

Word Embedding

- Second, map the one-hot vectors to low-dimensional vectors by

The diagram illustrates the mapping of a one-hot vector \mathbf{x}_i to a word embedding vector \mathbf{x}_i using a parameter matrix \mathbf{P}^T and a one-hot vector \mathbf{e}_i . The equation is represented as:

$$\mathbf{x}_i = \mathbf{P}^T \mathbf{e}_i$$

The dimensions of the matrices and vectors are indicated below them:

- \mathbf{x}_i is a $d \times 1$ vector (represented by a 2x1 grid of red squares).
- \mathbf{P}^T is a $d \times v$ matrix (represented by a 2x6 grid of colored squares: green, grey, red, blue, yellow, and red).
- \mathbf{e}_i is a $v \times 1$ vector (represented by a 6x1 grid of squares with values 0, 0, 1, 0, 0, 0, where the 1 is highlighted in grey).

- \mathbf{P} is parameter matrix which can be learned from training data.
- \mathbf{e}_i is the one-hot vector of the i -th word in dictionary.

```

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

embedding_dim = 8

model.add(Embedding(vocabulary, embedding_dim, input_length=word_num))

```

$v = 10K$ $d = 8$ $= 20$

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 20, 8)	80000

$\text{word_num} \times \text{embedding_dim}$

```

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

embedding_dim = 8

model = Sequential()
model.add(Embedding(vocabulary, embedding_dim, input_length=word_num))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

model.summary()

```

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		

```

from keras import optimizers

epochs = 50

model.compile(optimizer=optimizers.RMSprop(lr=0.0001),
              loss='binary_crossentropy', metrics=['acc'])

```

```

from keras import optimizers

epochs = 50

model.compile(optimizer=optimizers.RMSprop(lr=0.0001),
              loss='binary_crossentropy', metrics=['acc'])
history = model.fit(x_train, y_train, epochs=epochs,
                   batch_size=32, validation_data=(x_valid, y_valid))

```

- The training set is randomly split to a training set and a validation set.
- 80% for training and 20% for validation.
- x_train: $20,000 \times 20$ matrix
- X_valid: $5,000 \times 20$ matrix

Performance on test set

```

loss_and_acc = model.evaluate(x_test, labels_test)
print('loss = ' + str(loss_and_acc[0]))
print('acc = ' + str(loss_and_acc[1]))

```

```

25000/25000 [=====] - 0s 18us/step
loss = 0.5025235502243042
acc = 0.74928

```

- About 75% accuracy on the test set.
- Not bad, because we use only the last 20 words in each movie review. (word_num=20)

Logistic Regression for Sentiment Analysis

`seqs[i]:`

[27, 28, 29, 30, 31, 22, 32, 33, 34, 35, 1, 36, 29, 37, 38, 39, 40, 2, 41, 42]

10,000×8 parameters

Embedding Layer

`X[i]:`

`word_num × embedding_dim (20×8) matrix`

Flatten Layer

`x[i]:`

160-dim **vector**

161 parameters

Logistic Regression

`f[i]:`

Binary Prediction (positive or negative)