Implementing Deep Learning Methods and Feature Engineering for Text Data: The Skip-gram Model

<= Previous post
Next post =>

Tags: Deep Learning, Feature Engineering, NLP, Python, Text Mining, Word Embeddings

Just like we discussed in the CBOW model, we need to model this Skip-gram architecture now as a deep learning classification model such that we take in the target word as our input and try to predict the context words.

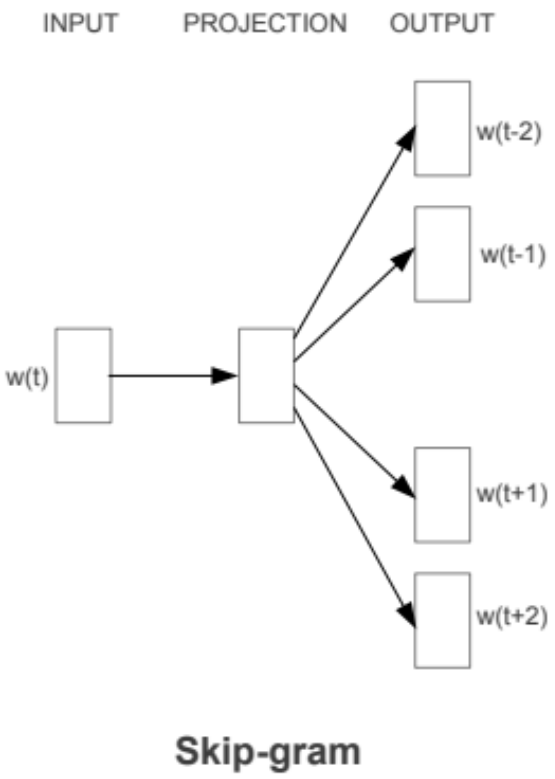
By Dipanjan Sarkar, RedHat.

comments

> **Editor's note:** This post is only one part of a far more thorough and in-depth original, found here, which covers much more than what is included here.

The Skip-gram Model

The Skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. It tries to predict the source context words (surrounding words) given a target word (the center word). Considering our simple sentence from earlier, ***"the quick brown fox jumps over the lazy dog".*** If we used the CBOW model, we get pairs of ***(context_window, target_word)***where if we consider a context window of size 2, we have examples like ***([quick, fox], brown), ([the, brown], quick), ([the, dog], lazy)*** and so on. Now considering that the skip-gram model's aim is to predict the context from the target word, the model typically inverts the contexts and targets, and tries to predict each context word from its target word. Hence the task becomes to predict the context ***[quick, fox]*** given target word ***'brown'*** or ***[the, brown]*** given target word ***'quick'*** and so on. Thus the model tries to predict the context_window words based on the target_word.



The Skip-gram model architecture (Source: https://arxiv.org/pdf/1301.3781.pdf Mikolov el al.)

Just like we discussed in the CBOW model, we need to model this Skip-gram architecture now as a deep learning classification model such that we take in the *target word as our input* and try to *predict the context words*.This becomes slightly complex since we have multiple words in our context. We simplify this further by breaking down each ***(target, context_words) pair*** into ***(target, context) pairs*** such that each context consists of only one word. Hence our dataset from earlier gets transformed into pairs like ***(brown, quick), (brown, fox), (quick, the), (quick, brown)*** and so on. But how to supervise or train the model to know what is contextual and what is not?

For this, we feed our skip-gram model pairs of ***(X, Y)*** where ***X*** is our ***input*** and ***Y*** is our ***label***. We do this by using ***[(target, context), 1]*** pairs as ***positive input samples*** where ***target*** is our word of interest and ***context*** is a context word occurring near the target word and the ***positive label 1*** indicates this is a contextually relevant pair. We also feed in ***[(target, random), 0]*** pairs as ***negative input samples*** where ***target*** is again our word of interest but ***random*** is just a randomly selected word from our vocabulary which has no context or association with our target word. Hence the ***negative label 0***indicates this is a contextually irrelevant pair. We do this so that the model can then learn which pairs of words are contextually relevant and which are not and generate similar embeddings for semantically similar words.

Implementing the Skip-gram Model

Let's now try and implement this model from scratch to gain some perspective on how things work behind the scenes and also so that we can compare it with our implementation of the CBOW model. We will leverage our Bible corpus as usual which is contained in the **`norm_bible`** variable for training our model. The implementation will focus on five parts

- **Build the corpus vocabulary**
- **Build a skip-gram [(target, context), relevancy] generator**
- **Build the skip-gram model architecture**
- **Train the Model**
- **Get Word Embeddings**

Let's get cracking and build our skip-gram Word2Vec model!

Build the corpus vocabulary

To start off, we will follow the standard process of building our corpus vocabulary where we extract out each unique word from our vocabulary and assign a unique identifier, similar to what we did in the CBOW model. We also maintain mappings to transform words to their unique identifiers and vice-versa.

```
from keras.preprocessing import text

tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(norm_bible)

word2id = tokenizer.word_index
id2word = {v:k for k, v in word2id.items()}

vocab_size = len(word2id) + 1
embed_size = 100

wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in norm_bible]
print('Vocabulary Size:', vocab_size)
print('Vocabulary Sample:', list(word2id.items())[:10])
```

feature_engg_text_21.py hosted with ♡ by GitHub view raw

```
Vocabulary Size: 12425
Vocabulary Sample: [('perceived', 1460), ('flagon', 7287), ('gardener', 11641), ('named', 973), ('re
```

Just like we wanted, each unique word from the corpus is a part of our vocabulary now with a unique numeric identifier.

Build a skip-gram [(target, context), relevancy] generator

It's now time to build out our skip-gram generator which will give us pair of words and their relevance like we discussed earlier. Luckily, `keras` has a nifty `skipgrams` utility which can be used and we don't have to manually implement this generator like we did in CBOW.

> **Note:** The function **skipgrams(…)** is present in **keras.preprocessing.sequence**
>
> This function transforms a sequence of word indexes (list of integers) into tuples of words of the form:
> - (word, word in the same window), with label 1 (positive samples).
> - (word, random word from the vocabulary), with label 0 (negative samples).

```
from keras.preprocessing.sequence import skipgrams

# generate skip-grams
skip_grams = [skipgrams(wid, vocabulary_size=vocab_size, window_size=10) for wid in wids]

# view sample skip-grams
pairs, labels = skip_grams[0][0], skip_grams[0][1]
for i in range(10):
    print("({:s} ({:d}), {:s} ({:d})) -> {:d}".format(
          id2word[pairs[i][0]], pairs[i][0],
          id2word[pairs[i][1]], pairs[i][1],
          labels[i]))
```

feature_engg_text_22.py hosted with ♡ by GitHub view raw

```
(james (1154), king (13)) -> 1
(king (13), james (1154)) -> 1
(james (1154), perform (1249)) -> 0
(bible (5766), dismissed (6274)) -> 0
(king (13), alter (5275)) -> 0
(james (1154), bible (5766)) -> 1
(king (13), bible (5766)) -> 1
(bible (5766), king (13)) -> 1
(king (13), compassion (1279)) -> 0
(james (1154), foreskins (4844)) -> 0
```

Thus you can see we have successfully generated our required skip-grams and based on the sample skip-grams in the preceding output, you can clearly see what is relevant and what is irrelevant based on the label (0 or 1).

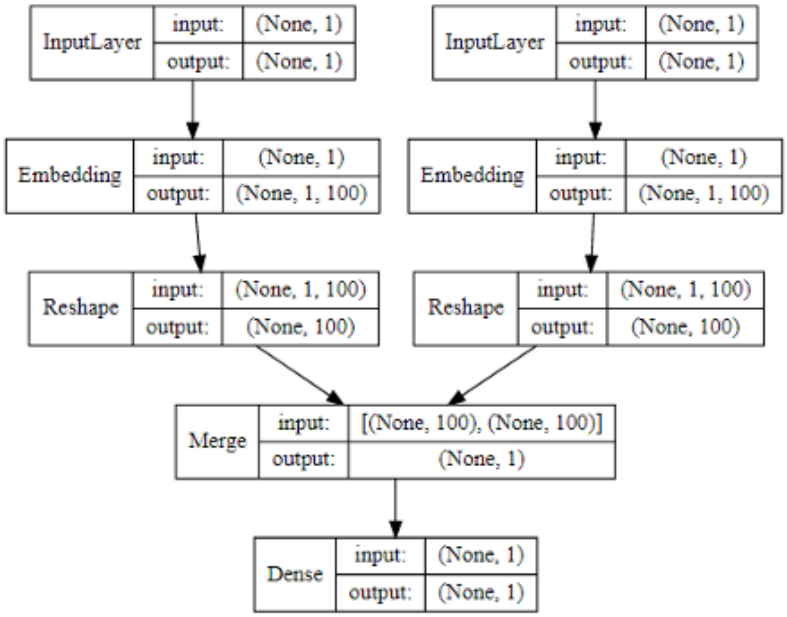
Build the skip-gram model architecture

We now leverage `keras` on top of `tensorflow` to build our deep learning architecture for the skip-gram model. For this our inputs will be our target word and context or random word pair. Each of which are passed to an embedding layer (initialized with random weights) of it's own. Once we obtain the word embeddings for the target and the context word, we pass it to a merge layer where we compute the dot product of these two vectors. Then we pass on this dot product value to a dense sigmoid layer which predicts either a 1 or a 0 depending on if the pair of words are contextually relevant or just random words (***Y'***). We match this with the actual relevance

label (***Y***), compute the loss by leveraging the `mean_squared_error` loss and perform backpropagation with each epoch to update the embedding layer in the process. Following code shows us our model architecture.

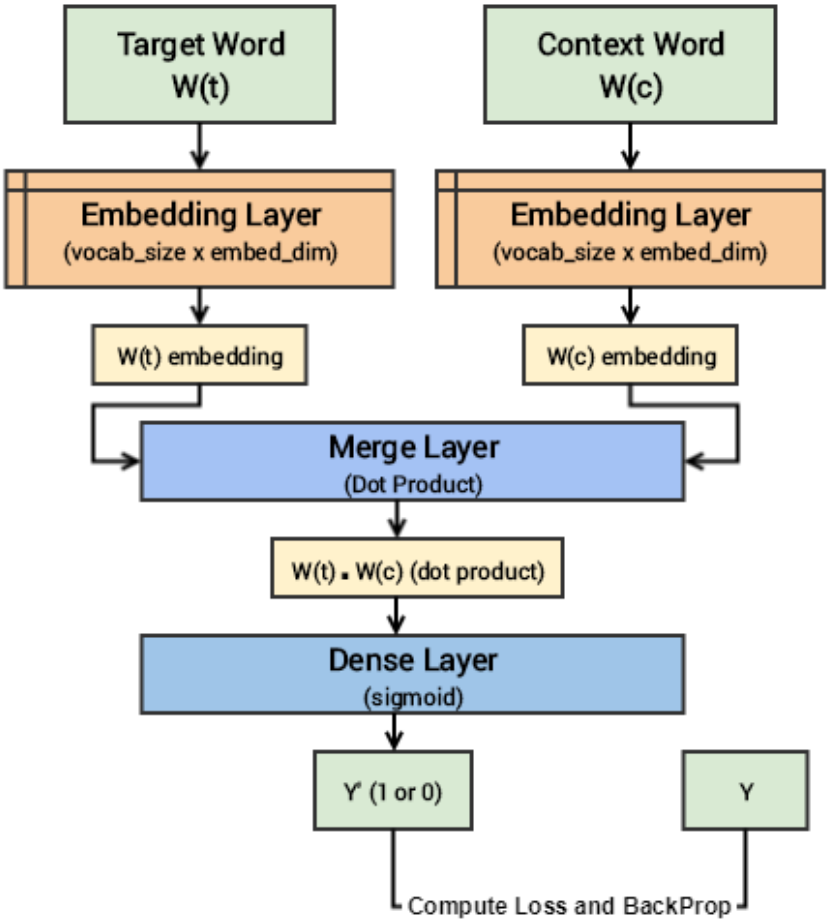
```
from keras.layers import Merge
from keras.layers.core import Dense, Reshape
from keras.layers.embeddings import Embedding
from keras.models import Sequential

# build skip-gram architecture
word_model = Sequential()
word_model.add(Embedding(vocab_size, embed_size,
                         embeddings_initializer="glorot_uniform",
                         input_length=1))
word_model.add(Reshape((embed_size, )))

context_model = Sequential()
context_model.add(Embedding(vocab_size, embed_size,
                  embeddings_initializer="glorot_uniform",
                  input_length=1))
context_model.add(Reshape((embed_size,)))

model = Sequential()
model.add(Merge([word_model, context_model], mode="dot"))
model.add(Dense(1, kernel_initializer="glorot_uniform", activation="sigmoid"))
model.compile(loss="mean_squared_error", optimizer="rmsprop")

# view model summary
print(model.summary())

# visualize model structure
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model, show_shapes=True, show_layer_names=False,
                 rankdir='TB').create(prog='dot', format='svg'))
```

```
____________________________________________________
Layer (type)                 Output Shape  Param #
====================================================
merge_2 (Merge)              (None, 1)     0
____________________________________________________
dense_3 (Dense)              (None, 1)     2
====================================================
Total params: 2,485,002
Trainable params: 2,485,002
Non-trainable params: 0
____________________________________________________
```



Skip-gram model summary and architecture

Understanding the above deep learning model is pretty straightforward. However, I will try to summarize the core concepts of this model in simple terms for ease of understanding. We have a pair of input words for each training example consisting of ***one input target word*** having a unique numeric identifier and ***one context word*** having a unique numeric identifier. If it is ***a positive sample*** the word has contextual meaning, is ***a context word***and our ***label Y=1***, else if it is a ***negative sample***, the word has no contextual meaning, is just ***a random word*** and our ***label Y=0***. We will pass each of them to an ***embedding layer*** of their own, having size **`(vocab_size x embed_size)`** which will give us ***dense word embeddings*** for each of these two words **`(1 x embed_size for each word)`**. Next up we use a ***merge layer*** to compute the ***dot product*** of these two embeddings and get the dot product value. This is then sent to the ***dense sigmoid layer*** which outputs either a 1 or 0. We compare this with the actual label Y (1 or 0), compute the loss, backpropagate the errors to adjust the weights (in the embedding layer) and repeat this process for all ***(target, context)*** pairs for multiple epochs. The following figure tries to explain the same.



Visual depiction of the Skip-gram deep learning model

Let's now start training our model with our skip-grams.

Train the Model

Running the model on our complete corpus takes a fair bit of time but lesser than the CBOW model. Hence I just ran it for 5 epochs. You can leverage the following code and increase it for more epochs if necessary.

```
for epoch in range(1, 6):
    loss = 0
    for i, elem in enumerate(skip_grams):
        pair_first_elem = np.array(list(zip(*elem[0]))[0], dtype='int32')
        pair_second_elem = np.array(list(zip(*elem[0]))[1], dtype='int32')
        labels = np.array(elem[1], dtype='int32')
        X = [pair_first_elem, pair_second_elem]
        Y = labels
        if i % 10000 == 0:
            print('Processed {} (skip_first, skip_second, relevance) pairs'.format(i))
        loss += model.train_on_batch(X,Y)

    print('Epoch:', epoch, 'Loss:', loss)
```

feature_engg_text_24.py hosted with ♡ by GitHub view raw

```
Epoch: 1 Loss: 4529.63803683
Epoch: 2 Loss: 3750.71884749
Epoch: 3 Loss: 3752.47489296
Epoch: 4 Loss: 3793.9177565
Epoch: 5 Loss: 3716.07605051
```

Once this model is trained, similar words should have similar weights based off the embedding layer and we can test out the same.

Get Word Embeddings

To get word embeddings for our entire vocabulary, we can extract out the same from our embedding layer by leveraging the following code. Do note that we are only interested in the target word embedding layer, hence we will extract the embeddings from our **`word_model`** embedding layer. We don't take the embedding at position 0 since none of our words in the vocabulary have a numeric identifier of 0 and we ignore it.

```
merge_layer = model.layers[0]
word_model = merge_layer.layers[0]
word_embed_layer = word_model.layers[0]
weights = word_embed_layer.get_weights()[0][1:]

print(weights.shape)
pd.DataFrame(weights, index=id2word.values()).head()
```

feature_engg_text_25.py hosted with ♡ by GitHub view raw

```
(12424, 100)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93
shall	-0.002272	0.015870	0.018349	0.022802	0.028364	-0.040064	-0.013263	0.136607	0.019667	0.033407	...	0.037663	-0.087140	0.073169	-0.028257
unto	0.034425	-0.102070	0.018051	0.017960	0.172954	-0.115672	-0.012632	0.096919	-0.049203	-0.040344	...	0.106373	-0.075703	0.013888	-0.134224
lord	0.051990	-0.113865	0.007226	0.031754	0.052963	-0.094523	-0.067664	0.001706	-0.112827	-0.078586	...	-0.041636	0.053685	0.041299	-0.026255
thou	-0.152183	-0.073681	-0.091472	0.022033	0.008415	-0.048438	-0.041181	0.082019	0.004648	0.044870	...	0.101531	-0.018404	-0.070462	-0.041363
thy	-0.257579	-0.023008	0.053303	0.013690	-0.083293	0.034279	0.078811	0.079851	-0.015215	-0.111211	...	-0.064527	0.112085	0.061625	0.026398

5 rows × 100 columns

Word Embeddings for our vocabulary based on the Skip-gram model

Thus you can clearly see that each word has a dense embedding of size **`(1x100)`** as depicted in the preceding output similar to what we had obtained from the CBOW model. Let's now apply the euclidean distance metric on these dense embedding vectors to generate a pairwise distance metric for each word in our vocabulary. We can then find out the n-nearest neighbors of each word of interest based on the shortest (euclidean) distance similar to what we did on the embeddings from our CBOW model.

```
from sklearn.metrics.pairwise import euclidean_distances

distance_matrix = euclidean_distances(weights)
print(distance_matrix.shape)

similar_words = {search_term: [id2word[idx] for idx in distance_matrix[word2id[search_term]-1].argsort()
```

```
                    for search_term in ['god', 'jesus', 'noah', 'egypt', 'john', 'gospel', 'moses','famin

similar_words
```

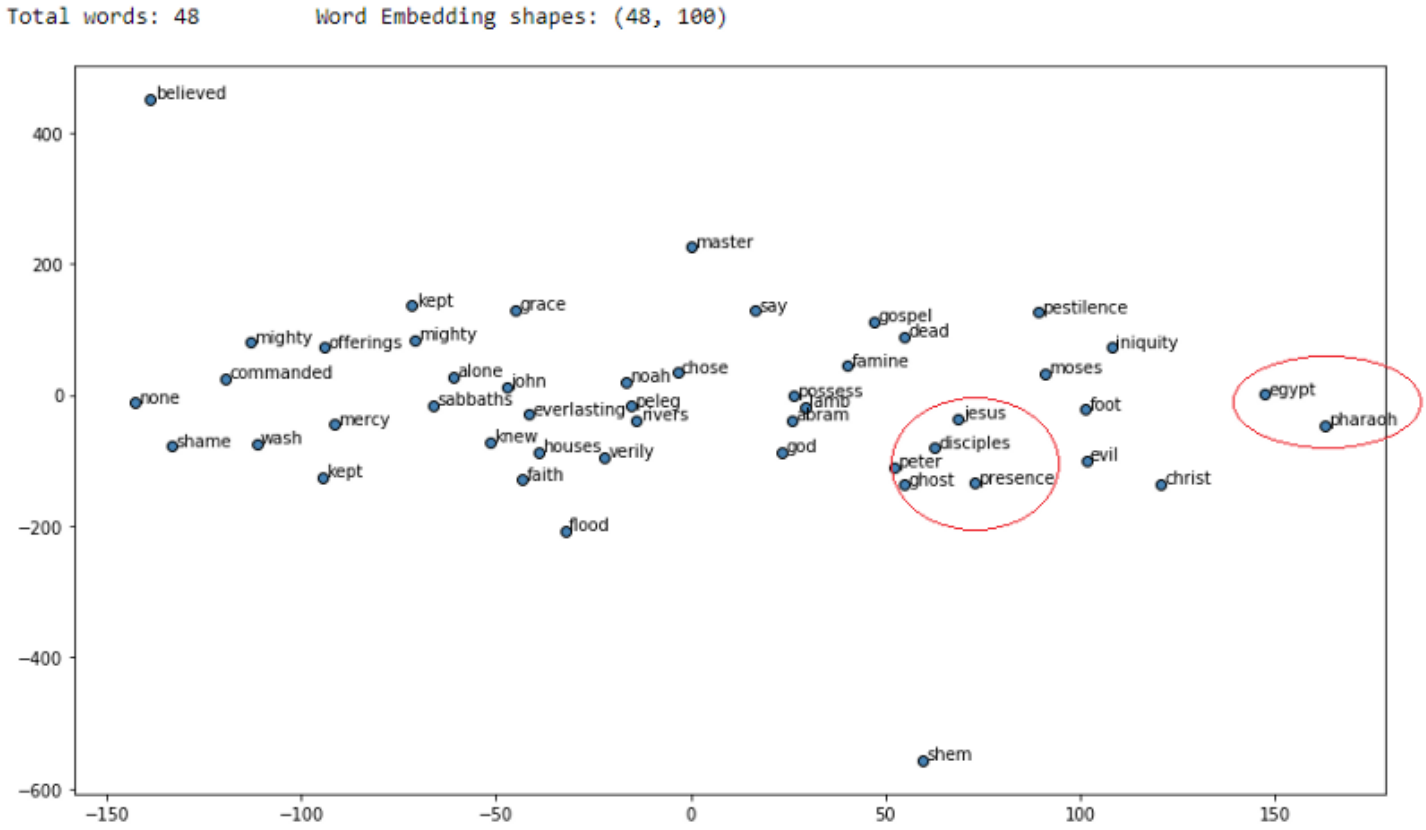
feature_engg_text_26.py hosted with ♡ by GitHub view raw

```
(12424, 12424)

{'egypt': ['pharaoh', 'mighty', 'houses', 'kept', 'possess'],
 'famine': ['rivers', 'foot', 'pestilence', 'wash', 'sabbaths'],
 'god': ['evil', 'iniquity', 'none', 'mighty', 'mercy'],
 'gospel': ['grace', 'shame', 'believed', 'verily', 'everlasting'],
 'jesus': ['christ', 'faith', 'disciples', 'dead', 'say'],
 'john': ['ghost', 'knew', 'peter', 'alone', 'master'],
 'moses': ['commanded', 'offerings', 'kept', 'presence', 'lamb'],
 'noah': ['flood', 'shem', 'peleg', 'abram', 'chose']}
```

You can clearly see from the results that a lot of the similar words for each of the words of interest are making sense and we have obtained better results as compared to our CBOW model. Let's visualize these words embeddings now using **t-SNE** which stands for ***t-distributed stochastic neighbor embedding*** a popular dimensionality reduction technique to visualize higher dimension spaces in lower dimensions (e.g. 2-D).

```
from sklearn.manifold import TSNE

words = sum([[k] + v for k, v in similar_words.items()], [])
words_ids = [word2id[w] for w in words]
word_vectors = np.array([weights[idx] for idx in words_ids])
print('Total words:', len(words), '\tWord Embedding shapes:', word_vectors.shape)

tsne = TSNE(n_components=2, random_state=0, n_iter=10000, perplexity=3)
np.set_printoptions(suppress=True)
T = tsne.fit_transform(word_vectors)
labels = words

plt.figure(figsize=(14, 8))
plt.scatter(T[:, 0], T[:, 1], c='steelblue', edgecolors='k')
for label, x, y in zip(labels, T[:, 0], T[:, 1]):
    plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points')
```

feature_engg_text_27.py hosted with ♡ by GitHub view raw



Visualizing skip-gram word2vec word embeddings using t-SNE

I have marked some circles in red which seemed to show different words of contextual similarity positioned near each other in the vector space. If you find any other interesting patterns feel free to let me know!

Bio: Dipanjan Sarkar is a Data Scientist @Intel, an author, a mentor @Springboard, a writer, and a sports and sitcom addict.

Original. Reposted with permission.

Related:

- Text Data Preprocessing: A Walkthrough in Python
- A General Approach to Preprocessing Text Data
- A Framework for Approaching Textual Data Science Tasks

Bora • 9 months ago

This model uses a separate embedding for each input. In word2vec and siamese type networks, embedding is shared. Can you give the idea behind your decision?

Reply •

Anoop Mudhollkar • a year ago

Can i know which version of keras is used here?

Reply •

Dieter Verbeemen • a year ago

Hey, I've a question about the architecture of the skip-gram model...
Because, now you're using 2 inputs, each has an embedding & then you connect them with the dot product...
But when you lookup some other tutorials, or explanations, you see that those architectures are diffferent...
the input layer contains 10.000 nodes (if you've 10.000words) and the output layer does also contain 10.000 nodes, but you're really trying to predict what the next words would be
E.g. input [0 0 0 1 0 0 0 ... 0] output [1 0 0 1 0 1 0 0 ... 0] etc
(https://cdn-images-1.medium...
Thus my question is, why is this network different? is it because of the negative sampling?

Reply •

Osama Salah • 2 years ago

Well, we got lists of relevant words. How can we get the context given a single word?

Reply •

<= Previous post
Next post =>

Top Stories Past 30 Days

Most Popular

1. **24 Best (and Free) Books To Understand Machine Learning**
2. **COVID-19 Visualized: The power of effective visualizations for pandemic storytelling**
3. **How (not) to use Machine Learning for time series forecasting: The sequel**
4. **50 Must-Read Free Books For Every Data Scientist in 2020**
5. **Free Mathematics Courses for Data Science & Machine Learning**
6. **Nine lessons learned during my first year as a Data Scientist**
7. **New Poll: Coronavirus impact on AI/Data Science/Machine Learning community**

Most Shared

1. **24 Best (and Free) Books To Understand Machine Learning**
2. **COVID-19 Visualized: The power of effective visualizations for pandemic storytelling**
3. **Introducing MIDAS: A New Baseline for Anomaly Detection in Graphs**
4. **Covid-19, your community, and you a data science perspective**
5. **How (not) to use Machine Learning for time series forecasting: The sequel**
6. **50 Must-Read Free Books For Every Data Scientist in 2020**
7. **Coronavirus Data and Poll Analysis – yes, there is hope, if we act now**

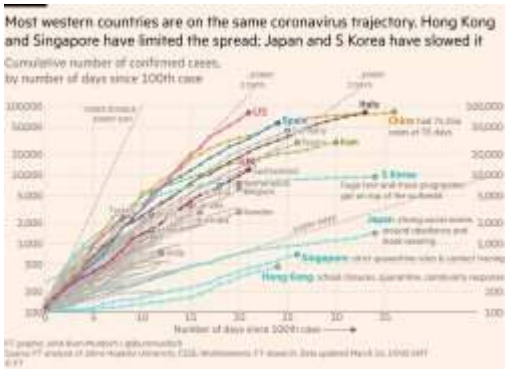
Latest News

- KNIME Spring Summit Online Edition
- Upcoming Webinars and online events in AI, Data Science...
- Has AI Come Full Circle? A data science journey, or wh...
- Successful Use Cases of Artificial Intelligence for Bus...
- How Data Science Is Being Used to Understand COVID-19
- Statistical Thinking for Industrial Problem Solving □...

Top Stories
Last Week

Most Popular

1. **COVID-19 Visualized: The power of effective visualizations for pandemic storytelling**



2. **How (not) to use Machine Learning for time series forecasting: The sequel**
3. **Stop Hurting Your Pandas!**
4. **Research into 1,001 Data Scientist LinkedIn Profiles, the latest**

5. **24 Best (and Free) Books To Understand Machine Learning**
6. **Best Free Epidemiology Courses for Data Scientists**
7. **Python for data analysis... is it really that simple?!?**

Most Shared

1. **Introducing MIDAS: A New Baseline for Anomaly Detection in Graphs**
2. **How (not) to use Machine Learning for time series forecasting: The sequel**
3. **Best Free Epidemiology Courses for Data Scientists**
4. **Research into 1,001 Data Scientist LinkedIn Profiles, the latest**
5. **More Performance Evaluation Metrics for Classification Problems You Should Know**
6. **Advice for a Successful Data Science Career**
7. **Introduction to the K-nearest Neighbour Algorithm Using Examples**

More Recent Stories

- Statistical Thinking for Industrial Problem Solving – a ...
- 3 Best Sites to Find Datasets for your Data Science Projects
- Build PyTorch Models Easily Using torchlayers
- Top March stories: 24 Best (and Free) Books To Understand Mach...
- 10 Must-read Machine Learning Articles (March 2020)
- Top tweets, Apr 01-07: How to change global policy on #coro...
- How to Do Hyperparameter Tuning on Any Python Script in 3 Easy...
- TensorFlow Dev Summit 2020: Top 10 Tricks for TensorFlow and G...
- 3 Reasons to Use Random Forest® Over a Neural Network: Com...
- KDnuggets 20:n14, Apr 8: Free Mathematics for Machine Learn...
- 2 Things You Need to Know about Reinforcement Learning – Com...
- Simple Question Answering (QA) Systems That Use Text Similarit...
- Build an app to generate photorealistic faces using TensorFlow...
- 5 Ways Data Scientists Can Help Respond to COVID-19 and 5 Acti...
- Uber Open Sourced Fiber, a Framework to Streamline Distributed...
- Top Stories, Mar 30 – Apr 5: COVID-19 Visualized: The po...
- Mathematics for Machine Learning: The Free eBook
- More Performance Evaluation Metrics for Classification Problem...
- Best Free Epidemiology Courses for Data Scientists
- Stop Hurting Your Pandas!