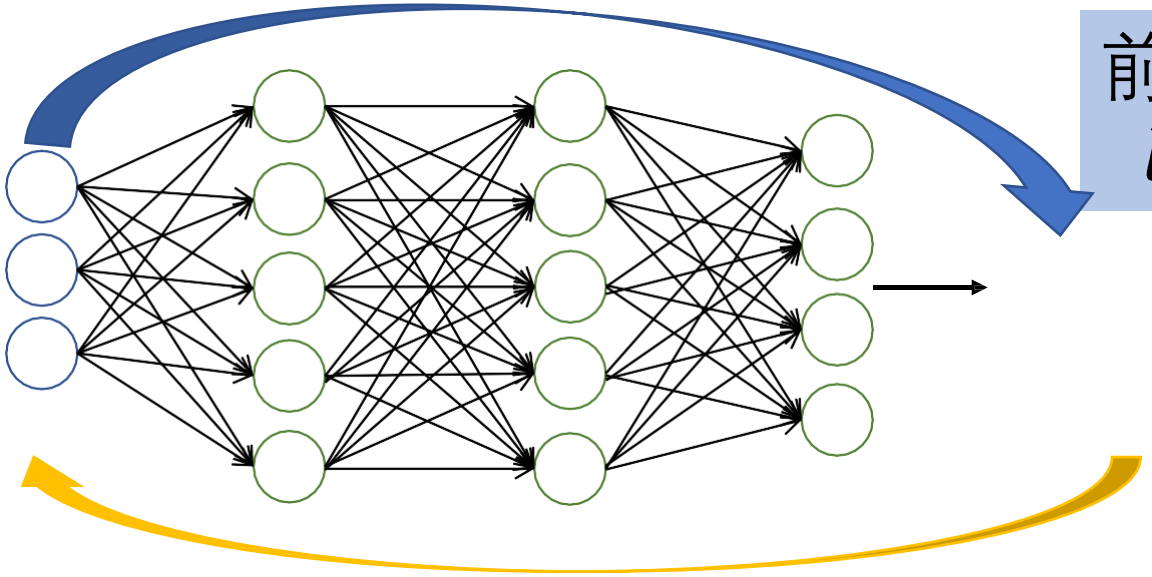
基于keras的多层感知机实现

(回顾) 多层感知机
Multi-Layer Perceptron(MLP)



前传计算输出和损失
$l(y, \hat{y}) = l(y, f_\theta(x))$

后传损失的梯度更新
模型参数θ

1.设计网络结构：层数、每层节点数、每层的激活函数
2.选择一个损失函数
3.设置学习率、优化器、Batch-size、其他超参数

- 如何确定输入/出层节点数?
- 常用激活函数有哪些?
- 分类一般用哪个损失函数?

课堂小练笔

- 深度学习为什么这么“火”？给出具体原因。
- 现在需要对某种数据（100维）用单隐层多层感知机进行分类（分成3个类别）。假设该隐藏层包含20个节点，给出网络结构示意图并计算总的参数量。
- 给出单个神经元节点从输入到输出的运算，假设连接其各个输入的权重为 $\boldsymbol{w}$, 输入为$\boldsymbol{x}$, 该节点的偏置为θ, 其中$\boldsymbol{w}$和$\boldsymbol{x}$为d维向量,激活函数用f表示。

基于MLP对Fashion MNIST数据分类

Keras

```
import tensorflow as tf
from tensorflow import keras
```

```
tf.__version__
```

```
'2.4.1'
```

```
keras.__version__
```

```
'2.4.0'
```

基于TensorFlow 2

tf.keras

Keras对各种底层张量库进行高层模块封装，可以用不同的后端 backend。

默认后端 TensorFlow

在keras.json文件的“backend”字段

想兼容不同后端?
用抽象的keras backend API提供的方法

from keras **import** backend **as** K

keras.datasets

Available datasets

MNIST digits classification dataset

- load_data function

CIFAR10 small images classification dataset

- load_data function

CIFAR100 small images classification dataset

- load_data function

图像数据集

IMDB movie review sentiment classification dataset

- load_data function
- get_word_index function

Reuters newswire classification dataset

- load_data function
- get_word_index function

文本数据集

Fashion MNIST dataset, an alternative to MNIST

- load_data function

图像数据集

Boston Housing price regression dataset

- load_data function

结构化数据集

Fashion MNIST数据分类

- 从Keras常用数据集keras.datasets导入

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

该数据集属于10个类别;
已经被分成训练集（6万）和测试集（1万）两部分;
每张图片为28*28像素的灰度图，灰度值属于（0，255）。

Fashion MNIST数据分类

- 查看数据集基本情况

```
y_train
```

```
array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)
```

最终得到的训练集为55000张，验证集5000张，测试集10000张。

此处，标签为标量，而不是one-hot向量

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

从训练集中再分出5000张作为验证集，并把像素值归一化到[0, 1]。

```
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

打印前面样本图像的代码

```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_plot', tight_layout=False)
plt.show()
```

用多层感知机(MLP)实现对以上数据的分类

- 构建多层感知机的结构

1. 隐藏层的层数、
2. 每层（输入、输出、隐藏层）节点数目、
3. 每层对应的激活函数

- 模型编译（损失函数、优化器、评价指标）
- 训练模型（超参数）
- 评估和测试

典型的用于分类的MLP结构

超参数	二分类 (Binary classification)	多分类（Multiclass classification）
输入节点数	一个特征一个输入节点	
隐藏层数目	根据具体问题，一般1-5层	
每个隐藏层节点数	根据具体问题，一般10-100	
隐藏层激活函数	Relu	
输出层节点数	1	一个类一个节点
输出层激活函数	Sigmoid/logistic	softmax
损失函数	交叉熵	

结构选择

- 输入层节点个数，由输入图像大小决定
- 两个隐藏层，分别包含300和100个节点
- 输出层：10个节点（10个类别）

- 隐藏层全部用Relu激活，输出层用softmax (多分类)

基于sequential API创建MLP网络结构

创建sequential 模型，然后从输入到输出一层一层往里添加

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

1.由于输入为二维图像，用**Flattern**层实现把二维转成1维向量，该层只做以上简单预处理，不包含任何要学习的参数。

　　也可以用**keras.layers.InputLayer**(input_shape=[28, 28])

2.两个隐藏层和输出层均为**Dense**层，其功能为实现加权求和，加偏置，再激活，即计算$f(\boldsymbol{W}^l\boldsymbol{a}^{l-1}+\boldsymbol{b}^l)$

基于sequential API创建MLP网络结构

除了一层一层加，还可以在创建sequential 模型时给出所有层

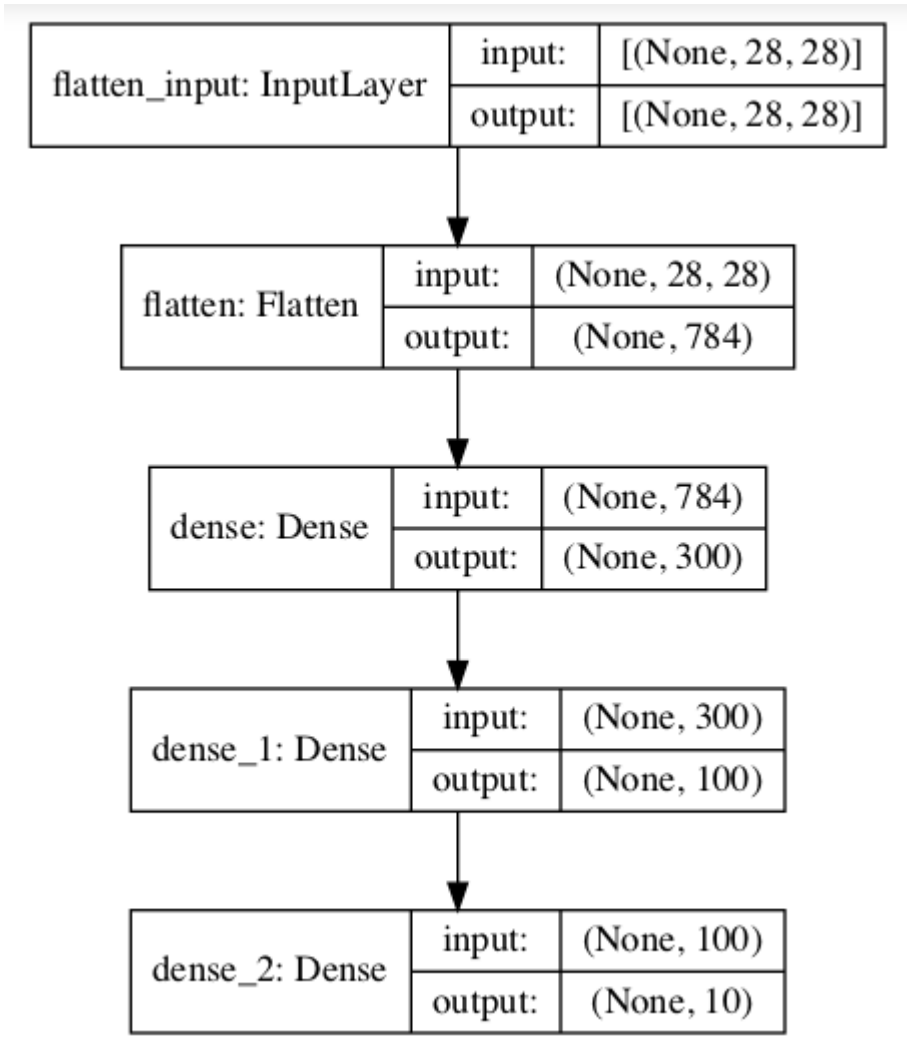
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

查看模型结构和参数

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

```
keras.utils.plot_model(model, "my_fashion_mnist_model.png", show_shapes=True)
```



查看初始化

```
weights, biases = hidden1.get_weights()
```

```
weights
```

```
]: array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
          0.03859074, -0.06889391],
        [ 0.00476504, -0.03105379, -0.0586676 , ...,  0.00602964,
         -0.02763776, -0.04165364],
        [-0.06189284, -0.06901957,  0.07102345, ..., -0.04238207,
          0.07121518, -0.07331658],
        ...,
        [-0.03048757,  0.02155137, -0.05400612, ..., -0.00113463,
          0.00228987,  0.05581069],
        [ 0.07061854, -0.06960931,  0.07038955, ..., -0.00384101,
          0.00034875,  0.02878492],
        [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
          0.00272203, -0.06793761]], dtype=float32)
```

```
biases
```

```
]: array([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., 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., 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., 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., 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., 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., 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., 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., 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., 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., 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., 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.], dtype=float32)
```

权重为初始化为随机值，偏置初始化为0。

模型编译

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

```
optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True)) 配制优化器
```

由于标签y表示为类别索引，而不是one-hot向量，所以这里的损失函数选择 sparse_categorical_crossentropy。
如果标签y表示为one-hot向量，那么应该选 categorical_crossentropy作为损失函数。

如果是二分类，最后一层激活用sigmoid而不是softmax，那么损失函数选binary_crossentropy。

模型训练

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

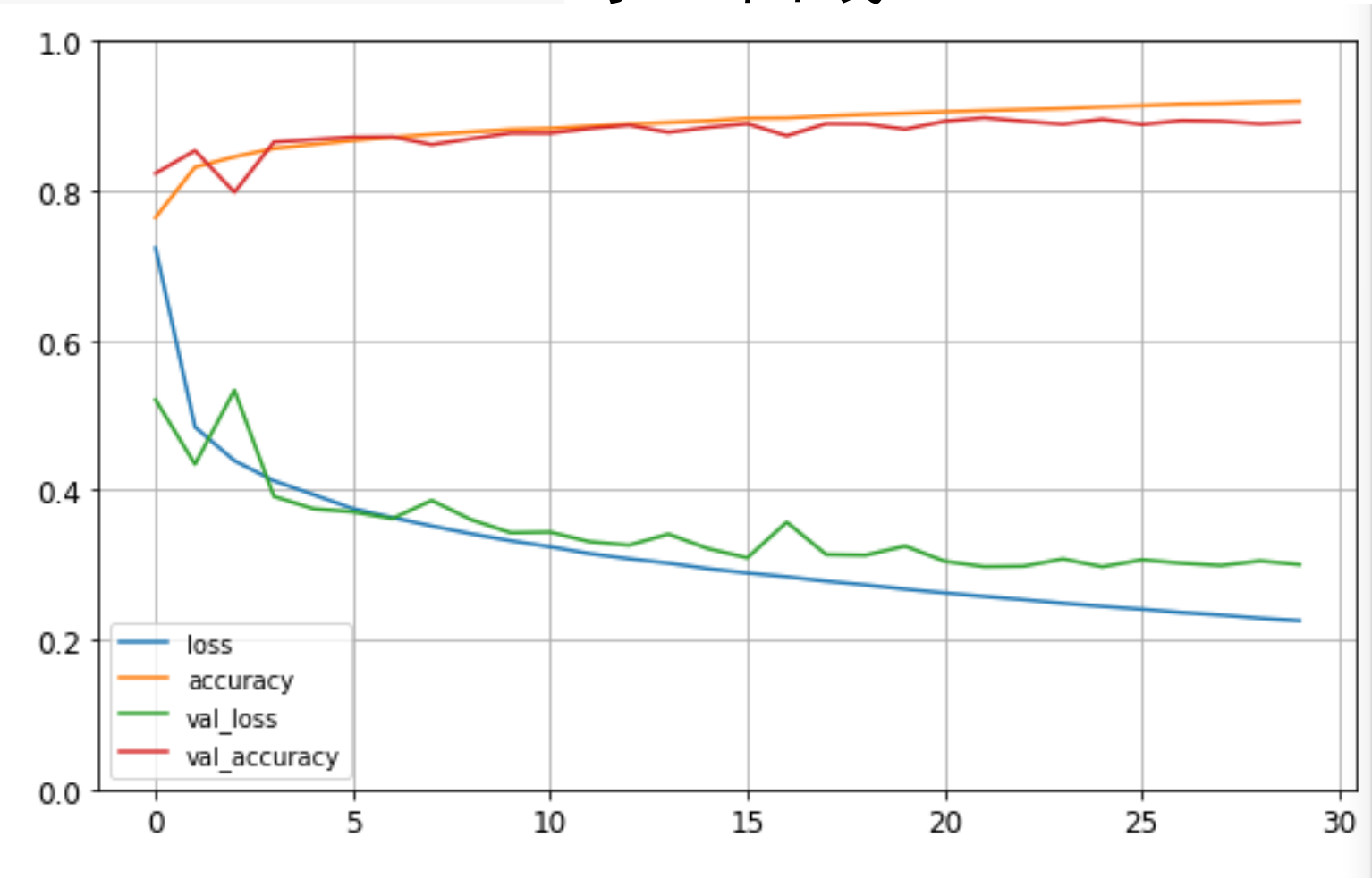
```
Epoch 1/30
1719/1719 [==============================] - 2s 1ms/step - loss: 1.0187 - accurac
y: 0.6807 - val_loss: 0.5207 - val_accuracy: 0.8234
Epoch 2/30
1719/1719 [==============================] - 2s 921us/step - loss: 0.5028 - accura
cy: 0.8260 - val_loss: 0.4345 - val_accuracy: 0.8538
Epoch 3/30
1719/1719 [==============================] - 2s 881us/step - loss: 0.4485 - accura
cy: 0.8423 - val_loss: 0.5334 - val_accuracy: 0.7982
Epoch 4/30
1719/1719 [==============================] - 2s 902us/step - loss: 0.4209 - accura
cy: 0.8535 - val_loss: 0.3916 - val_accuracy: 0.8652
```

可以通过设定batch_size改变每个batch的大小，默认32。

```
import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
save_fig("keras_learning_cur
plt.show()
```

学习曲线



从学习曲线看出，随着训练的进行，训练损失和验证损失大致都在减小，验证集准确率没有比训练集准确率低太多，说明模型应该没有过拟合。

测试模型

```
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 0s 639us/step - loss: 0.3357 - accuracy: 0.8837
```

```
]: [0.3357059359550476, 0.8837000131607056]
```

对前面三个测试样本查看具体结果

```
X_new = X_test[:3]
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
]: array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.03, 0.  , 0.96],
          [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
          [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
         dtype=float32)
```

```
y_new = y_test[:3]
y_new
```

```
array([9, 2, 1], dtype=uint8)
```

三个样本的预测结果与真实标签一致。

Dropout层

对需要dropout的层后面加上

keras.layers.Dropout(rate =0.2)

rate的值越大，丢弃的越多

一般采用dropout后收敛速度会变慢，但如果合理设置，一般会取得效果提升。

不一定每层都要用dropout，如果rate太大，可能会导致欠拟合（每层宽度太小）。

模型保存和载入

对于sequential 模型，用以下简单方法就可以保存模型结构以及模型参数，保存为HDF5格式。

```
model.save("my_keras_model.h5")
```

用下面方法载入保存的模型

```
model = keras.models.load_model("my_keras_model.h5")
```

训练过程中用callbacks实现保存和提前停止

ModelCheckpoint用于训练过程中在一定间隔（默认每个epoch）自动保存模型

只有在验证集上的表现是当前最好时，才保存模型： save_best_only = True

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5", save_best_only=True)
```

EarlyStopping用于提前停止，patience 表示等待验证集准确率没有改进的epoch数目。

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
```

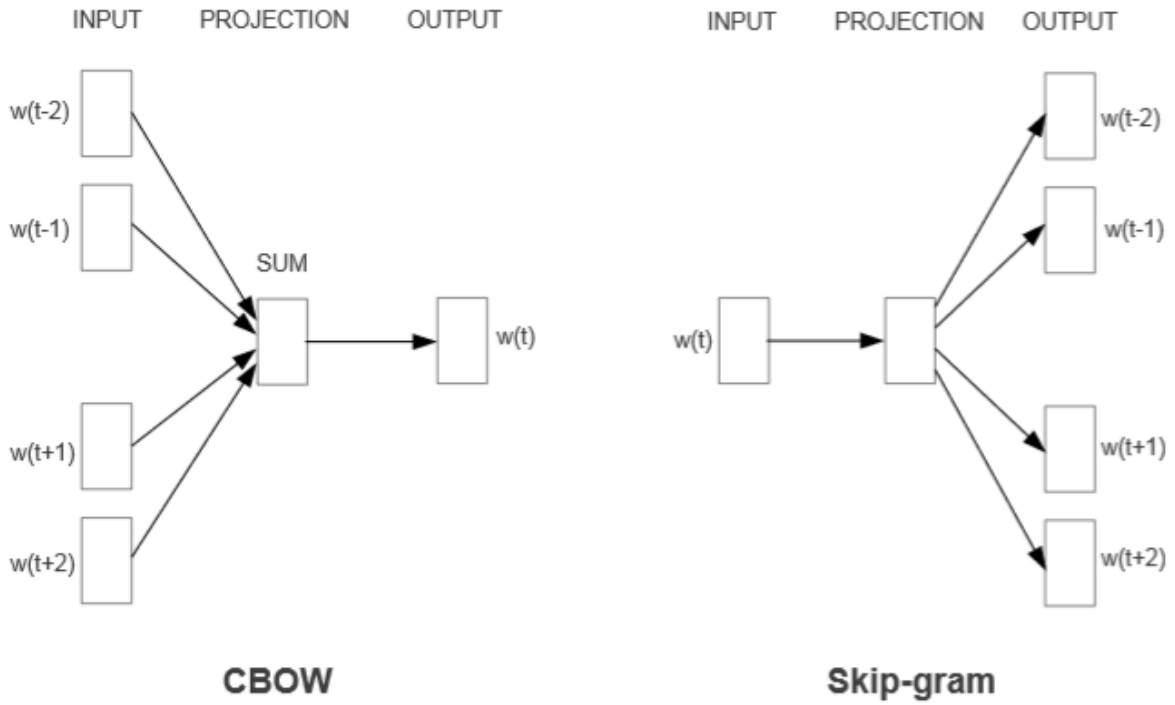
在model.fit（）里面使用设置好的callbacks。

```
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

Word2Vec-基于Keras的实现

Continuous Bag of Words (CBOW)

Skip-gram



CBOW基本实现步骤

建立词典索引：为每个token分配一个id

建立由中心词、上下文组成的（输入、输出）关系对

设定每一层的输入输出，包括映射层（降维）、lambda层（取平均）、全连接层（分类器）；

用（上下文，中心词）中的上下文作为输入，中心词作为标签。以预测结果和标签计算损失后，通过反向传播对模型参数进行更新优化。

训练结束后，从映射层输出词嵌入向量

bible数据集

```
from nltk.corpus import gutenberg
from string import punctuation
bible = gutenberg.sents('bible-kjv.txt')
#只抽取第1000到2000行
bible = bible[1000:2000]
remove_terms = punctuation + '0123456789'
norm_bible = [[word.lower() for word in sent if word not in remove_terms] for sent in bible]
norm_bible = [' '.join(tok_sent) for tok_sent in norm_bible]
norm_bible = [tok_sent for tok_sent in norm_bible if len(tok_sent.split()) > 2]

print('Total lines:', len(bible))
print('\nSample line:', bible[10])
print('\nProcessed line:', norm_bible[10])
```

对bibile-kjv进行分句，以列表赋值。
(kjv: King James version)

```
Total lines: 1000

Sample line: ['36', ':', '24', 'And', 'these', 'are', 'the', 'children', 'of', 'Zibeon', ';', 'both', 'Ajah',
',', 'and', 'Anah', ':', 'this', 'was', 'that', 'Anah', 'that', 'found', 'the', 'mules', 'in', 'the', 'wilder
ness', ',', 'as', 'he', 'fed', 'the', 'asses', 'of', 'Zibeon', 'his', 'father', '.']

Processed line: 36 24 and these are the children of zibeon both ajah and anah this was that anah that found t
he mules in the wilderness as he fed the asses of zibeon his father
```

基于keras实现CBOW-步骤1：建立词索引

```
from keras.preprocessing import text
from keras.utils import np_utils
from keras.preprocessing import sequence
tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(norm_bible)
word2id = tokenizer.word_index

# build vocabulary of unique words
word2id['PAD'] = 0
id2word = {v:k for k, v in word2id.items()}
wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in norm_bible]

vocab_size = len(word2id)
embed_size = 100
window_size = 2

print('Vocabulary Size:', vocab_size)
print('Vocabulary Sample:', list(word2id.items())[:10])
```

最开始左边窗口和结束时右边窗口用PAD填充，设定其id为0 。

```
Vocabulary Size: 2066
Vocabulary Sample: [('and', 1), ('the', 2), ('of', 3), ('in', 4), ('unto', 5), ('to', 6), ('his', 7), ('tha
t', 8), ('he', 9), ('lord', 10)]
```

基于keras实现CBOW-步骤2：建立(context，target）对

```
import numpy as np

def generate_context_word_pairs(corpus, window_size, vocab_size):
    context_length = window_size*2
    for words in corpus:
        sentence_length = len(words)
        for index, word in enumerate(words):
            context_words = []
            label_word   = []
            start = index - window_size
            end = index + window_size + 1

            context_words.append([words[i]
                                  for i in range(start, end)
                                  if 0 <= i < sentence_length
                                  and i != index])
            label_word.append(word)

            x = sequence.pad_sequences(context_words, maxlen=context_length)
            y = np_utils.to_categorical(label_word, vocab_size)
            yield (x, y)
```

基于keras实现CBOW-步骤2：建立(context，target）对

测试前面函数

```
# 测试前面的（上下文，中心词）产生函数
window_size=2
i = 0
for x, y in generate_context_word_pairs(corpus=wids, window_size=window_size, vocab_size=vocab_size):
    if 0 not in x[0]:
        print('Context (X):', [id2word[w] for w in x[0]], '-> Target (Y):', id2word[np.argwhere(y[0])[0][0]])

        if i == 2:
            break
        i += 1
```

只打印context words里面没有包含PAD的。X[0]为context words的id。

结果：

```
Context (X): ['36', 'and', 'was', 'concubine'] -> Target (Y): timna
Context (X): ['and', 'timna', 'concubine', 'to'] -> Target (Y): was
Context (X): ['timna', 'was', 'to', 'eliphaz'] -> Target (Y): concubine
```

CBOW神经网络模型

输入：以上下文(2n个词)作为输入传到映射层；每个输入为长度为vocab_size 的one-hot向量。

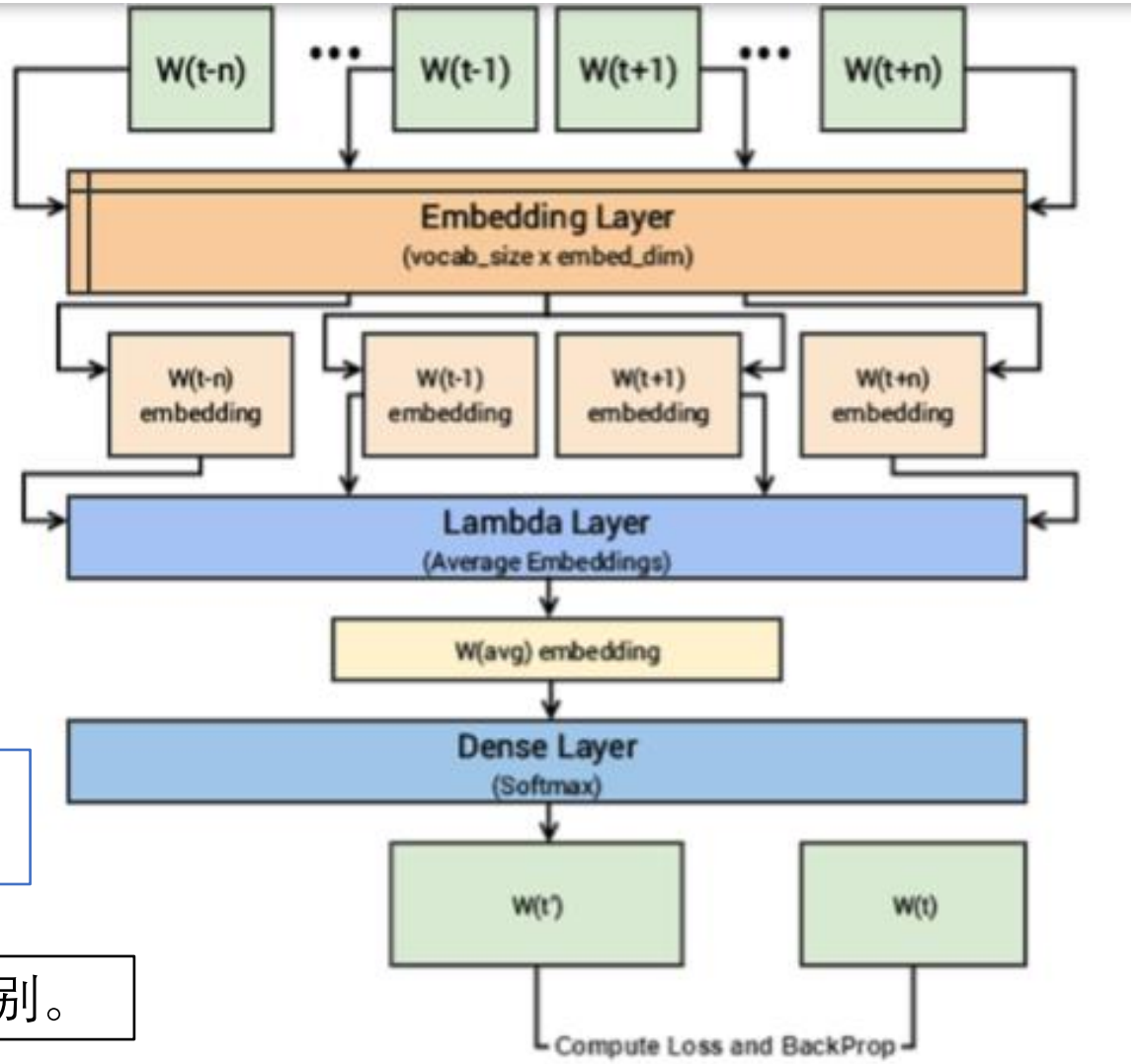
映射到embed_dim维空间

对上下文中所有词的映射向量取平均;

得到降维后的平均向量;

Softmax全连接层，得到预测后的vocab_size长度的向量，“升维“

交叉熵损失计算中心词的预测与真实的差别。



基于keras实现CBOW-步骤3：建立神经网络构建

```
import keras.backend as K
from keras.models import Sequential
from keras.layers import Dense, Embedding, Lambda

# 建立 CBOW 神经网络结构
cbow = Sequential()
cbow.add(Embedding(input_dim=vocab_size, output_dim=embed_size, input_length=window_size*2))
cbow.add(Lambda(lambda x: K.mean(x, axis=1), output_shape=(embed_size,)))
cbow.add(Dense(vocab_size, activation='softmax'))
cbow.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Embedding层：加权求和，也叫线性层。

Lambda层把表达式封装为Layer对象

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 4, 100)            206600
_________________________________________________________________
lambda (Lambda)              (None, 100)               0
_________________________________________________________________
dense (Dense)                (None, 2066)              208666
=================================================================
Total params: 415,266
Trainable params: 415,266
Non-trainable params: 0
```

```
print(cbow.summary())
```

2066（vocab_size）*100(embed_size)

权重100*2066+偏置2066

基于keras实现CBOW-步骤4：训练模型

- 用步骤2中得到的所有(上下文、中心词)对进行模型训练。

```
for epoch in range(1, 5):
    loss = 0.
    i = 0
    for x, y in generate_context_word_pairs(corpus=wids, window_size=window_size, vocab_size=vocab_size):
        i += 1
        loss += cbow.train_on_batch(x, y)
        if i % 100000 == 0:
            print('Processed {} (context, word) pairs'.format(i))

    print('Epoch:', epoch, '\tLoss:', loss)
    print()
```

train_on_batch手动地将每个batch的数据提供给模型

```
Epoch: 1        Loss: 179718.4568679426

Epoch: 2        Loss: 212429.9486697223

Epoch: 3        Loss: 210725.63478016388

Epoch: 4        Loss: 208425.4731876091
```

计算损失后，通过反向传播对模型参数进行更新优化。重复多个epoch。一个epoch代表所有数据过一遍。

CBOW得到词向量

#从embedding输出

```
import pandas as pd

weights = cbow.get_weights()[0]
weights = weights[1:]
print(weights.shape)

pd.DataFrame(weights, index=list(id2word.values())[1:]).head()
```

```
(2065, 100)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93	
the	0.276072	0.858237	-0.310999	-0.357828	0.533174	-0.124512	-0.491242	-0.502159	-0.430883	1.472598	...	0.603843	0.551637	-0.327865	-0.938406	0.
of	0.599945	0.525048	-0.813825	-0.135538	0.606239	-0.879153	-0.639023	-0.463408	-0.208838	-0.349287	...	0.938561	0.643156	-0.750471	0.024057	1.
in	0.480106	0.728034	-1.132503	-0.534229	0.342036	-0.420580	-0.505904	-0.303745	-0.581676	0.477676	...	0.187716	0.341084	0.833562	-0.786493	0.
unto	0.284021	0.534929	-0.448616	-0.477943	0.680894	-0.459638	-0.521146	-0.642003	-0.639177	0.276057	...	0.209853	0.469468	-0.715785	-0.552234	0.
to	-0.065055	0.041785	-0.261908	-0.116401	0.356854	-0.362174	-0.260104	-0.488839	-0.663701	0.784728	...	0.038225	0.453967	-0.180321	-0.351071	0.

5 rows × 100 columns

Word2Vec-两种方法

- Continuous Bag of Words (CBOW)

通过周边的词来预测中心词

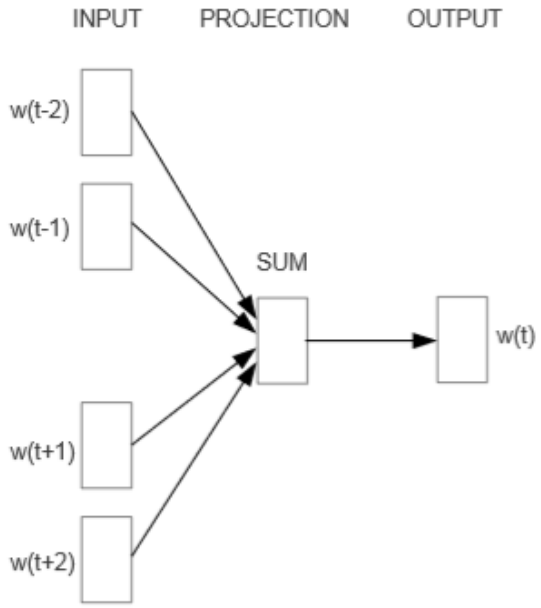
Skip-gram

用中心词来预测周边词

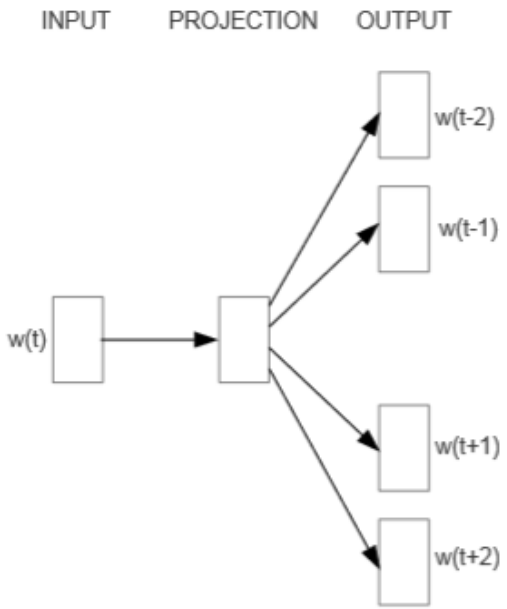
方法-CBOW:

窗口内的词在映射空间表示的平均得到中心词在该空间的映射。

取平均时并未考虑上下文中词的顺序。



CBOW



Skip-gram

- "Distributed Representations of Words and Phrases and their Compositionality" by Mikolov et al.
- "Efficient Estimation of Word Representations in Vector Space" by Mikolov et al

Skip-gram

基本思想：以 target word 作为输入，来预测窗口内的上下文。

主要步骤：

- 根据(target, context _ words)，得到windowsize*2个由该中心词到单个上下文词组成的 (target, context) 对。
- 把 所有(target, context) 对标记为正样本表示relevant，同时产生 (target, random) 作为负样本表示不相关。这里random表示一个从词库中随机选择的词。
- 通过上面的正负样本对模型进行训练，模型可以学习到哪些是上下文相关的词，哪些不是，使得相关词的词向量也相似。

在CBOW中是多分类（类别数目等于词汇总数），在Skip-gram是二分类（相关、不相关）

Skip-Gram [(target, context), relevancy]产生器

利用keras.preprocessing.sequence.中的skipgrams()函数得到以下形式的数据：

1. (word, word in the same window), with label 1 (positive samples).
2. (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_si
# 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]))
```

```
(esau (265), bare (225)) -> 1
(sons (71), as (61)) -> 0
(eliphaz (605), to (6)) -> 1
(to (6), silver (443)) -> 0
(bare (225), to (6)) -> 1
(of (3), prove (1963)) -> 0
(eliphaz (605), feed (614)) -> 0
(son (111), was (28)) -> 1
(to (6), oath (1649)) -> 0
(sons (71), she (96)) -> 1
```

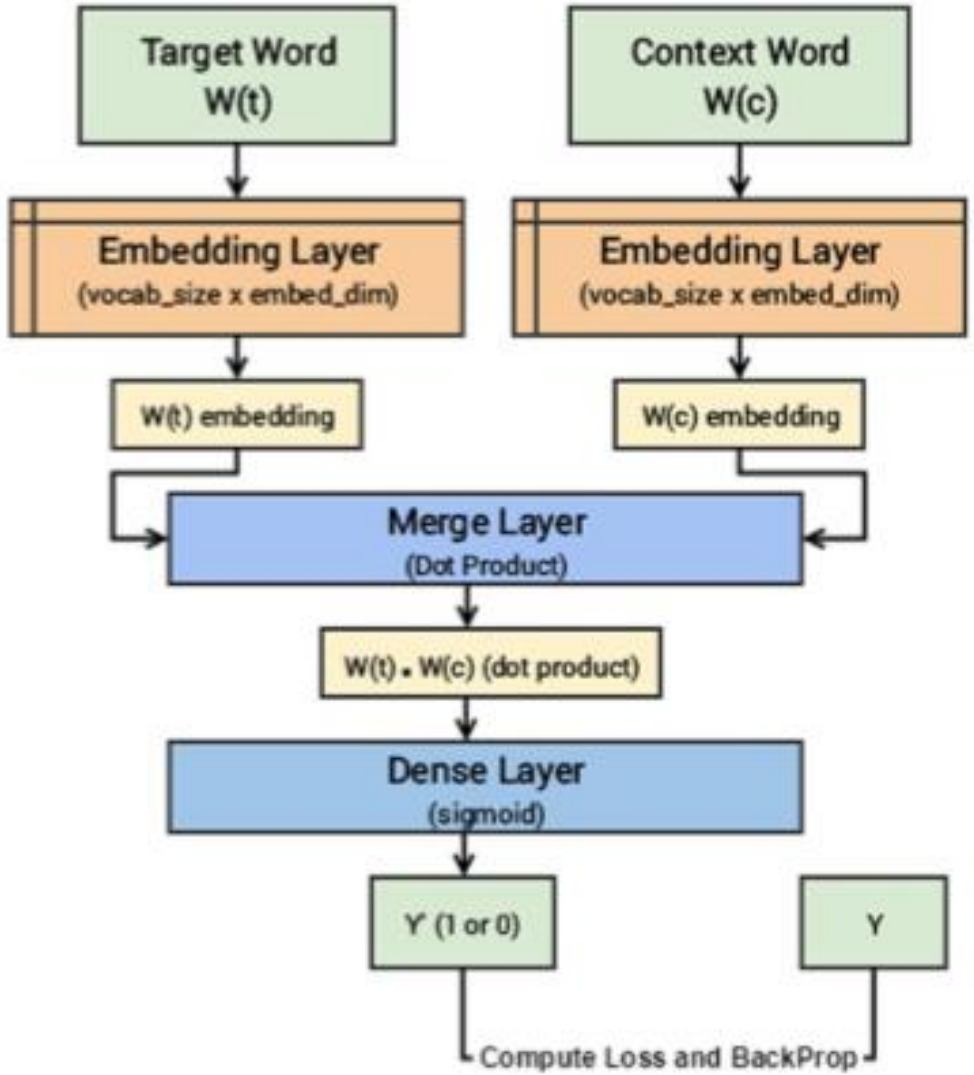
Skip-gram神经网络模型

把target和context/random分别输入到各自的embedding 层（随机初始化）

对得到的映射空间内的向量求点积。

通过Sigmoid层得到预测结果0或1，代表输入的两个词是否相关。

计算均方误差（MSE)损失，反向传播更新网络参数。



有2个embedding层

Skip-gram模型构架设置

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

# 建立skip-gram 网络结构
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_arch = Dot(axes=1)([word_model.output, context_model.output])
model_arch = Dense(1, kernel_initializer="glorot_uniform",activation="sigmoid")(model_arch)
model = Model([word_model.input,context_model.input], model_arch)
model.compile(loss="mean_squared_error", optimizer="rmsprop")
```

Sequential 只能设置简单的前馈神经网络结构。这里需要2个输入，所以用keras.models.Model 来把两个sequential结构组合成一个网络。

Skip-gram模型构架设置

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
embedding_1_input (InputLayer)	[(None, 1)]	0	
embedding_2_input (InputLayer)	[(None, 1)]	0	
embedding_1 (Embedding)	(None, 1, 100)	206600	embedding_1_input[0][0]
embedding_2 (Embedding)	(None, 1, 100)	206600	embedding_2_input[0][0]
reshape (Reshape)	(None, 100)	0	embedding_1[0][0]
reshape_1 (Reshape)	(None, 100)	0	embedding_2[0][0]
dot (Dot)	(None, 1)	0	reshape[0][0] reshape_1[0][0]
dense_1 (Dense)	(None, 1)	2	dot[0][0]

Total params: 413,202
Trainable params: 413,202
Non-trainable params: 0

Skip-gram模型训练

```
for epoch in range(1, 5):
    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)
```

```
Processed 0 (skip_first, skip_second, relevance) pairs
Epoch: 1 Loss: 177.72522800788283
Processed 0 (skip_first, skip_second, relevance) pairs
Epoch: 2 Loss: 137.79669494181870
Processed 0 (skip_first, skip_second, relevance) pairs
Epoch: 3 Loss: 128.55963351950050
Processed 0 (skip_first, skip_second, relevance) pairs
Epoch: 4 Loss: 123.41796318441630
```

Skip-gram得到词向量

我们有两个embedding 层，从 target word 对应的embedding 层，即代码中的word_model embedding layer输出得到词向量：

```
word_embed_layer = model.layers[2]
weights = word_embed_layer.get_weights()[0][1:]
print(weights.shape)
pd.DataFrame(weights, index=id2word.values()).head()
```

```
(2065, 100)
```

	0	1	2	3	4	5	6	7	8	9	...	90	91	92	93
and	-0.047693	0.025919	-0.023931	-0.033510	0.008227	0.044176	-0.027554	0.038666	0.025812	-0.045777	...	0.001122	0.010759	0.005660	-0.048318
the	-0.034364	-0.026349	-0.000768	-0.051639	-0.034292	-0.050003	0.014922	0.002428	-0.024234	-0.019147	...	-0.046160	-0.041331	-0.019112	0.027922
of	0.017875	-0.050923	-0.037167	-0.026110	-0.041660	0.015647	-0.003527	-0.018902	-0.015438	0.002075	...	0.038508	-0.015974	-0.029119	-0.012335
in	-0.034968	-0.007897	0.025955	-0.017882	-0.005144	-0.016282	-0.014672	0.016514	-0.038185	0.008748	...	-0.022279	0.039365	-0.020319	-0.000492
unto	0.011371	0.047877	-0.025185	0.049433	0.023195	-0.010020	-0.035604	-0.001297	0.038603	0.010094	...	0.017898	0.012751	0.037399	0.021824

5 rows × 100 columns

基于预训练句子向量

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

tensorflow_hub 有各种预训练好的模型以及词向量，句子向量。

载入预训练好的句子向量，这里是50维，然后传给一个128个节点的非线性隐藏层(Dense)，最后输出分类结果。

默认情况下，hub.KerasLayer不进行训练，你也可以设置trainable = True进行微调。

实际做的工作：用Google News 进行训练得到词嵌入，然后对词嵌入向量取平均得到句子向量（实际上再乘以句子包含词的个数的平方根）。