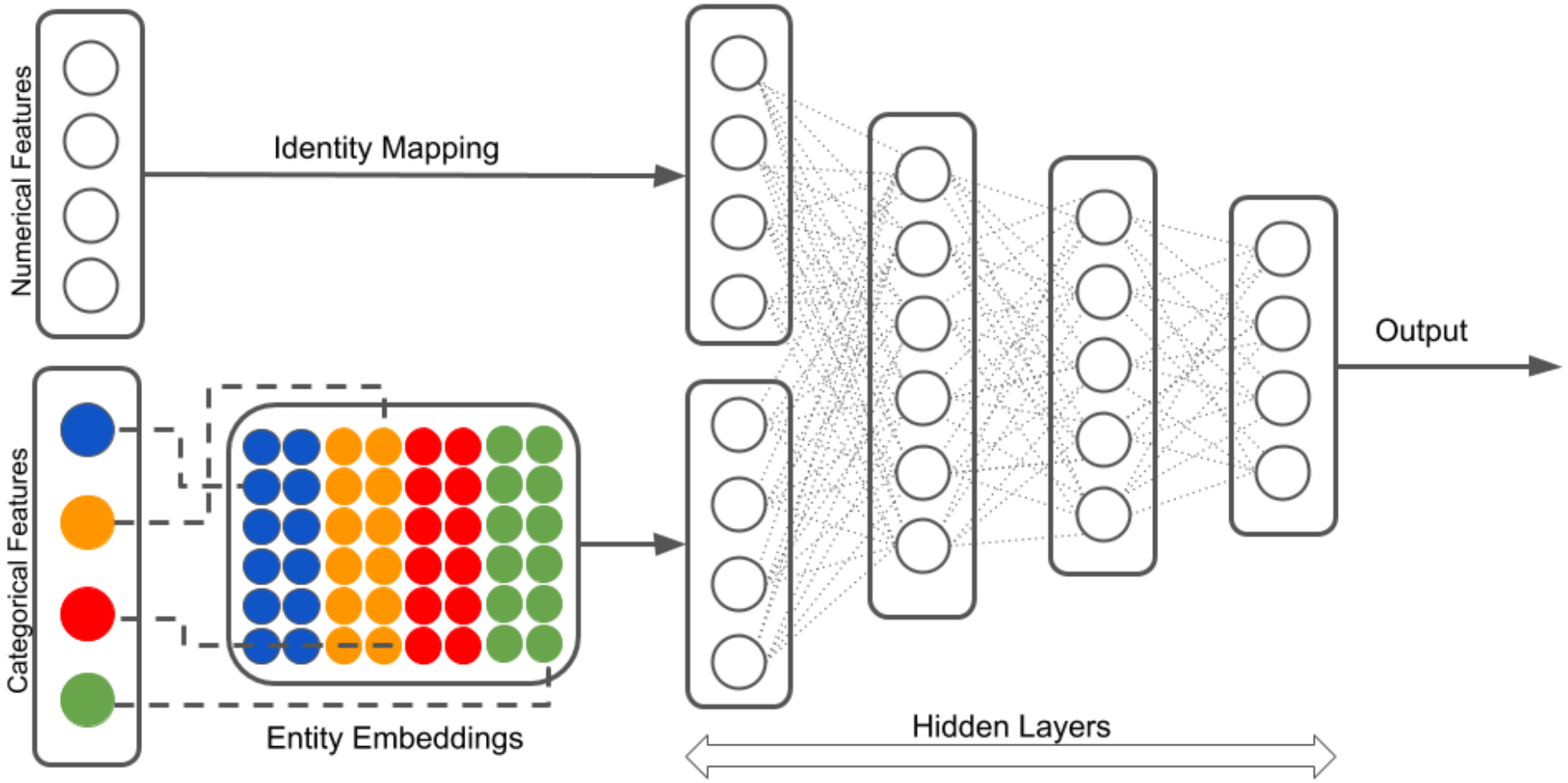


Illustration of a neural network with both continuous and categorical inputs.

Source: LotusLabs Blog, Accurate insurance claims prediction with Deep Learning.

Preprocess all French motor inputs

Transform the categorical variables to integers:

```
num_brands, num_regions = X_train_raw[["VehBrand", "Region"]].nunique()

ct = make_column_transformer(
  (OrdinalEncoder(), ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train = ct.fit_transform(X_train_raw)
X_test = ct.transform(X_test_raw)
```

Split the brand and region data apart from the rest:

```
X_train_brand = X_train["VehBrand"]
X_train_region = X_train["Region"]
X_train_rest = X_train.drop(["VehBrand", "Region"], axis=1)

X_test_brand = X_test["VehBrand"]
X_test_region = X_test["Region"]
X_test_rest = X_test.drop(["VehBrand", "Region"], axis=1)
```

Organise the inputs

Make a Keras `Input` for: vehicle brand, region, & others.

```
veh_brand = Input(shape=(1,), name="veh_brand")
region = Input(shape=(1,), name="region")
other_inputs = Input(shape=X_train_rest.shape[1:], name="other_inputs")
```

Create embeddings and join them with the other inputs.

```
from keras.layers import Reshape

random.seed(1337)
veh_brand_ee = Embedding(input_dim=num_brands, output_dim=2,
    name="veh_brand_ee")(veh_brand)
veh_brand_ee = Reshape(target_shape=(2,))(veh_brand_ee)

region_ee = Embedding(input_dim=num_regions, output_dim=2,
    name="region_ee")(region)
region_ee = Reshape(target_shape=(2,))(region_ee)

x = Concatenate(name="combined")([veh_brand_ee, region_ee, other_inputs])
```

Complete the model and fit it

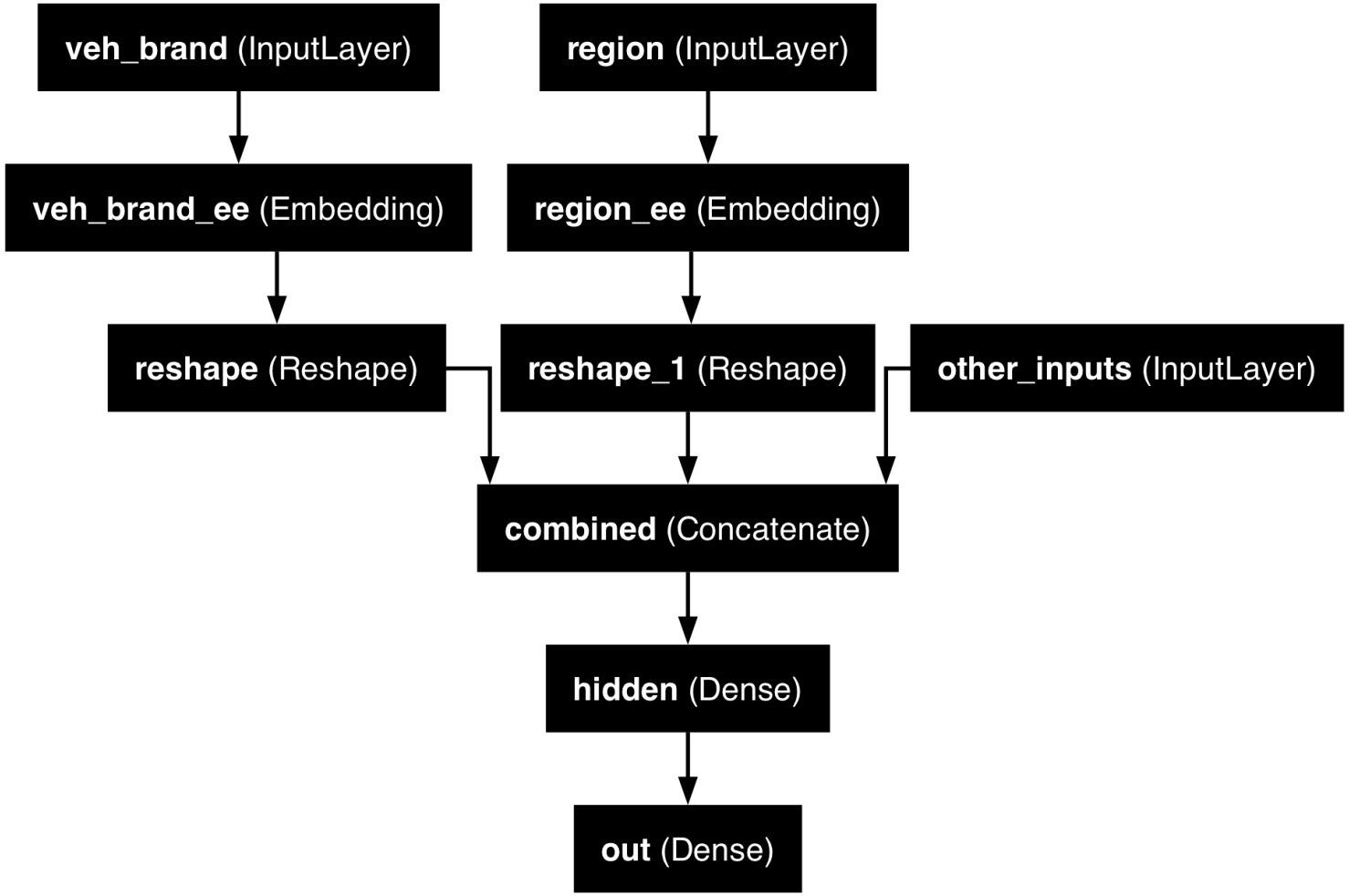
Feed the combined embeddings & continuous inputs to some normal dense layers.

```
x = Dense(30, "relu", name="hidden")(x)
out = Dense(1, "exponential", name="out")(x)

model = Model([veh_brand, region, other_inputs], out)
model.compile(optimizer="adam", loss="poisson")

hist = model.fit((X_train_brand, X_train_region, X_train_rest),
    y_train, epochs=100, verbose=0,
    callbacks=[EarlyStopping(patience=5)], validation_split=0.2)
np.min(hist.history["val_loss"])
```

```
np.float64(0.6853022575378418)
```

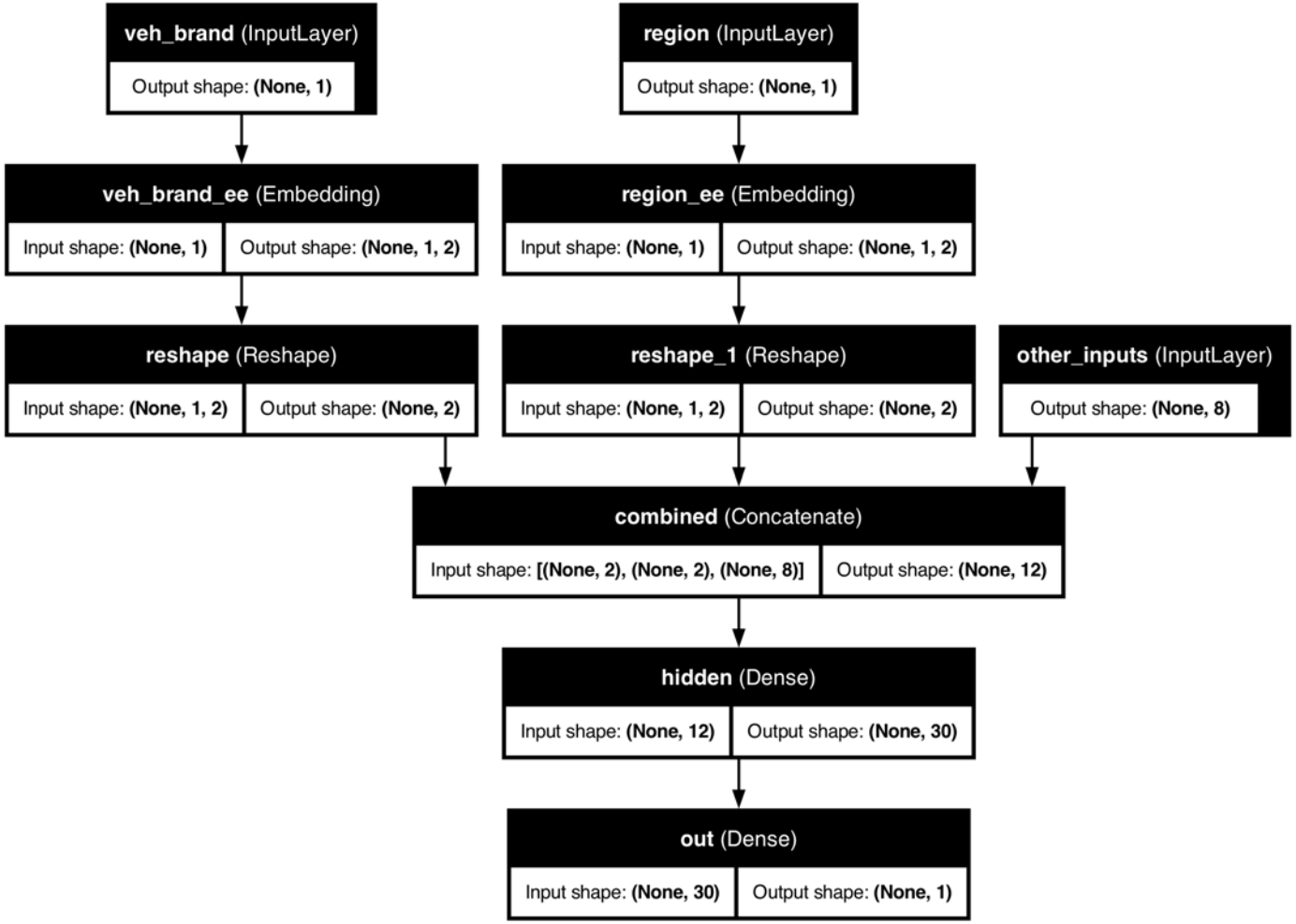
Plotting this model

```
1 plot_model(model, show_layer_names=True)
```



Why we need to reshape

```
1 plot_model(model, show_layer_names=True, show_shapes=True)
```



UNSW SYDNEY

Lecture Outline

- Word Embeddings
- Word Embeddings II
- Car Crash NLP Part II
- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- **Scale By Exposure**

Two different models

Have $\{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,n}$ for $\mathbf{x}_i \in \mathbb{R}^{47}$ and $y_i \in \mathbb{N}_0$.

Model 1: Say $Y_i \sim \mathsf{Poisson}(\lambda(\mathbf{x}_i))$.

But, the exposures are different for each policy. $\lambda(\mathbf{x}_i)$ is the expected number of claims for the duration of policy i's contract.

Model 2: Say $Y_i \sim \mathsf{Poisson}(\text{Exposure}_i \times \lambda(\mathbf{x}_i))$.

Now, $\text{Exposure}_i \notin \mathbf{x}_i$, and $\lambda(\mathbf{x}_i)$ is the rate *per year*.

Just take continuous variables

```
ct = make_column_transformer(
  ("passthrough", ["Exposure"]),
  ("drop", ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train = ct.fit_transform(X_train_raw)
X_test = ct.transform(X_test_raw)
```

Split exposure apart from the rest:

```
X_train_exp = X_train["Exposure"]
X_test_exp = X_test["Exposure"]
X_train_rest = X_train.drop("Exposure", axis=1)
X_test_rest = X_test.drop("Exposure", axis=1)
```

Organise the inputs:

```
exposure = Input(shape=(1,), name="exposure")
other_inputs = Input(shape=X_train_rest.shape[1:], name="other_inputs")
```

Make & fit the model

Feed the continuous inputs to some normal dense layers.

```
random.seed(1337)
x = Dense(30, "relu", name="hidden1")(other_inputs)
x = Dense(30, "relu", name="hidden2")(x)
lambda_ = Dense(1, "exponential", name="lambda")(x)
```

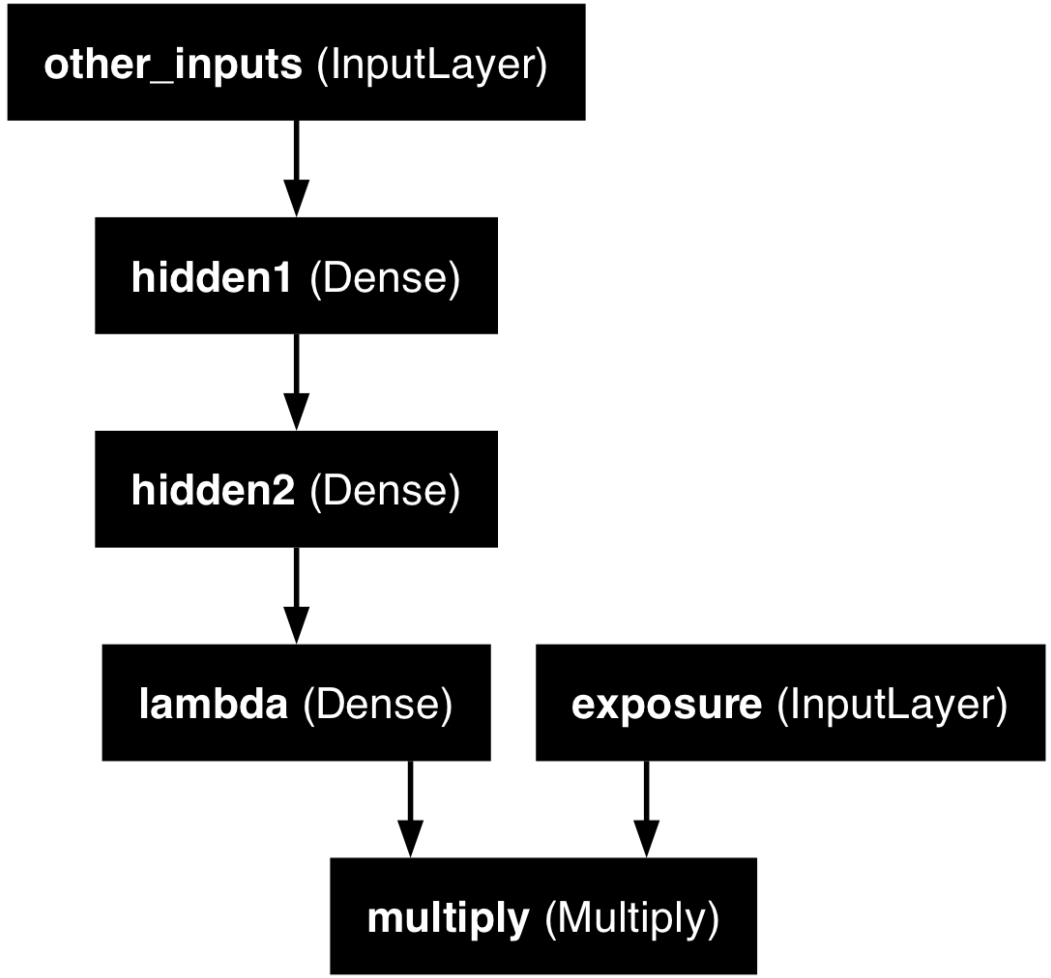
```
out = lambda_ * exposure # In past, need keras.layers.Multiply()[lambda_, exposure]
model = Model([exposure, other_inputs], out)
model.compile(optimizer="adam", loss="poisson")

es = EarlyStopping(patience=10, restore_best_weights=True, verbose=1)
hist = model.fit((X_train_exp, X_train_rest),
    y_train, epochs=100, verbose=0,
    callbacks=[es], validation_split=0.2)
np.min(hist.history["val_loss"])
```

```
Epoch 29: early stopping
Restoring model weights from the end of the best epoch: 19.
```

```
np.float64(0.9086945056915283)
```

Plot the model

```
1 plot_model(model, show_layer_names=True)
```



Package Versions

```
1 from watermark import watermark
2 print(watermark(python=True, packages="keras,matplotlib,numpy,pandas,seaborn,scipy,torch"
```

```
Python implementation: CPython
Python version       : 3.14.3
IPython version      : 9.13.0

keras     : 3.14.1
matplotlib: 3.10.9
numpy     : 2.4.4
pandas    : 3.0.2
seaborn   : 0.13.2
scipy     : 1.17.1
torch     : 2.11.0
```

Glossary

- entity embeddings
- Input layer
- Keras functional API
- Reshape layer
- skip connection
- wide & deep network