diplomatura universitaria en
inteligencia artificial

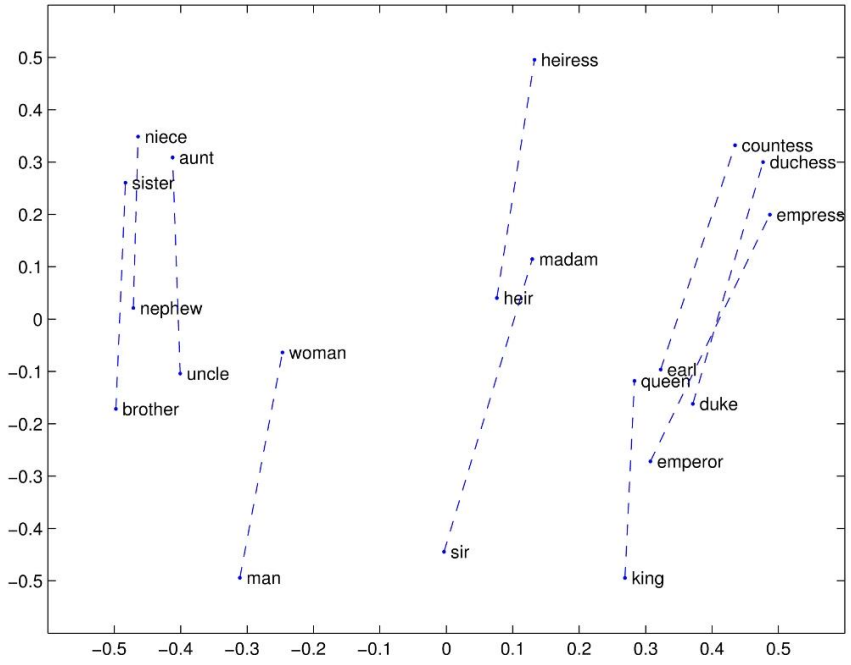
FACULTAD DE CIENCIAS
EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO
DE LA PROVINCIA DE BUENOS AIRES

Redes Neuronales

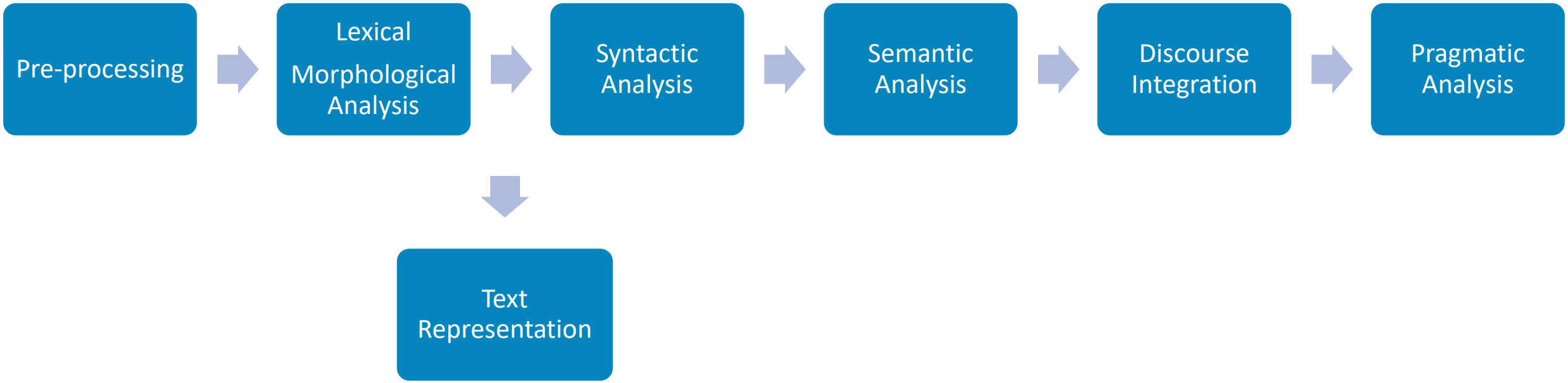
Word Embeddings- Word2Vec, GloVe

Agenda

- Repaso de representación de texto tradicional
- Embeddings (conceto)
- Word2Vec
- GloVe: Global Vectors for Word Representation
- Embeddings para recomendación



Workflow NLP



Representación Tradicional

Tradicionales

- Bag of Words
- Bag of N-Grams

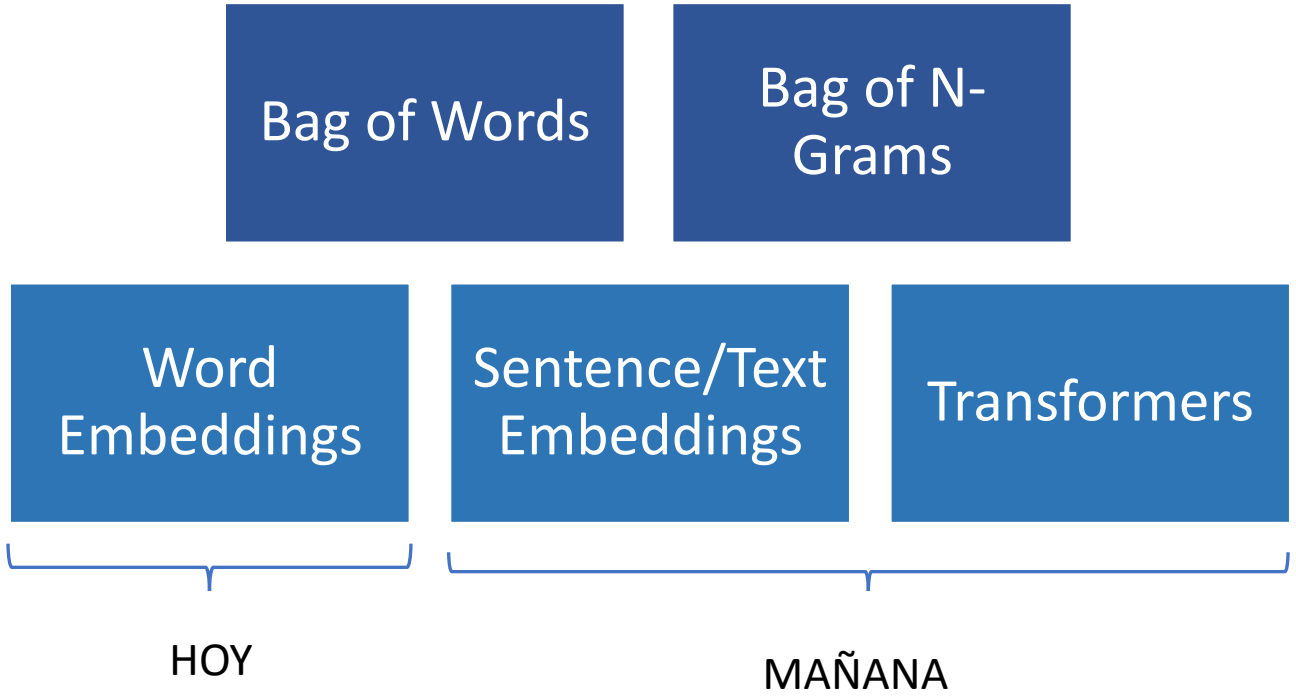
Deep Learning

- Word Embeddings
- Sentence/Text Embeddings
- Transformers

Representación Tradicional

Tradicionales

Deep Learning



Representación Tradicional

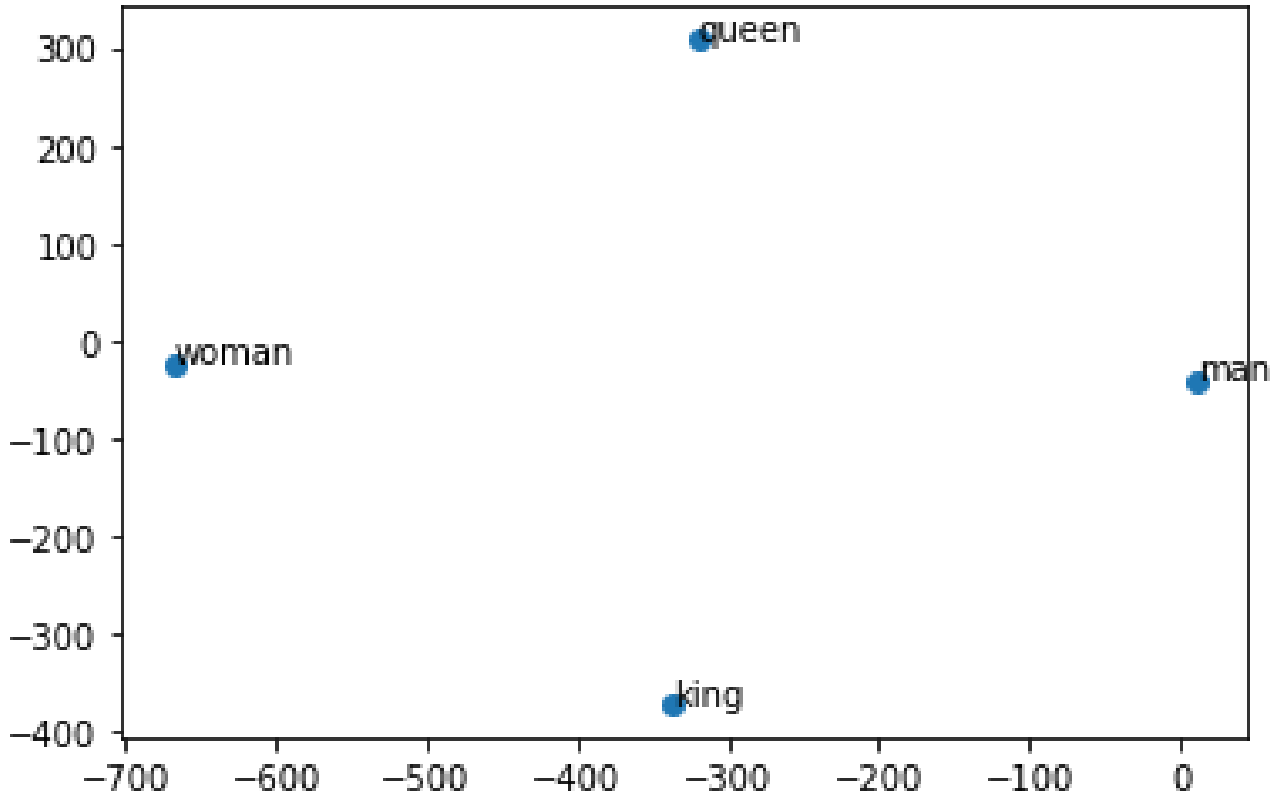
Bag-of-Words, TF-IDF,…

- La representación se basa en componentes léxicos del lenguaje.
- Generalmente, sufre del “course of dimensionality”. Pocas palabras son usada en cada texto, la representación es sparse.
- Zipf law: un subconjunto muy pequeño de palabras aparecen muy comúnmente en el texto.
- Pierden información semántica.
- Se pierde relación entre las palabras. Por ejemplo, “computadora” y “ordenador” son distintas para estas representaciones.

Embeddings

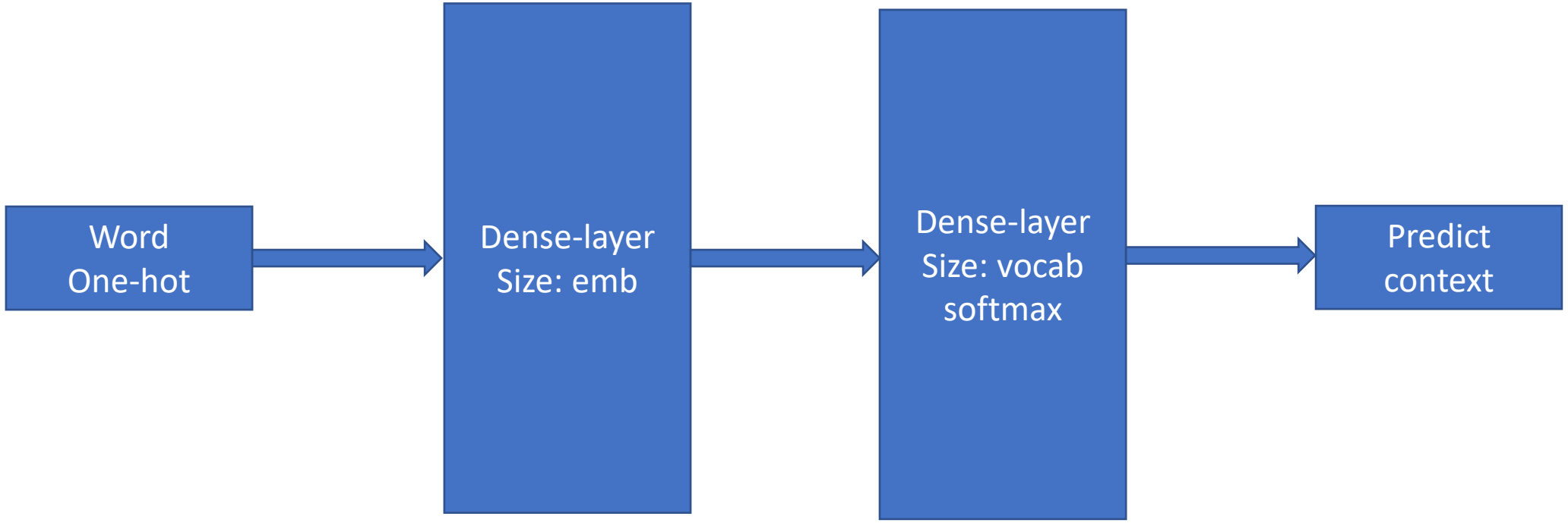
- El objetivo es mapear las palabras a vectores densos.
- Las dimensiones de esos vectores representan información semántica de las palabras.
- Palabras similares tienen vectores similares.

Embeddings


queen
woman
man
king
300
200
100
0
−100
−200
−300
−400
−700
−600
−500
−400
−300
−200
−100
0

Word2Vec

- Se basa en una red que:
  - Dado una palabra, predice el contexto (CBOW).
  - Dado un contexto, predice la palabra (Skip-Gram).



Word2Vec

Orange juice is a liquid extract of the orange tree fruit, produced by squeezing…

CBOW: Predecir las palabras en naranjas a partir de la palabra en azul.

Skip-Grma: Predecir las palabras en azul a partir de la palabra en naranja.

Orange Juice - Wikipedia:https://en.wikipedia.org/wiki/Orange_juice

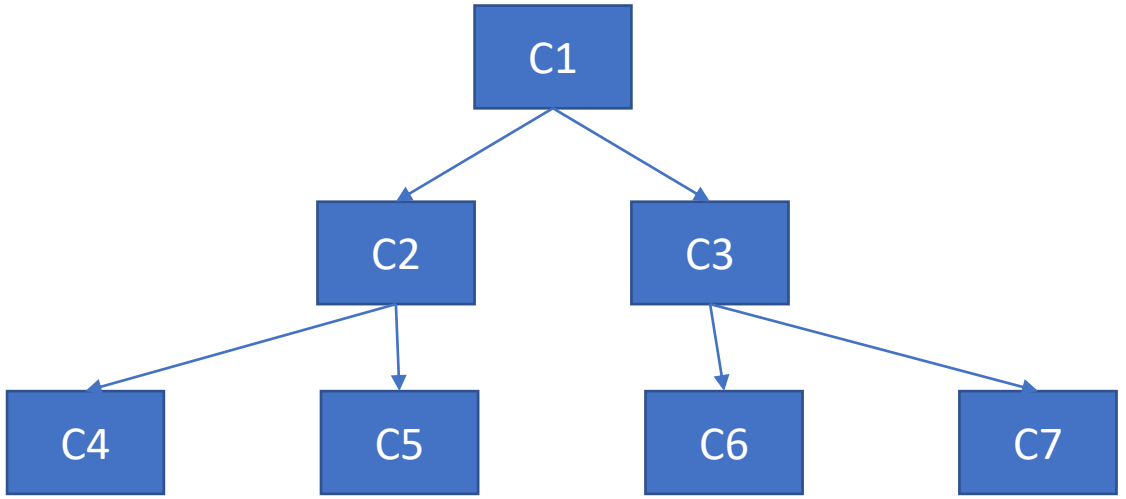
Word2Vec

$$P(w'_o|w_i) = \frac{\exp(w'^T_o w_i)}{\sum(\exp(w'^T w_i)}$$

- Básicamente es una softmax que calcula las probabilidades dado un vector para la palabra de entrada y otro para la salida.
- El problema es que los tamañas de las matrices son grande, ya que suele haber gran cantidad de términos (10^5, 10^7).

Distributed Representations of Words and Phrases and their Compositionality: https://arxiv.org/pdf/1310.4546.pdf

Word2Vec: Hierarchical softmax



Word2Vec: Negative sambling

Orange juice is a liquid extract of the orange tree fruit, produced by squeezing…

Orange Juice - Wikipedia:https://en.wikipedia.org/wiki/Orange_juice

Word2Vec: Negative sambling

Orange juice is a liquid extract of the orange tree fruit, produced by squeezing…

Target	Context	Output
Orange	Tree	1
Orange	Fruit	1
Orange	Pizza	0
Orange	King	0
Orange	Plane	0

Orange Juice - Wikipedia:https://en.wikipedia.org/wiki/Orange_juice

Optimización

$$(0,0,1,0,0)\cdot\begin{pmatrix} v_{1,1} & v_{1,2} & v_{1,3} \\ v_{2,1} & v_{2,2} & v_{2,3} \\ v_{3,1} & v_{3,2} & v_{3,3} \\ v_{4,1} & v_{4,2} & v_{4,3} \\ v_{5,1} & v_{5,2} & v_{5,3} \end{pmatrix} = (0v_{1,1} + 0v_{2,1} + 1v_{3,1} + 0v_{4,1} + 0v_{5,1} \quad \ldots + 1v_{3,2} + \cdots \quad \ldots + 1v_{3,3} + \cdots)$$

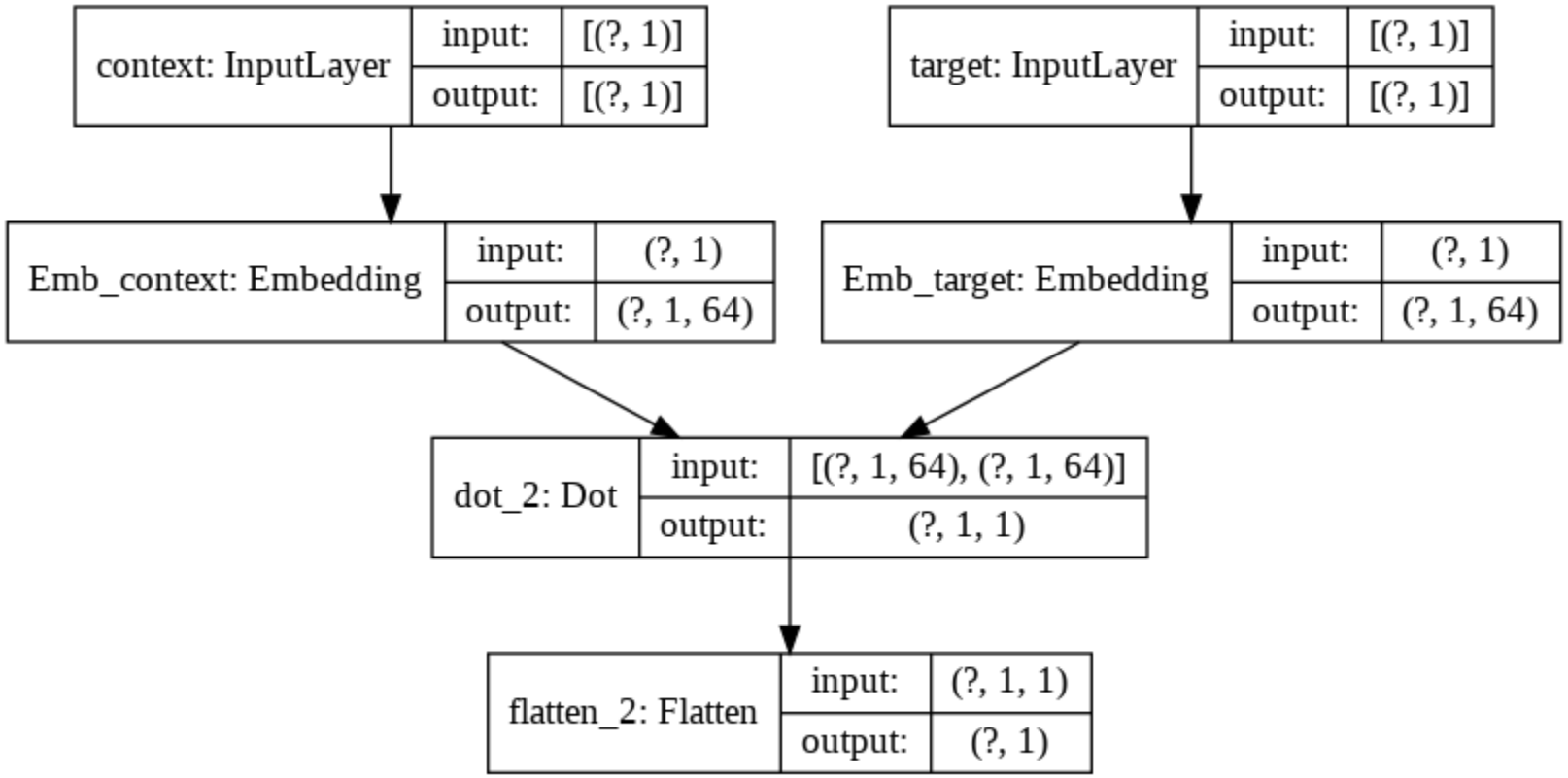
Optimización

$$\begin{pmatrix} v_{1,1} & v_{1,2} & v_{1,3} \\ v_{2,1} & v_{2,2} & v_{2,3} \\ v_{3,1} & v_{3,2} & v_{3,3} \\ v_{4,1} & v_{4,2} & v_{4,3} \\ v_{5,1} & v_{5,2} & v_{5,3} \end{pmatrix} [3,:] = (v_{3,1} \quad v_{3,2} \quad v_{3,3})$$

Word2Vec

```
context_emb = Embedding(len(id_words), 64, name='Emb_context')
target_emb = Embedding(len(id_words), 64, name='Emb_target')

context = Input((1,), name='context')
emb = context_emb(context)
target = Input((1,), name='target')
embT = target_emb(target)
lam = dot([emb, embT], axes=(-1))
lam = Flatten()(lam)
#lam = Activation('sigmoid')(lam)

model = Model(inputs=[context, target], outputs=lam)
model.compile('adam', minus_max_likelihood)
#model.compile('adam', 'binary_crossentropy')
model.summary()
```

Word2Vec


context: InputLayer
input: [(?, 1)]
output: [(?, 1)]
target: InputLayer
input: [(?, 1)]
output: [(?, 1)]
Emb_context: Embedding
input: (?, 1)
output: (?, 1, 64)
Emb_target: Embedding
input: (?, 1)
output: (?, 1, 64)
dot_2: Dot
input: [(?, 1, 64), (?, 1, 64)]
output: (?, 1, 1)
flatten_2: Flatten
input: (?, 1, 1)
output: (?, 1)

Word2Vec

```
def cos(v1, v2):
    return np.dot(v1, v2.T) / (np.dot(v1, v1.T) ** 0.5 * np.sum(v2 * v2, axis=-
1) ** 0.5)


def nearest(voc, wv, top=11):
    dist = cos(wv, voc)
    a = range(len(dist))
    a = sorted(a, key=lambda x: dist[x], reverse=True)
    return a[0:top]

print('Similares a car:')
for i in nearest(vectors, vectors[words_id['car'], :])[1:]:
    print('\t{}'.format(id_words[i]))

print('Similares a ford:')
for i in nearest(vectors, vectors[words_id['ford'], :])[1:]:
    print('\t{}'.format(id_words[i]))
```

Word2Vec

```
Similares a car: bike little problem get person time
helmet thing engine buy

Similares a ford: gt yamaha mk explorer eagle coupe gate
gap vhf sedan

Similares a law: enforcement government agencies
communications agency federal security encryption
legitimate technology
```

GloVe

- Basado en matriz de coocurrencia. Se cuentan todas las coocurencias entre 2 palabras dividiendo por las distancias.

- Se entrena un modelo que prediga el valor del logaritmo de las coocurrencias.Type equation here.

$$J = f(X_{ij})(w_i^T w_j + b_i + b_j - \log(X_{ij}))^2$$

GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/pubs/glove.pdf

GloVe

- Basado en matriz de coocurrencia. Se cuentan todas las coocurencias entre 2 palabras dividiendo por las distancias.

- Se entrena un modelo que prediga el valor del logaritmo de las coocurrencias.Type equation here.

$$J = f(X_{ij})(w_i^T w_j + b_i + b_j - \log(X_{ij}))^2$$

$$f(X_{ij}) = \begin{cases} \left(\frac{X_{ij}}{XMAX}\right)^a & if\ X_{ij} < XMAX \\ 1 & \end{cases}$$

GloVe: Global Vectors for Word Representation https://nlp.stanford.edu/pubs/glove.pdf

GloVe

```
def custom_loss(y_true, y_pred, a = 3.0/4.0, X_MAX=100):
    return K.sum(K.pow(K.clip(y_true / X_MAX, 0.0, 1.0), a) *
                       K.square(y_pred - K.log(y_true)), axis=-1)
```

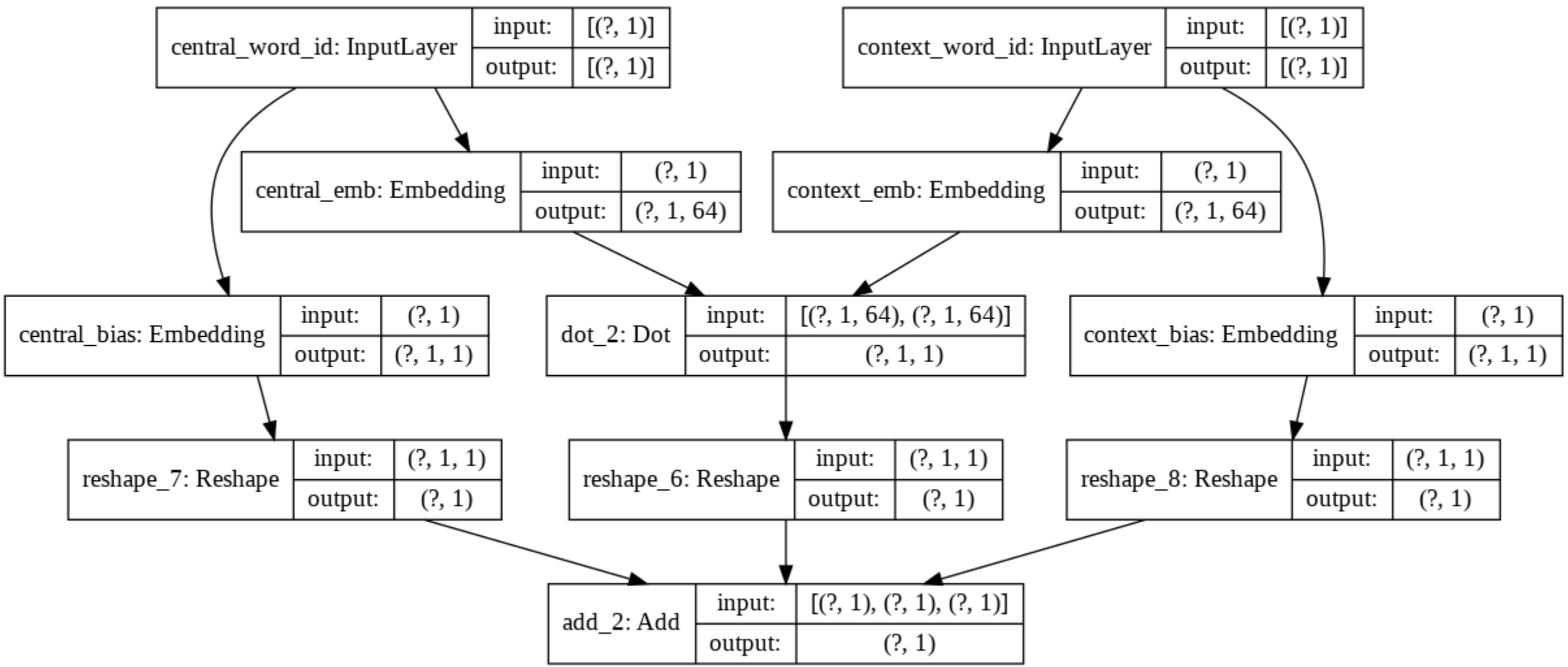
GloVe

```
central_embedding = Embedding(vocab_size+1, vector_dim, input_length=1, name='central_
emb')
central_bias = Embedding(vocab_size+1, 1, input_length=1, name='central_bias')

context_embedding = Embedding(vocab_size, vector_dim, input_length=1, name='context_em
b')
context_bias = Embedding(vocab_size, 1, input_length=1, name='context_bias')
```

GloVe

```
input_target = Input((1,), name='central_word_id')
input_context = Input((1,), name='context_word_id’)

vector_target = central_embedding(input_target)
vector_context = context_embedding(input_context)

bias_target = central_bias(input_target)
bias_context = context_bias(input_context)

dot_product = Dot(axes=-1)([vector_target, vector_context])
dot_product = Reshape((1, ))(dot_product)
bias_target = Reshape((1,))(bias_target)
bias_context = Reshape((1,))(bias_context)

prediction = Add()([dot_product, bias_target, bias_context])

model = Model(inputs=[input_target, input_context], outputs=prediction)
model.compile(loss=custom_loss, optimizer='adam')
```

GloVe


central_word_id: InputLayer
input: [(?, 1)]
output: [(?, 1)]
context_word_id: InputLayer
input: [(?, 1)]
output: [(?, 1)]
central_emb: Embedding
input: (?, 1)
output: (?, 1, 64)
context_emb: Embedding
input: (?, 1)
output: (?, 1, 64)
central_bias: Embedding
input: (?, 1)
output: (?, 1, 1)
dot_2: Dot
input: [(?, 1, 64), (?, 1, 64)]
output: (?, 1, 1)
context_bias: Embedding
input: (?, 1)
output: (?, 1, 1)
reshape_7: Reshape
input: (?, 1, 1)
output: (?, 1)
reshape_6: Reshape
input: (?, 1, 1)
output: (?, 1)
reshape_8: Reshape
input: (?, 1, 1)
output: (?, 1)
add_2: Add
input: [(?, 1), (?, 1), (?, 1)]
output: (?, 1)

GloVe

```
input_target = Input((1,), name='central_word_id')
input_context = Input((1,), name='context_word_id’)

vector_target = central_embedding(input_target)
vector_context = context_embedding(input_context)

bias_target = central_bias(input_target)
bias_context = context_bias(input_context)

dot_product = Dot(axes=-1)([vector_target, vector_context])
dot_product = Reshape((1, ))(dot_product)
bias_target = Reshape((1,))(bias_target)
bias_context = Reshape((1,))(bias_context)

prediction = Add()([dot_product, bias_target, bias_context])

model = Model(inputs=[input_target, input_context], outputs=prediction)
model.compile(loss=custom_loss, optimizer='adam')
```

GloVe

Similares a car: bike problem case point question place post answer best own

Similares a ford: gt user hypothesis mustang probe balloon physician patient reds goalie

Similares a law: agencies enforcement government police federal agency block output free president

Embeddins en clasificadores tradicionales

- Calcular el promedio de los embeddings para cada documento.
- Calcular el promedio ponderado por TF-IDF

Embeddins en clasificadores tradicionales

Promedio simple

```
def avg_vector(x, model):
    res = []
    for s in tqdm(x):
        s = tokenizer_simple(s)
        v = []
        for w in s:
            if w in model:
                v.append(model[w])
        res.append(np.mean(np.asarray(v), axis=0))
    return np.asarray(res)
```

Embeddins en clasificadores tradicionales

Promedio tf-idf

```
def tfidf_vector(x, x_tfidf, tfidf, model):
    res = []
    tokenizer = tfidf.build_analyzer()
    tfidf_map = {w: i for i, w in enumerate(tfidf.get_feature_names())}
    for i, s in tqdm(enumerate(x), total=len(x)):
        s = tokenizer(s)
        v = []
        total = 0
        for w in s:
            if w in model:
                if w not in tfidf_map:
                    continue
                j = tfidf_map[w]
                total += x_tfidf[i, j]
                v.append(x_tfidf[i, j] * model[w])
        res.append(np.sum(np.asarray(v), axis=0) / total)
    return np.asarray(res)
```

Embeddins en clasificadores tradicionales

```
tfidf = TfidfVectorizer()
xp_train = tfidf.fit_transform(x_train)
xp_test = tfidf.transform(x_test)
```

```
cls = SVC()
cls.fit(xp_train, y_train)

print(classification_report(y_test, cls.predict(xp_test)))
```

Embeddins en clasificadores tradicionales

```
xv_train = avg_vector(x_train, model)
xv_test = avg_vector(x_test, model)

cls = SVC()
cls.fit(xv_train, y_train)

print(classification_report(y_test, cls.predict(xv_test)))
```

Embeddins en clasificadores tradicionales

```
xv_train = tfidf_vector(x_train, xp_train, tfidf, model)
xv_test = tfidf_vector(x_test, xp_test, tfidf, model)

cls = SVC()
cls.fit(xv_train, y_train)

print(classification_report(y_test, cls.predict(xv_test)))
```

Recomendación

También se pueden usar los embeddings para hacer recomendación, por ejemplo de películas, productos, libros, etc.

	Producto 1	Producto 2	Producto 3	Producto 4	Producto 5	Producto 6	….
Usuario 1	4	?	2	?	5	1	…
Usuario 2	?	4	2	?	4	2	…
Usuario 3	2	1	5	2	?	?	…
Usuario 4	?	?	4	2	5	1	…
Usuario 5	?	3	?	4	?	?	…
…	…	…	…	…	…	…	…

Recomendación

Factorización de Matrices

	Pref. 1	Pref. 2	Pref. 3	Pref. 4	Pref. 5	Pref. 6	....
Usuario 1	?	?	?	?	?	?	...
Usuario 2	?	?	?	?	?	?	...
Usuario 3	?	?	?	?	?	?	...
Usuario 4	?	?	?	?	?	?	...
Usuario 5	?	?	?	?	?	?	...
...	...	...	...	...	...	...	...

Recomendación

Factorización de Matrices

	Pref. 1	Pref. 2	Pref. 3	Pref. 4	Pref. 5	Pref. 6	....
Producto 1	?	?	?	?	?	?	...
Producto 2	?	?	?	?	?	?	...
Producto 3	?	?	?	?	?	?	...
Producto 4	?	?	?	?	?	?	...
Producto 5	?	?	?	?	?	?	...
...	...	...	...	...	...	...	...

Buscamos las matrices tal que:

$$\hat{R} = U \cdot M^T + b_u + b_m$$

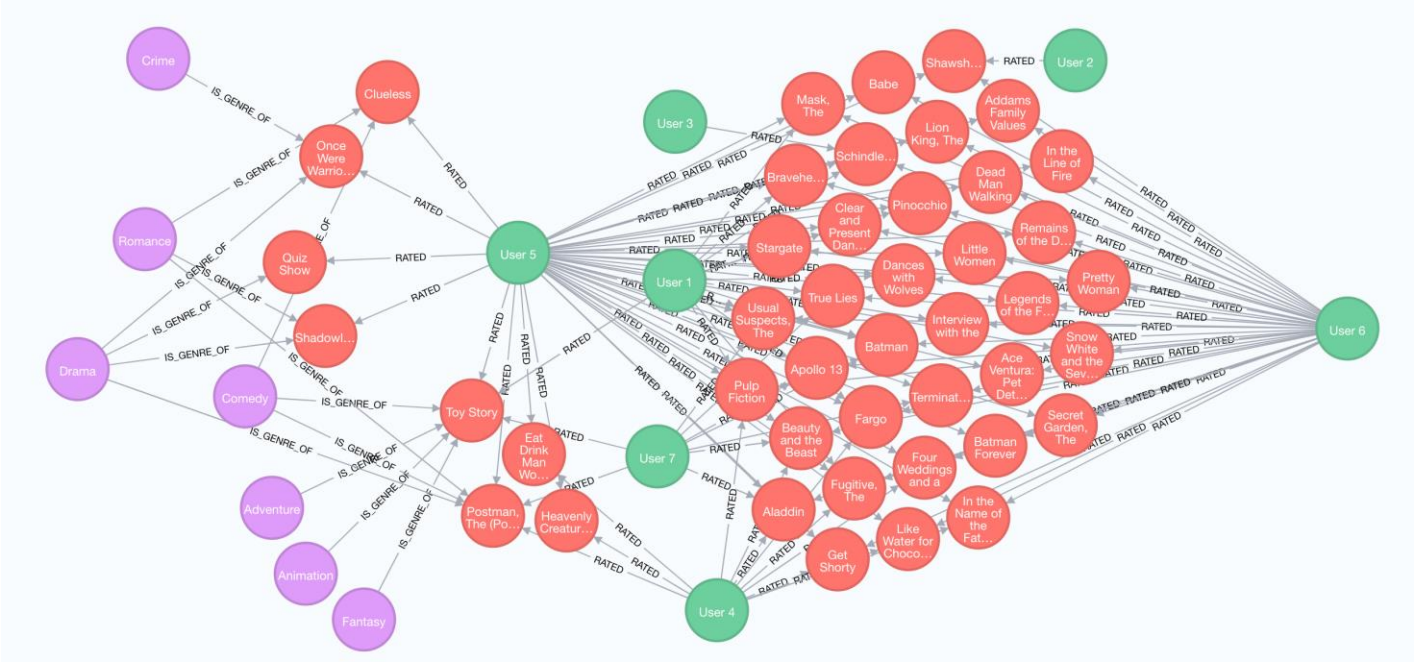
$$\underset{U,M}{\operatorname{argmin}} L(R, \hat{R}) = \sum \left(R_{ij} - \widehat{R_{ij}}\right)^2$$

Recomendación

Movielens

- Movilens es un dataset que contiene películas, usuarios y ratings.

- Además de esta información contiene otra información sobre las películas. (La cual no usaremos en este ejemplo)

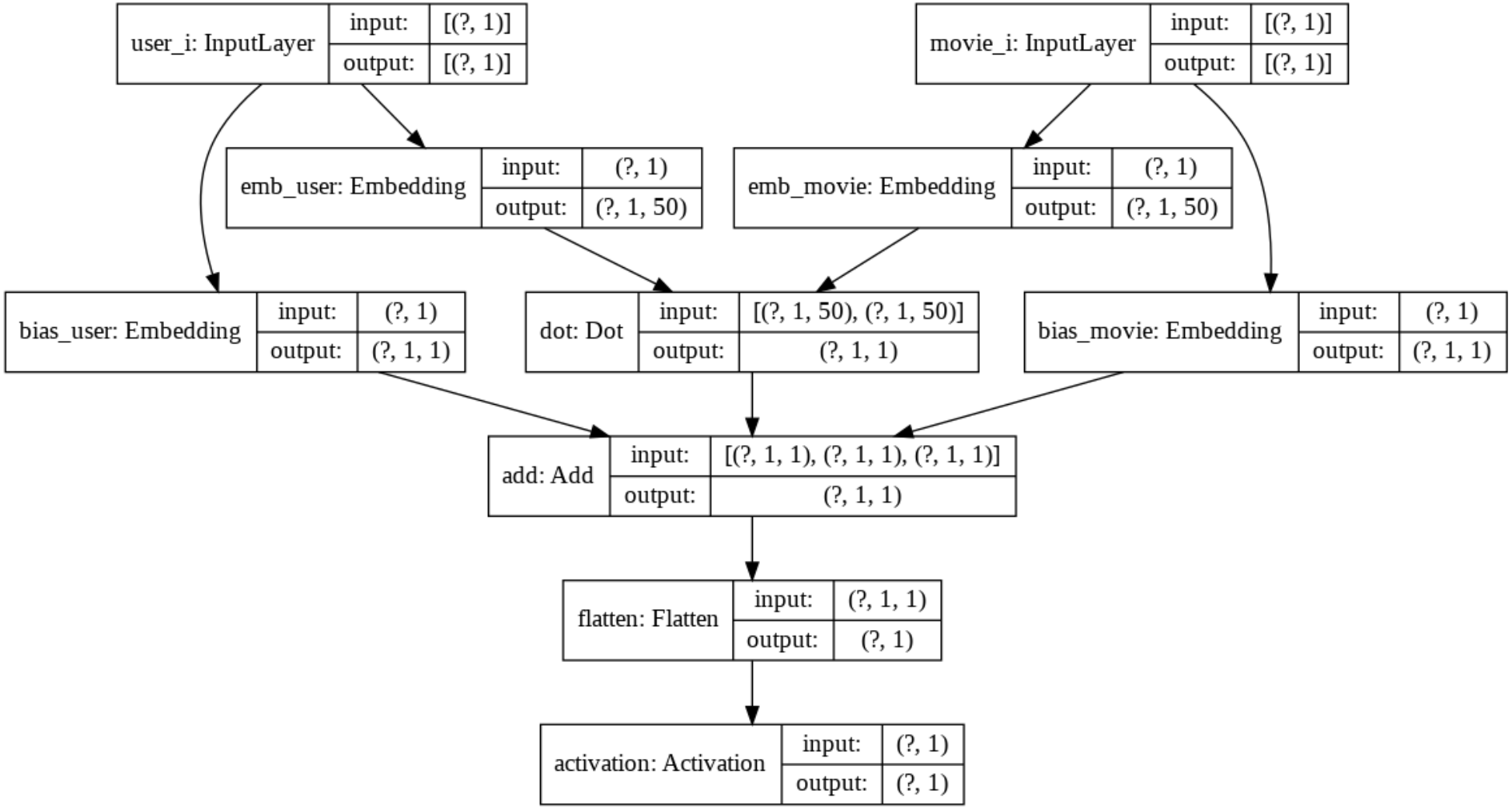


https://grouplens.org/datasets/movielens/

Recomendación

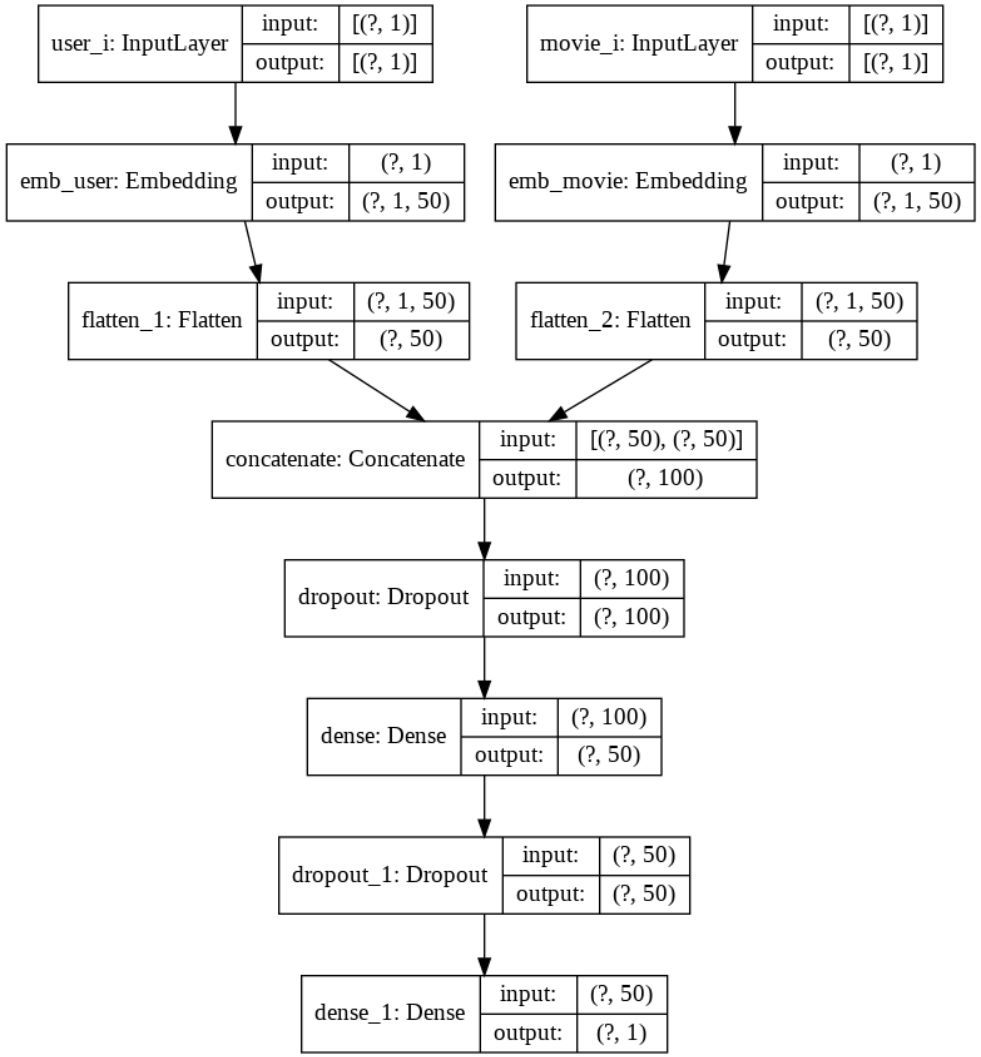
```
iu = Input((1,), name='user_i')
ue = Embedding(len(set(users))+1, 50, name='emb_user')(iu)
ub = Embedding(len(set(users))+1, 1, name='bias_user')(iu)
im = Input((1,), name='movie_i’)

me = Embedding(len(set(movies))+1,50, name='emb_movie')(im)
mb = Embedding(len(set(movies))+1, 1, name='bias_movie')(im)

dot = Dot(axes=-1)([ue, me])
biases = Add()([dot, ub, mb])
out = Activation('sigmoid')(Flatten()(biases))

model = Model([iu, im], out)

model.compile(loss='binary_crossentropy', optimizer='nadam', metrics=['mae'])
model.summary()
```

Recomendación



Recomendación

Otra arquitecturas…



Recomendación

Otra arquitecturas…

```
iu = Input((1,), name='user_i')
ue = Embedding(len(set(users))+1, 50, name='emb_user')(iu)

im = Input((1,), name='movie_i')
me = Embedding(len(set(movies))+1,50, name='emb_movie')(im)

f = Concatenate(axis=-1)([Flatten()(ue), Flatten()(me)])
f = Dropout(0.5)(f)

d = Dense(50)(f)
d = Dropout(0.5)(d)
d = Dense(1, activation='sigmoid')(d)

model = Model([iu, im], d)
```

Conclusiones

- Los embeddings de palabras capturan la semántica de la palabra mediante la relación de las palabras con palabras cercanas.
- Se pueden entrenar sin necesidad de tener un conjunto de datos entrenados.
- Manejan bien los sinónimos, ya que palabras similares son mapeadas a vectores similares.
- Tienen problema con los homónimos, ya que no distinguen el sentido de una palabra en contexto. Por ejemplo, el embedding de banco es uno, independientemente si es institución financiera o un banco de plaza.
- Los embeddings pueden usarse para representar cualquier elemento de un conjunto discreto, por ejemplo en recomendación