以下资料来自英文网站

<https://tensorflow.google.cn/tutorials/keras/>

是机器翻译的结果，请对照英文资料，学习并运行所有代码，修改文字使之通顺流畅。

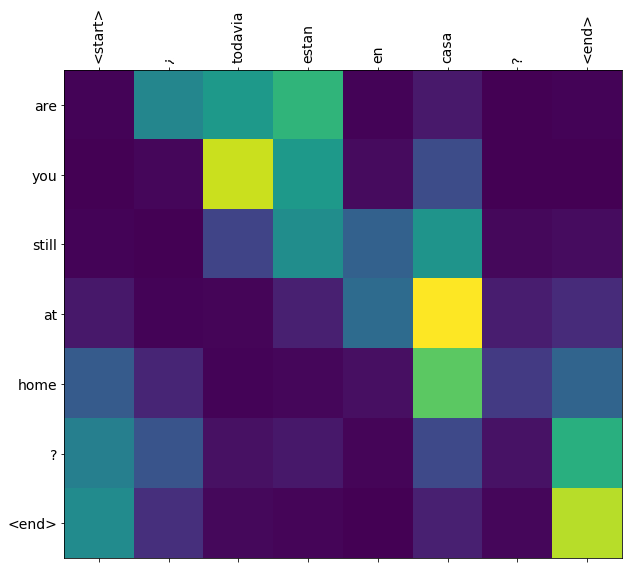
（使用Word的‘审阅’修订功能进行修改，保留所有修改痕迹）

# 使用注意的神经机器翻译

这个笔记本使用[tf.keras](https://www.tensorflow.org/programmers_guide/keras)和[eager执行](https://www.tensorflow.org/programmers_guide/eager)训练序列到序列（seq2seq）模型，用于西班牙语到英语的翻译。这是一个高级示例，它假定序列模型的序列知识。

在本笔记本中训练模型后，您将能够输入西班牙语句子，例如*“¿todavia estan en casa？”*，并返回英文翻译：*“你还在家吗？”*

对于玩具示例而言，翻译质量是合理的，但生成的关注图可能更有趣。这表明输入句子的哪些部分在翻译时具有模型的注意力：



注意：此示例在单个P100 GPU上运行大约需要10分钟。

In [0]:

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function

*# Import TensorFlow >= 1.10 and enable eager execution*

**import** **tensorflow** **as** **tf**

tf.enable\_eager\_execution()

**import** **matplotlib.pyplot** **as** **plt**

**from** **sklearn.model\_selection** **import** train\_test\_split

**import** **unicodedata**

**import** **re**

**import** **numpy** **as** **np**

**import** **os**

**import** **time**

print(tf.\_\_version\_\_)

## 下载并准备数据集

我们将使用<http://www.manythings.org/anki/>提供的语言数据集。此数据集包含以下格式的语言翻译对：

May I borrow this book? ¿Puedo tomar prestado este libro?

有多种语言可供选择，但我们将使用英语 - 西班牙语数据集。为方便起见，我们在Google Cloud上托管了此数据集的副本，但您也可以下载自己的副本。下载数据集后，以下是我们准备数据的步骤：

1. 为每个句子添加*开始*和*结束*标记。
2. 删除特殊字符来清除句子。
3. 创建一个单词索引和反向单词索引（从单词→id和id→单词映射的字典）。
4. 将每个句子填充到最大长度。

In [0]:

*# Download the file*

path\_to\_zip = tf.keras.utils.get\_file(

'spa-eng.zip', origin='http://download.tensorflow.org/data/spa-eng.zip',

extract=**True**)

path\_to\_file = os.path.dirname(path\_to\_zip)+"/spa-eng/spa.txt"

In [0]:

*# Converts the unicode file to ascii*

**def** unicode\_to\_ascii(s):

**return** ''.join(c **for** c **in** unicodedata.normalize('NFD', s)

**if** unicodedata.category(c) != 'Mn')

**def** preprocess\_sentence(w):

w = unicode\_to\_ascii(w.lower().strip())

*# creating a space between a word and the punctuation following it*

*# eg: "he is a boy." => "he is a boy ."*

*# Reference:- https://stackoverflow.com/questions/3645931/python-padding-punctuation-with-white-spaces-keeping-punctuation*

w = re.sub(r"([?.!,¿])", r" \1 ", w)

w = re.sub(r'[" "]+', " ", w)

*# replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")*

w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)

w = w.rstrip().strip()

*# adding a start and an end token to the sentence*

*# so that the model know when to start and stop predicting.*

w = '<start> ' + w + ' <end>'

**return** w

In [0]:

*# 1. Remove the accents*

*# 2. Clean the sentences*

*# 3. Return word pairs in the format: [ENGLISH, SPANISH]*

**def** create\_dataset(path, num\_examples):

lines = open(path, encoding='UTF-8').read().strip().split('**\n**')

word\_pairs = [[preprocess\_sentence(w) **for** w **in** l.split('**\t**')] **for** l **in** lines[:num\_examples]]

**return** word\_pairs

In [0]:

*# This class creates a word -> index mapping (e.g,. "dad" -> 5) and vice-versa*

*# (e.g., 5 -> "dad") for each language,*

**class** **LanguageIndex**():

**def** \_\_init\_\_(self, lang):

self.lang = lang

self.word2idx = {}

self.idx2word = {}

self.vocab = set()

self.create\_index()

**def** create\_index(self):

**for** phrase **in** self.lang:

self.vocab.update(phrase.split(' '))

self.vocab = sorted(self.vocab)

self.word2idx['<pad>'] = 0

**for** index, word **in** enumerate(self.vocab):

self.word2idx[word] = index + 1

**for** word, index **in** self.word2idx.items():

self.idx2word[index] = word

In [0]:

**def** max\_length(tensor):

**return** max(len(t) **for** t **in** tensor)

**def** load\_dataset(path, num\_examples):

*# creating cleaned input, output pairs*

pairs = create\_dataset(path, num\_examples)

*# index language using the class defined above*

inp\_lang = LanguageIndex(sp **for** en, sp **in** pairs)

targ\_lang = LanguageIndex(en **for** en, sp **in** pairs)

*# Vectorize the input and target languages*

*# Spanish sentences*

input\_tensor = [[inp\_lang.word2idx[s] **for** s **in** sp.split(' ')] **for** en, sp **in** pairs]

*# English sentences*

target\_tensor = [[targ\_lang.word2idx[s] **for** s **in** en.split(' ')] **for** en, sp **in** pairs]

*# Calculate max\_length of input and output tensor*

*# Here, we'll set those to the longest sentence in the dataset*

max\_length\_inp, max\_length\_tar = max\_length(input\_tensor), max\_length(target\_tensor)

*# Padding the input and output tensor to the maximum length*

input\_tensor = tf.keras.preprocessing.sequence.pad\_sequences(input\_tensor,

maxlen=max\_length\_inp,

padding='post')

target\_tensor = tf.keras.preprocessing.sequence.pad\_sequences(target\_tensor,

maxlen=max\_length\_tar,

padding='post')

**return** input\_tensor, target\_tensor, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_tar

### 限制数据集的大小以更快地进行实验（可选）

对> 100,000个句子的完整数据集进行培训需要很长时间。为了更快地训练，我们可以将数据集的大小限制为30,000个句子（当然，翻译质量会随着数据的减少而降低）：

In [0]:

*# Try experimenting with the size of that dataset*

num\_examples = 30000

input\_tensor, target\_tensor, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ = load\_dataset(path\_to\_file, num\_examples)

In [0]:

*# Creating training and validation sets using an 80-20 split*

input\_tensor\_train, input\_tensor\_val, target\_tensor\_train, target\_tensor\_val = train\_test\_split(input\_tensor, target\_tensor, test\_size=0.2)

*# Show length*

len(input\_tensor\_train), len(target\_tensor\_train), len(input\_tensor\_val), len(target\_tensor\_val)

### 创建tf.data数据集

In [0]:

BUFFER\_SIZE = len(input\_tensor\_train)

BATCH\_SIZE = 64

N\_BATCH = BUFFER\_SIZE//BATCH\_SIZE

embedding\_dim = 256

units = 1024

vocab\_inp\_size = len(inp\_lang.word2idx)

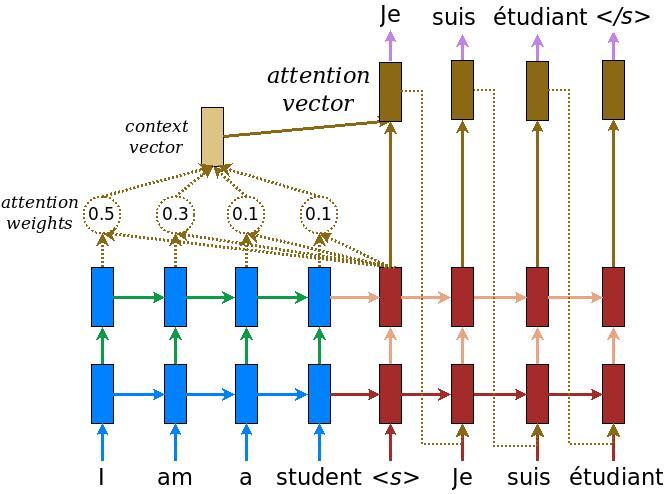
vocab\_tar\_size = len(targ\_lang.word2idx)

dataset = tf.data.Dataset.from\_tensor\_slices((input\_tensor\_train, target\_tensor\_train)).shuffle(BUFFER\_SIZE)

dataset = dataset.batch(BATCH\_SIZE, drop\_remainder=**True**)

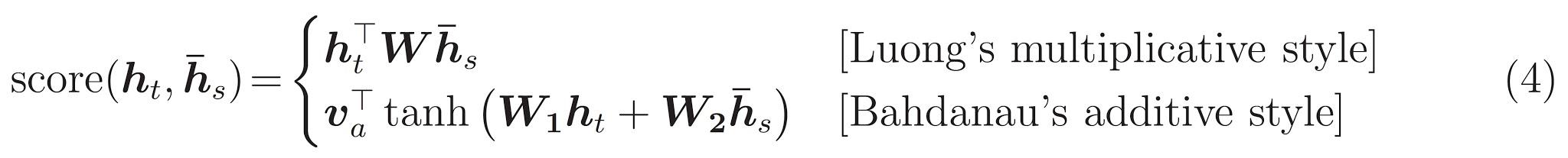
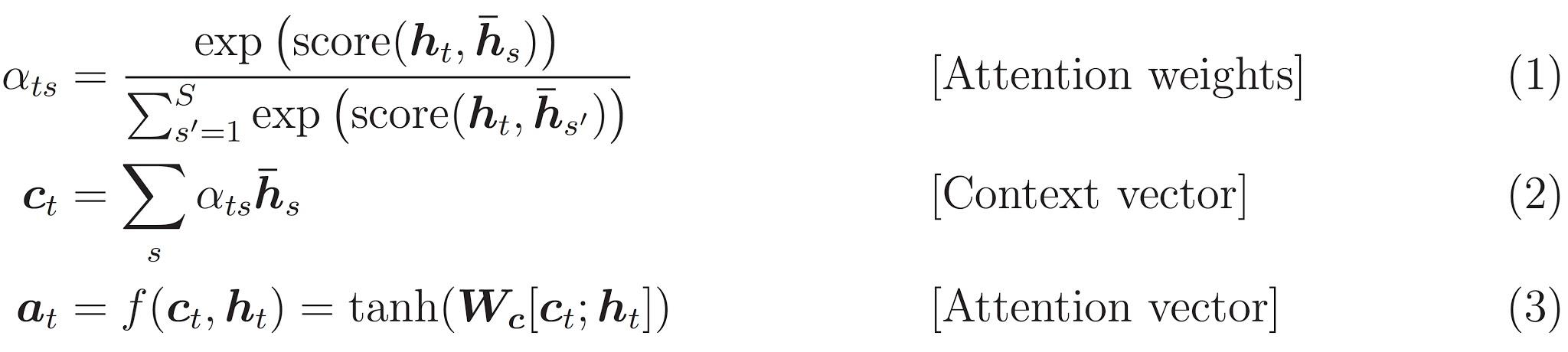
## 编写编码器和解码器模型

在这里，我们将实现一个关注的编码器 - 解码器模型，您可以在TensorFlow [神经机器翻译（seq2seq）教程中阅读](https://www.tensorflow.org/tutorials/seq2seq)。此示例使用更新的API集。这款笔记本实现了seq2seq教程中的[注意力方程式](https://www.tensorflow.org/tutorials/seq2seq#background_on_the_attention_mechanism)。下图显示了每个输入词由注意机制分配权重，然后解码器使用该权重来预测句子中的下一个词。



输入通过编码器模型，该模型为我们提供形状的编码器输出*（batch\_size，max\_length，hidden\_​​size）*和编码器隐藏的形状状态*（batch\_size，hidden\_​​size）*。

以下是实施的方程式：



我们正在使用*Bahdanau的注意力*。让我们在编写简化表格之前决定表示法：

* FC =完全连接（密集）层
* EO =编码器输出
* H =隐藏状态
* X =解码器的输入

和伪代码：

* score = FC(tanh(FC(EO) + FC(H)))
* attention weights = softmax(score, axis = 1)。默认情况下，Softmax应用于最后一个轴，但是我们要在第*一个轴上*应用它，因为得分的形状是*（batch\_size，max\_length，hidden\_​​size）*。Max\_length是我们输入的长度。由于我们尝试为每个输入分配权重，因此应在该轴上应用softmax。
* context vector = sum(attention weights \* EO, axis = 1)。选择轴为1的原因与上述相同。
* embedding output =解码器X的输入通过嵌入层。
* merged vector = concat(embedding output, context vector)
* 然后将该合并的矢量提供给GRU

每个步骤中所有向量的形状都已在代码中的注释中指定：

In [0]:

**def** gru(units):

*# If you have a GPU, we recommend using CuDNNGRU(provides a 3x speedup than GRU)*

*# the code automatically does that.*

**if** tf.test.is\_gpu\_available():

**return** tf.keras.layers.CuDNNGRU(units,

return\_sequences=**True**,

return\_state=**True**,

recurrent\_initializer='glorot\_uniform')

**else**:

**return** tf.keras.layers.GRU(units,

return\_sequences=**True**,

return\_state=**True**,

recurrent\_activation='sigmoid',

recurrent\_initializer='glorot\_uniform')

In [0]:

**class** **Encoder**(tf.keras.Model):

**def** \_\_init\_\_(self, vocab\_size, embedding\_dim, enc\_units, batch\_sz):

super(Encoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.enc\_units = enc\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = gru(self.enc\_units)

**def** call(self, x, hidden):

x = self.embedding(x)

output, state = self.gru(x, initial\_state = hidden)

**return** output, state

**def** initialize\_hidden\_state(self):

**return** tf.zeros((self.batch\_sz, self.enc\_units))

In [0]:

**class** **Decoder**(tf.keras.Model):

**def** \_\_init\_\_(self, vocab\_size, embedding\_dim, dec\_units, batch\_sz):

super(Decoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.dec\_units = dec\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = gru(self.dec\_units)

self.fc = tf.keras.layers.Dense(vocab\_size)

*# used for attention*

self.W1 = tf.keras.layers.Dense(self.dec\_units)

self.W2 = tf.keras.layers.Dense(self.dec\_units)

self.V = tf.keras.layers.Dense(1)

**def** call(self, x, hidden, enc\_output):

*# enc\_output shape == (batch\_size, max\_length, hidden\_size)*

*# hidden shape == (batch\_size, hidden size)*

*# hidden\_with\_time\_axis shape == (batch\_size, 1, hidden size)*

*# we are doing this to perform addition to calculate the score*

hidden\_with\_time\_axis = tf.expand\_dims(hidden, 1)

*# score shape == (batch\_size, max\_length, hidden\_size)*

score = tf.nn.tanh(self.W1(enc\_output) + self.W2(hidden\_with\_time\_axis))

*# attention\_weights shape == (batch\_size, max\_length, 1)*

*# we get 1 at the last axis because we are applying score to self.V*

attention\_weights = tf.nn.softmax(self.V(score), axis=1)

*# context\_vector shape after sum == (batch\_size, hidden\_size)*

context\_vector = attention\_weights \* enc\_output

context\_vector = tf.reduce\_sum(context\_vector, axis=1)

*# x shape after passing through embedding == (batch\_size, 1, embedding\_dim)*

x = self.embedding(x)

*# x shape after concatenation == (batch\_size, 1, embedding\_dim + hidden\_size)*

x = tf.concat([tf.expand\_dims(context\_vector, 1), x], axis=-1)

*# passing the concatenated vector to the GRU*

output, state = self.gru(x)

*# output shape == (batch\_size \* 1, hidden\_size)*

output = tf.reshape(output, (-1, output.shape[2]))

*# output shape == (batch\_size \* 1, vocab)*

x = self.fc(output)

**return** x, state, attention\_weights

**def** initialize\_hidden\_state(self):

**return** tf.zeros((self.batch\_sz, self.dec\_units))

In [0]:

encoder = Encoder(vocab\_inp\_size, embedding\_dim, units, BATCH\_SIZE)

decoder = Decoder(vocab\_tar\_size, embedding\_dim, units, BATCH\_SIZE)

## 定义优化器和损失函数

In [0]:

optimizer = tf.train.AdamOptimizer()

**def** loss\_function(real, pred):

mask = 1 - np.equal(real, 0)

loss\_ = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=real, logits=pred) \* mask

**return** tf.reduce\_mean(loss\_)

## 检查点（基于对象的保存）

In [0]:

checkpoint\_dir = './training\_checkpoints'

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

checkpoint = tf.train.Checkpoint(optimizer=optimizer,

encoder=encoder,

decoder=decoder)

**Training**

1. 通过*编码器*传递*输入*，*编码器*返回*编码器输出*和*编码器隐藏状态*。
2. 编码器输出，编码器隐藏状态和解码器输入（它是*开始标记*）被传递给解码器。
3. 解码器返回*预测*和*解码器隐藏状态*。
4. 然后将解码器隐藏状态传递回模型，并使用预测来计算损失。
5. 使用*教师强制*决定解码器的下一个输入。
6. *教师强制*是将*目标字*作为*下一个输入*传递给解码器的技术。
7. 最后一步是计算渐变并将其应用于优化器并反向传播。

In [0]:

EPOCHS = 10

**for** epoch **in** range(EPOCHS):

start = time.time()

hidden = encoder.initialize\_hidden\_state()

total\_loss = 0

**for** (batch, (inp, targ)) **in** enumerate(dataset):

loss = 0

**with** tf.GradientTape() **as** tape:

enc\_output, enc\_hidden = encoder(inp, hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([targ\_lang.word2idx['<start>']] \* BATCH\_SIZE, 1)

*# Teacher forcing - feeding the target as the next input*

**for** t **in** range(1, targ.shape[1]):

*# passing enc\_output to the decoder*

predictions, dec\_hidden, \_ = decoder(dec\_input, dec\_hidden, enc\_output)

loss += loss\_function(targ[:, t], predictions)

*# using teacher forcing*

dec\_input = tf.expand\_dims(targ[:, t], 1)

batch\_loss = (loss / int(targ.shape[1]))

total\_loss += batch\_loss

variables = encoder.variables + decoder.variables

gradients = tape.gradient(loss, variables)

optimizer.apply\_gradients(zip(gradients, variables))

**if** batch % 100 == 0:

print('Epoch **{}** Batch **{}** Loss **{:.4f}**'.format(epoch + 1,

batch,

batch\_loss.numpy()))

*# saving (checkpoint) the model every 2 epochs*

**if** (epoch + 1) % 2 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)

print('Epoch **{}** Loss **{:.4f}**'.format(epoch + 1,

total\_loss / N\_BATCH))

print('Time taken for 1 epoch **{}** sec**\n**'.format(time.time() - start))

**Translate**

* evaluate函数类似于训练循环，除了我们在这里不使用*教师强制*。在每个时间步骤对解码器的输入是其先前的预测以及隐藏状态和编码器输出。
* 停止预测模型何时预测*结束标记*。
* 并存储*每个时间步*的*注意力*。

注意：编码器输出仅针对一个输入计算一次。

In [0]:

**def** evaluate(sentence, encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ):

attention\_plot = np.zeros((max\_length\_targ, max\_length\_inp))

sentence = preprocess\_sentence(sentence)

inputs = [inp\_lang.word2idx[i] **for** i **in** sentence.split(' ')]

inputs = tf.keras.preprocessing.sequence.pad\_sequences([inputs], maxlen=max\_length\_inp, padding='post')

inputs = tf.convert\_to\_tensor(inputs)

result = ''

hidden = [tf.zeros((1, units))]

enc\_out, enc\_hidden = encoder(inputs, hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([targ\_lang.word2idx['<start>']], 0)

**for** t **in** range(max\_length\_targ):

predictions, dec\_hidden, attention\_weights = decoder(dec\_input, dec\_hidden, enc\_out)

*# storing the attention weigths to plot later on*

attention\_weights = tf.reshape(attention\_weights, (-1, ))

attention\_plot[t] = attention\_weights.numpy()

predicted\_id = tf.multinomial(predictions, num\_samples=1)[0][0].numpy()

result += targ\_lang.idx2word[predicted\_id] + ' '

**if** targ\_lang.idx2word[predicted\_id] == '<end>':

**return** result, sentence, attention\_plot

*# the predicted ID is fed back into the model*

dec\_input = tf.expand\_dims([predicted\_id], 0)

**return** result, sentence, attention\_plot

In [0]:

*# function for plotting the attention weights*

**def** plot\_attention(attention, sentence, predicted\_sentence):

fig = plt.figure(figsize=(10,10))

ax = fig.add\_subplot(1, 1, 1)

ax.matshow(attention, cmap='viridis')

fontdict = {'fontsize': 14}

ax.set\_xticklabels([''] + sentence, fontdict=fontdict, rotation=90)

ax.set\_yticklabels([''] + predicted\_sentence, fontdict=fontdict)

plt.show()

In [0]:

**def** translate(sentence, encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ):

result, sentence, attention\_plot = evaluate(sentence, encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ)

print('Input: **{}**'.format(sentence))

print('Predicted translation: **{}**'.format(result))

attention\_plot = attention\_plot[:len(result.split(' ')), :len(sentence.split(' '))]

plot\_attention(attention\_plot, sentence.split(' '), result.split(' '))

## 恢复最新的检查点和测试

In [0]:

*# restoring the latest checkpoint in checkpoint\_dir*

checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

In [0]:

translate('hace mucho frio aqui.', encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ)

In [0]:

translate('esta es mi vida.', encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ)

In [0]:

translate('¿todavia estan en casa?', encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ)

In [0]:

*# wrong translation*

translate('trata de averiguarlo.', encoder, decoder, inp\_lang, targ\_lang, max\_length\_inp, max\_length\_targ)

**后续步骤**

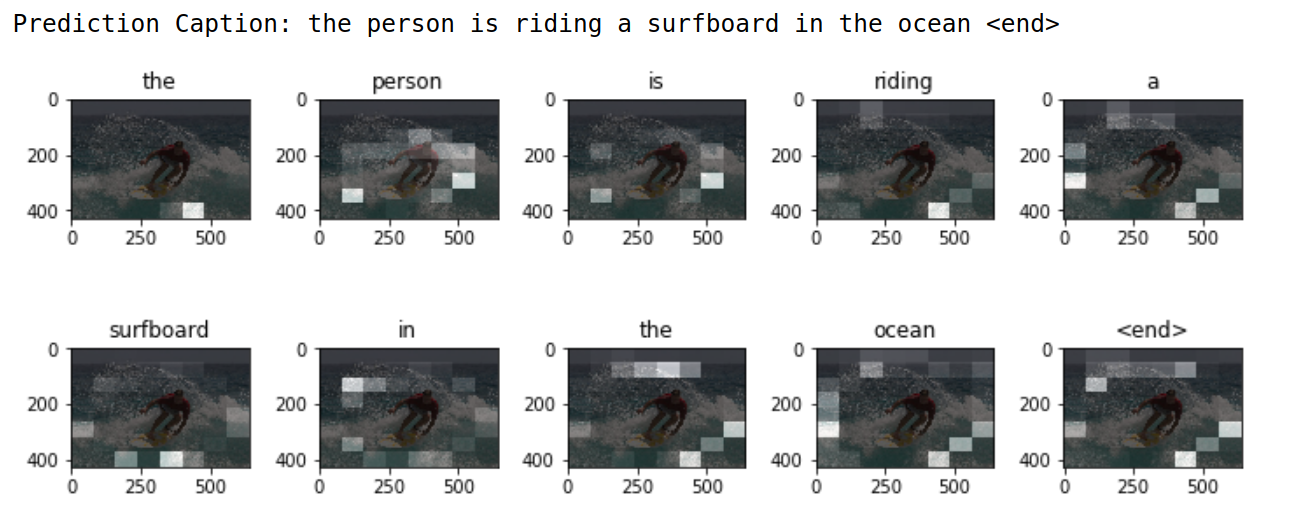
* [下载不同的数据集](http://www.manythings.org/anki/)以试验翻译，例如，英语到德语，或英语到法语。
* 尝试对更大的数据集进行训练，或使用更多的时期

# 使用注意的图像字幕

图像字幕是为图像生成标题的任务。给出这样的图像：



我们的目标是产生一个标题，例如“冲浪者骑在波浪上”。在这里，我们将使用基于注意力的模型。这使我们能够在生成标题时查看模型关注的图像部分。



下面的这个模型体系结构类似于[Show，Attend和Tell：使用Visual Attention生成神经图像标题](https://arxiv.org/abs/1502.03044)。

该代码使用[tf.keras](https://www.tensorflow.org/programmers_guide/keras)和[eager执行](https://www.tensorflow.org/programmers_guide/eager)，您可以在链接指南中了解更多信息。

这款笔记本是一个端到端的例子。如果你运行它，它将下载 [MS-COCO](http://cocodataset.org/#home)数据集，使用Inception V3预处理和缓存图像的子集，训练编码器 - 解码器模型，并使用它来生成新图像的标题。

该代码需要TensorFlow版本> = 1.9。如果你在[Colab中](https://render.githubusercontent.com/view/ipynb?commit=c19e29306ce1777456b2dbb3a14f511edf7883a8&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f74656e736f72666c6f772f74656e736f72666c6f772f633139653239333036636531373737343536623264626233613134663531316564663738383361382f74656e736f72666c6f772f636f6e747269622f65616765722f707974686f6e2f6578616d706c65732f67656e657261746976655f6578616d706c65732f696d6167655f63617074696f6e696e675f776974685f617474656e74696f6e2e6970796e62&nwo=tensorflow%2Ftensorflow&path=tensorflow%2Fcontrib%2Feager%2Fpython%2Fexamples%2Fgenerative_examples%2Fimage_captioning_with_attention.ipynb&repository_id=45717250&repository_type=Repository)运行它

在这个例子中，我们正在训练相对少量的数据作为例子。在单个P100 GPU上，此示例大约需要2个小时才能进行训练。我们训练前30,000个字幕（对应于约20,000个图像，取决于改组，因为数据集中每个图像有多个字幕）

In [0]:

*# Import TensorFlow and enable eager execution*

*# This code requires TensorFlow version >=1.9*

**import** **tensorflow** **as** **tf**

tf.enable\_eager\_execution()

*# We'll generate plots of attention in order to see which parts of an image*

*# our model focuses on during captioning*

**import** **matplotlib.pyplot** **as** **plt**

*# Scikit-learn includes many helpful utilities*

**from** **sklearn.model\_selection** **import** train\_test\_split

**from** **sklearn.utils** **import** shuffle

**import** **re**

**import** **numpy** **as** **np**

**import** **os**

**import** **time**

**import** **json**

**from** **glob** **import** glob

**from** **PIL** **import** Image

**import** **pickle**

## 下载并准备MS-COCO数据集

我们将使用[MS-COCO数据集](http://cocodataset.org/#home)来训练我们的模型。此数据集包含> 82,000个图像，每个图像都注释了至少5个不同的标题。下面的代码将自动下载并提取数据集。

**注意：提前大量下载**。我们将使用训练集，它是一个13GB的文件。

In [0]:

annotation\_zip = tf.keras.utils.get\_file('captions.zip',

cache\_subdir=os.path.abspath('.'),

origin = 'http://images.cocodataset.org/annotations/annotations\_trainval2014.zip',

extract = **True**)

annotation\_file = os.path.dirname(annotation\_zip)+'/annotations/captions\_train2014.json'

name\_of\_zip = 'train2014.zip'

**if** **not** os.path.exists(os.path.abspath('.') + '/' + name\_of\_zip):

image\_zip = tf.keras.utils.get\_file(name\_of\_zip,

cache\_subdir=os.path.abspath('.'),

origin = 'http://images.cocodataset.org/zips/train2014.zip',

extract = **True**)

PATH = os.path.dirname(image\_zip)+'/train2014/'

**else**:

PATH = os.path.abspath('.')+'/train2014/'

## （可选）限制训练集的大小以加快训练

对于此示例，我们将选择30,000个字幕的子集，并使用这些和相应的图像来训练我们的模型。与往常一样，如果您选择使用更多数据，字幕质量将会提高。

In [0]:

*# read the json file*

**with** open(annotation\_file, 'r') **as** f:

annotations = json.load(f)

*# storing the captions and the image name in vectors*

all\_captions = []

all\_img\_name\_vector = []

**for** annot **in** annotations['annotations']:

caption = '<start> ' + annot['caption'] + ' <end>'

image\_id = annot['image\_id']

full\_coco\_image\_path = PATH + 'COCO\_train2014\_' + '**%012d**.jpg' % (image\_id)

all\_img\_name\_vector.append(full\_coco\_image\_path)

all\_captions.append(caption)

*# shuffling the captions and image\_names together*

*# setting a random state*

train\_captions, img\_name\_vector = shuffle(all\_captions,

all\_img\_name\_vector,

random\_state=1)

*# selecting the first 30000 captions from the shuffled set*

num\_examples = 30000

train\_captions = train\_captions[:num\_examples]

img\_name\_vector = img\_name\_vector[:num\_examples]

In [0]:

len(train\_captions), len(all\_captions)

**使用InceptionV3预处理图像**

接下来，我们将使用InceptionV3（在Imagenet上预训练）对每个图像进行分类。我们将从最后一个卷积层中提取特征。

首先，我们需要将图像转换为格式inceptionV3期望：

* 调整图像大小为（299,299）
* 使用[preprocess\_input](https://www.tensorflow.org/api_docs/python/tf/keras/applications/inception_v3/preprocess_input)方法将像素放置在-1到1的范围内（以匹配用于训练InceptionV3的图像的格式）。

In [0]:

**def** load\_image(image\_path):

img = tf.read\_file(image\_path)

img = tf.image.decode\_jpeg(img, channels=3)

img = tf.image.resize\_images(img, (299, 299))

img = tf.keras.applications.inception\_v3.preprocess\_input(img)

**return** img, image\_path

**初始化InceptionV3并加载预训练的Imagenet权重**

为此，我们将创建一个tf.keras模型，其中输出层是InceptionV3体系结构中的最后一个卷积层。

* 每个图像都通过网络转发，我们最后得到的矢量存储在字典中（image\_name - > feature\_vector）。
* 我们使用最后一个卷积层，因为我们在这个例子中使用了注意力。这层的输出形状是8x8x2048。
* 我们在培训期间避免这样做，因此它不会成为瓶颈。
* 在所有图像通过网络传递之后，我们挑选字典并将其保存到磁盘。

In [0]:

image\_model = tf.keras.applications.InceptionV3(include\_top=**False**,

weights='imagenet')

new\_input = image\_model.input

hidden\_layer = image\_model.layers[-1].output

image\_features\_extract\_model = tf.keras.Model(new\_input, hidden\_layer)

## 缓存从InceptionV3中提取的功能

我们将使用InceptionV3预处理每个图像并将输出缓存到磁盘。缓存RAM中的输出会更快但内存密集，每个映像需要8 \* 8 \* 2048个浮点数。在撰写本文时，这将超出Colab的内存限制（尽管这些可能会发生变化，但实例似乎目前有大约12GB的内存）。

通过更复杂的缓存策略（例如，通过分割图像以减少随机访问磁盘I / O）可以以更多代码为代价来提高性能。

使用GPU在Colab中运行大约需要10分钟。如果您想查看进度条，可以：安装[tqdm](https://github.com/tqdm/tqdm)（!pip install tqdm），然后更改此行：

for img, path in image\_dataset:

至：

for img, path in tqdm(image\_dataset):.

In [0]:

*# getting the unique images*

encode\_train = sorted(set(img\_name\_vector))

*# feel free to change the batch\_size according to your system configuration*

image\_dataset = tf.data.Dataset.from\_tensor\_slices(

encode\_train).map(load\_image).batch(16)

**for** img, path **in** image\_dataset:

batch\_features = image\_features\_extract\_model(img)

batch\_features = tf.reshape(batch\_features,

(batch\_features.shape[0], -1, batch\_features.shape[3]))

**for** bf, p **in** zip(batch\_features, path):

path\_of\_feature = p.numpy().decode("utf-8")

np.save(path\_of\_feature, bf.numpy())

**预处理并标记字幕**

* 首先，我们将标记字幕（例如，通过拆分空格）。这将为我们提供数据中所有独特单词的词汇表（例如，“冲浪”，“足球”等）。
* 接下来，我们将词汇量限制为前5,000个单词以节省内存。我们将用令牌“UNK”替换所有其他单词（对于未知）。
* 最后，我们创建一个单词 - >索引映射，反之亦然。
* 然后我们将所有序列填充到与最长序列相同的长度。

In [0]:

*# This will find the maximum length of any caption in our dataset*

**def** calc\_max\_length(tensor):

**return** max(len(t) **for** t **in** tensor)

In [0]:

*# The steps above is a general process of dealing with text processing*

*# choosing the top 5000 words from the vocabulary*

top\_k = 5000

tokenizer = tf.keras.preprocessing.text.Tokenizer(num\_words=top\_k,

oov\_token="<unk>",

filters='!"#$%&()\*+.,-/:;=?@[\]^\_`{|}~ ')

tokenizer.fit\_on\_texts(train\_captions)

train\_seqs = tokenizer.texts\_to\_sequences(train\_captions)

In [0]:

tokenizer.word\_index = {key:value **for** key, value **in** tokenizer.word\_index.items() **if** value <= top\_k}

*# putting <unk> token in the word2idx dictionary*

tokenizer.word\_index[tokenizer.oov\_token] = top\_k + 1

tokenizer.word\_index['<pad>'] = 0

In [0]:

*# creating the tokenized vectors*

train\_seqs = tokenizer.texts\_to\_sequences(train\_captions)

In [0]:

*# creating a reverse mapping (index -> word)*

index\_word = {value:key **for** key, value **in** tokenizer.word\_index.items()}

In [0]:

*# padding each vector to the max\_length of the captions*

*# if the max\_length parameter is not provided, pad\_sequences calculates that automatically*

cap\_vector = tf.keras.preprocessing.sequence.pad\_sequences(train\_seqs, padding='post')

In [0]:

*# calculating the max\_length*

*# used to store the attention weights*

max\_length = calc\_max\_length(train\_seqs)

## 将数据拆分为培训和测试

In [0]:

*# Create training and validation sets using 80-20 split*

img\_name\_train, img\_name\_val, cap\_train, cap\_val = train\_test\_split(img\_name\_vector,

cap\_vector,

test\_size=0.2,

random\_state=0)

In [0]:

len(img\_name\_train), len(cap\_train), len(img\_name\_val), len(cap\_val)

## 我们的图片和标题已准备就绪！接下来，让我们创建一个tf.data数据集来用于训练我们的模型。

In [0]:

*# feel free to change these parameters according to your system's configuration*

BATCH\_SIZE = 64

BUFFER\_SIZE = 1000

embedding\_dim = 256

units = 512

vocab\_size = len(tokenizer.word\_index)

*# shape of the vector extracted from InceptionV3 is (64, 2048)*

*# these two variables represent that*

features\_shape = 2048

attention\_features\_shape = 64

In [0]:

*# loading the numpy files*

**def** map\_func(img\_name, cap):

img\_tensor = np.load(img\_name.decode('utf-8')+'.npy')

**return** img\_tensor, cap

In [0]:

dataset = tf.data.Dataset.from\_tensor\_slices((img\_name\_train, cap\_train))

*# using map to load the numpy files in parallel*

*# NOTE: Be sure to set num\_parallel\_calls to the number of CPU cores you have*

*# https://www.tensorflow.org/api\_docs/python/tf/py\_func*

dataset = dataset.map(**lambda** item1, item2: tf.py\_func(

map\_func, [item1, item2], [tf.float32, tf.int32]), num\_parallel\_calls=8)

*# shuffling and batching*

dataset = dataset.shuffle(BUFFER\_SIZE)

*# https://www.tensorflow.org/api\_docs/python/tf/contrib/data/batch\_and\_drop\_remainder*

dataset = dataset.batch(BATCH\_SIZE)

dataset = dataset.prefetch(1)

**Model**

有趣的是，下面的解码器[与具有注意力的神经机器翻译](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb)的示例中的解码器相同。

模型架构的灵感来自[Show，Attend和Tell](https://arxiv.org/pdf/1502.03044.pdf)纸张。

* 在这个例子中，我们从InceptionV3的下卷积层中提取特征，给出了一个形状向量（8,8,2048）。
* 我们将其压成（64,2048）的形状。
* 然后该矢量通过CNN编码器（由单个完全连接的层组成）。
* RNN（此处为GRU）参与图像以预测下一个单词。

In [0]:

**def** gru(units):

*# If you have a GPU, we recommend using the CuDNNGRU layer (it provides a*

*# significant speedup).*

**if** tf.test.is\_gpu\_available():

**return** tf.keras.layers.CuDNNGRU(units,

return\_sequences=**True**,

return\_state=**True**,

recurrent\_initializer='glorot\_uniform')

**else**:

**return** tf.keras.layers.GRU(units,

return\_sequences=**True**,

return\_state=**True**,

recurrent\_activation='sigmoid',

recurrent\_initializer='glorot\_uniform')

In [0]:

**class** **BahdanauAttention**(tf.keras.Model):

**def** \_\_init\_\_(self, units):

super(BahdanauAttention, self).\_\_init\_\_()

self.W1 = tf.keras.layers.Dense(units)

self.W2 = tf.keras.layers.Dense(units)

self.V = tf.keras.layers.Dense(1)

**def** call(self, features, hidden):

*# features(CNN\_encoder output) shape == (batch\_size, 64, embedding\_dim)*

*# hidden shape == (batch\_size, hidden\_size)*

*# hidden\_with\_time\_axis shape == (batch\_size, 1, hidden\_size)*

hidden\_with\_time\_axis = tf.expand\_dims(hidden, 1)

*# score shape == (batch\_size, 64, hidden\_size)*

score = tf.nn.tanh(self.W1(features) + self.W2(hidden\_with\_time\_axis))

*# attention\_weights shape == (batch\_size, 64, 1)*

*# we get 1 at the last axis because we are applying score to self.V*

attention\_weights = tf.nn.softmax(self.V(score), axis=1)

*# context\_vector shape after sum == (batch\_size, hidden\_size)*

context\_vector = attention\_weights \* features

context\_vector = tf.reduce\_sum(context\_vector, axis=1)

**return** context\_vector, attention\_weights

In [0]:

**class** **CNN\_Encoder**(tf.keras.Model):

*# Since we have already extracted the features and dumped it using pickle*

*# This encoder passes those features through a Fully connected layer*

**def** \_\_init\_\_(self, embedding\_dim):

super(CNN\_Encoder, self).\_\_init\_\_()

*# shape after fc == (batch\_size, 64, embedding\_dim)*

self.fc = tf.keras.layers.Dense(embedding\_dim)

**def** call(self, x):

x = self.fc(x)

x = tf.nn.relu(x)

**return** x

In [0]:

**class** **RNN\_Decoder**(tf.keras.Model):

**def** \_\_init\_\_(self, embedding\_dim, units, vocab\_size):

super(RNN\_Decoder, self).\_\_init\_\_()

self.units = units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = gru(self.units)

self.fc1 = tf.keras.layers.Dense(self.units)

self.fc2 = tf.keras.layers.Dense(vocab\_size)

self.attention = BahdanauAttention(self.units)

**def** call(self, x, features, hidden):

*# defining attention as a separate model*

context\_vector, attention\_weights = self.attention(features, hidden)

*# x shape after passing through embedding == (batch\_size, 1, embedding\_dim)*

x = self.embedding(x)

*# x shape after concatenation == (batch\_size, 1, embedding\_dim + hidden\_size)*

x = tf.concat([tf.expand\_dims(context\_vector, 1), x], axis=-1)

*# passing the concatenated vector to the GRU*

output, state = self.gru(x)

*# shape == (batch\_size, max\_length, hidden\_size)*

x = self.fc1(output)

*# x shape == (batch\_size \* max\_length, hidden\_size)*

x = tf.reshape(x, (-1, x.shape[2]))

*# output shape == (batch\_size \* max\_length, vocab)*

x = self.fc2(x)

**return** x, state, attention\_weights

**def** reset\_state(self, batch\_size):

**return** tf.zeros((batch\_size, self.units))

In [0]:

encoder = CNN\_Encoder(embedding\_dim)

decoder = RNN\_Decoder(embedding\_dim, units, vocab\_size)

In [0]:

optimizer = tf.train.AdamOptimizer()

*# We are masking the loss calculated for padding*

**def** loss\_function(real, pred):

mask = 1 - np.equal(real, 0)

loss\_ = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=real, logits=pred) \* mask

**return** tf.reduce\_mean(loss\_)

**Training**

* 我们提取存储在相应.npy文件中的功能，然后通过编码器传递这些功能。
* 编码器输出，隐藏状态（初始化为0）和解码器输入（开始标记）被传递给解码器。
* 解码器返回预测和解码器隐藏状态。
* 然后将解码器隐藏状态传递回模型，并使用预测来计算损失。
* 使用教师强制决定解码器的下一个输入。
* 教师强制是将目标字作为下一个输入传递给解码器的技术。
* 最后一步是计算渐变并将其应用于优化器并反向传播。

In [0]:

*# adding this in a separate cell because if you run the training cell*

*# many times, the loss\_plot array will be reset*

loss\_plot = []

In [0]:

EPOCHS = 20

**for** epoch **in** range(EPOCHS):

start = time.time()

total\_loss = 0

**for** (batch, (img\_tensor, target)) **in** enumerate(dataset):

loss = 0

*# initializing the hidden state for each batch*

*# because the captions are not related from image to image*

hidden = decoder.reset\_state(batch\_size=target.shape[0])

dec\_input = tf.expand\_dims([tokenizer.word\_index['<start>']] \* BATCH\_SIZE, 1)

**with** tf.GradientTape() **as** tape:

features = encoder(img\_tensor)

**for** i **in** range(1, target.shape[1]):

*# passing the features through the decoder*

predictions, hidden, \_ = decoder(dec\_input, features, hidden)

loss += loss\_function(target[:, i], predictions)

*# using teacher forcing*

dec\_input = tf.expand\_dims(target[:, i], 1)

total\_loss += (loss / int(target.shape[1]))

variables = encoder.variables + decoder.variables

gradients = tape.gradient(loss, variables)

optimizer.apply\_gradients(zip(gradients, variables), tf.train.get\_or\_create\_global\_step())

**if** batch % 100 == 0:

print ('Epoch **{}** Batch **{}** Loss **{:.4f}**'.format(epoch + 1,

batch,

loss.numpy() / int(target.shape[1])))

*# storing the epoch end loss value to plot later*

loss\_plot.append(total\_loss / len(cap\_vector))

print ('Epoch **{}** Loss **{:.6f}**'.format(epoch + 1,

total\_loss/len(cap\_vector)))

print ('Time taken for 1 epoch **{}** sec**\n**'.format(time.time() - start))

In [0]:

plt.plot(loss\_plot)

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Loss Plot')

plt.show()

**字幕！**

* evaluate函数类似于训练循环，除了我们在这里不使用教师强制。在每个时间步骤对解码器的输入是其先前的预测以及隐藏状态和编码器输出。
* 停止预测模型何时预测结束标记。
* 并存储每个时间步的注意力。

In [0]:

**def** evaluate(image):

attention\_plot = np.zeros((max\_length, attention\_features\_shape))

hidden = decoder.reset\_state(batch\_size=1)

temp\_input = tf.expand\_dims(load\_image(image)[0], 0)

img\_tensor\_val = image\_features\_extract\_model(temp\_input)

img\_tensor\_val = tf.reshape(img\_tensor\_val, (img\_tensor\_val.shape[0], -1, img\_tensor\_val.shape[3]))

features = encoder(img\_tensor\_val)

dec\_input = tf.expand\_dims([tokenizer.word\_index['<start>']], 0)

result = []

**for** i **in** range(max\_length):

predictions, hidden, attention\_weights = decoder(dec\_input, features, hidden)

attention\_plot[i] = tf.reshape(attention\_weights, (-1, )).numpy()

predicted\_id = tf.multinomial(predictions, num\_samples=1)[0][0].numpy()

result.append(index\_word[predicted\_id])

**if** index\_word[predicted\_id] == '<end>':

**return** result, attention\_plot

dec\_input = tf.expand\_dims([predicted\_id], 0)

attention\_plot = attention\_plot[:len(result), :]

**return** result, attention\_plot

In [0]:

**def** plot\_attention(image, result, attention\_plot):

temp\_image = np.array(Image.open(image))

fig = plt.figure(figsize=(10, 10))

len\_result = len(result)

**for** l **in** range(len\_result):

temp\_att = np.resize(attention\_plot[l], (8, 8))

ax = fig.add\_subplot(len\_result//2, len\_result//2, l+1)

ax.set\_title(result[l])

img = ax.imshow(temp\_image)

ax.imshow(temp\_att, cmap='gray', alpha=0.6, extent=img.get\_extent())

plt.tight\_layout()

plt.show()

In [0]:

*# captions on the validation set*

rid = np.random.randint(0, len(img\_name\_val))

image = img\_name\_val[rid]

real\_caption = ' '.join([index\_word[i] **for** i **in** cap\_val[rid] **if** i **not** **in** [0]])

result, attention\_plot = evaluate(image)

print ('Real Caption:', real\_caption)

print ('Prediction Caption:', ' '.join(result))

plot\_attention(image, result, attention\_plot)

*# opening the image*

Image.open(img\_name\_val[rid])

## 在你自己的图像上试试

为了好玩，下面我们提供了一种方法，您可以使用我们刚训练过的模型为您自己的图像添加标题。请记住，它是在相对少量的数据上训练的，您的图像可能与训练数据不同（因此请为奇怪的结果做好准备！）

In [0]:

image\_url = 'https://tensorflow.org/images/surf.jpg'

image\_extension = image\_url[-4:]

image\_path = tf.keras.utils.get\_file('image'+image\_extension,

origin=image\_url)

result, attention\_plot = evaluate(image\_path)

print ('Prediction Caption:', ' '.join(result))

plot\_attention(image\_path, result, attention\_plot)

*# opening the image*

Image.open(image\_path)

## 后续步骤

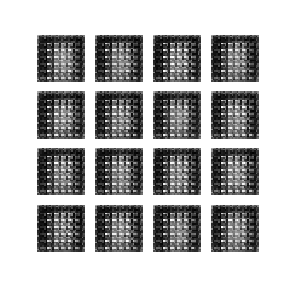
恭喜！你刚刚训练了一个注意图像字幕模型。接下来，我们建议您查看此示例[神经机器翻译与注意](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb)。它使用类似的架构来翻译西班牙语和英语句子。您还可以尝试在不同的数据集上训练此笔记本中的代码。

# DCGAN：tf.keras和eager的一个例子

这个笔记本演示了如何使用[tf.keras](https://www.tensorflow.org/programmers_guide/keras)和[急切执行](https://www.tensorflow.org/programmers_guide/eager)生成手写数字的图像。为此，我们使用Deep Convolutional Generative Adverserial Networks（[DCGAN](https://arxiv.org/pdf/1511.06434.pdf)）。

从2018年7月开始，该模型每个时期约需30秒（使用tf.contrib.eager.defun创建图形函数）以在Colab上的单个特斯拉K80上进行训练。

下面是训练150个时期的发生器和鉴别器模型后生成的输出。



In [0]:

*# to generate gifs*

!pip install imageio

## 导入TensorFlow并启用急切执行

In [0]:

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function

*# Import TensorFlow >= 1.10 and enable eager execution*

**import** **tensorflow** **as** **tf**

tf.enable\_eager\_execution()

**import** **os**

**import** **time**

**import** **numpy** **as** **np**

**import** **glob**

**import** **matplotlib.pyplot** **as** **plt**

**import** **PIL**

**import** **imageio**

**from** **IPython** **import** display

## 加载数据集

我们将使用MNIST数据集来训练发生器和鉴别器。然后，生成器将生成手写数字。

In [0]:

(train\_images, train\_labels), (\_, \_) = tf.keras.datasets.mnist.load\_data()

In [0]:

train\_images = train\_images.reshape(train\_images.shape[0], 28, 28, 1).astype('float32')

*# We are normalizing the images to the range of [-1, 1]*

train\_images = (train\_images - 127.5) / 127.5

In [0]:

BUFFER\_SIZE = 60000

BATCH\_SIZE = 256

## 使用tf.data创建批处理并对数据集进行混洗

In [0]:

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

**写发生器和鉴别器模型**

* **发电机**
  + 它负责**创造令人信服的图像，足以愚弄鉴别器**。
  + 它由Conv2DTranspose（Upsampling）层组成。我们从完全连接的层开始，并对图像进行2次上采样，以便达到所需的图像尺寸（mnist图像尺寸），即（28,28,1）。
  + 我们使用**漏泄relu**激活，除了使用**tanh**激活的**最后一层**。
* **鉴别**
  + **鉴别器负责对来自真实图像的伪图像进行分类。**
  + 换句话说，鉴别器被给予生成的图像（来自生成器）和真实的MNIST图像。鉴别器的工作是将这些图像分类为伪（生成）和真实（MNIST图像）。
  + **基本上，生成器应该足以欺骗生成的图像是真实的鉴别器**。

In [0]:

**class** **Generator**(tf.keras.Model):

**def** \_\_init\_\_(self):

super(Generator, self).\_\_init\_\_()

self.fc1 = tf.keras.layers.Dense(7\*7\*64, use\_bias=**False**)

self.batchnorm1 = tf.keras.layers.BatchNormalization()

self.conv1 = tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(1, 1), padding='same', use\_bias=**False**)

self.batchnorm2 = tf.keras.layers.BatchNormalization()

self.conv2 = tf.keras.layers.Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same', use\_bias=**False**)

self.batchnorm3 = tf.keras.layers.BatchNormalization()

self.conv3 = tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=**False**)

**def** call(self, x, training=**True**):

x = self.fc1(x)

x = self.batchnorm1(x, training=training)

x = tf.nn.relu(x)

x = tf.reshape(x, shape=(-1, 7, 7, 64))

x = self.conv1(x)

x = self.batchnorm2(x, training=training)

x = tf.nn.relu(x)

x = self.conv2(x)

x = self.batchnorm3(x, training=training)

x = tf.nn.relu(x)

x = tf.nn.tanh(self.conv3(x))

**return** x

In [0]:

**class** **Discriminator**(tf.keras.Model):

**def** \_\_init\_\_(self):

super(Discriminator, self).\_\_init\_\_()

self.conv1 = tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same')

self.conv2 = tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')

self.dropout = tf.keras.layers.Dropout(0.3)

self.flatten = tf.keras.layers.Flatten()

self.fc1 = tf.keras.layers.Dense(1)

**def** call(self, x, training=**True**):

x = tf.nn.leaky\_relu(self.conv1(x))

x = self.dropout(x, training=training)

x = tf.nn.leaky\_relu(self.conv2(x))

x = self.dropout(x, training=training)

x = self.flatten(x)

x = self.fc1(x)

**return** x

In [0]:

generator = Generator()

discriminator = Discriminator()

In [0]:

*# Defun gives 10 secs/epoch performance boost*

generator.call = tf.contrib.eager.defun(generator.call)

discriminator.call = tf.contrib.eager.defun(discriminator.call)

**定义损失函数和优化器**

* **判别者损失**
  + 鉴别器丢失功能需要2个输入; **真实图像，生成图像**
  + real\_loss是**真实图像**的sigmoid交叉熵损失和一个**数组（因为这些是真实的图像）**
  + generated\_loss是**生成的图像**的sigmoid交叉熵丢失和**零**的**数组（因为这些是伪图像）**
  + 然后total\_loss是real\_loss和generated\_loss的总和
* **发电机损耗**
  + 它是生成的图像的sigmoid交叉熵损失和一**组数组**
* 鉴别器和生成器优化器是不同的，因为我们将单独训练它们。

In [0]:

**def** discriminator\_loss(real\_output, generated\_output):

*# [1,1,...,1] with real output since it is true and we want*

*# our generated examples to look like it*

real\_loss = tf.losses.sigmoid\_cross\_entropy(multi\_class\_labels=tf.ones\_like(real\_output), logits=real\_output)

*# [0,0,...,0] with generated images since they are fake*

generated\_loss = tf.losses.sigmoid\_cross\_entropy(multi\_class\_labels=tf.zeros\_like(generated\_output), logits=generated\_output)

total\_loss = real\_loss + generated\_loss

**return** total\_loss

In [0]:

**def** generator\_loss(generated\_output):

**return** tf.losses.sigmoid\_cross\_entropy(tf.ones\_like(generated\_output), generated\_output)

In [0]:

discriminator\_optimizer = tf.train.AdamOptimizer(1e-4)

generator\_optimizer = tf.train.AdamOptimizer(1e-4)

## 检查点（基于对象的保存）

In [0]:

checkpoint\_dir = './training\_checkpoints'

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

checkpoint = tf.train.Checkpoint(generator\_optimizer=generator\_optimizer,

discriminator\_optimizer=discriminator\_optimizer,

generator=generator,

discriminator=discriminator)

**Training**

* 我们首先迭代数据集
* 生成器被赋予**噪声作为输入**，当通过生成器模型时将输出看起来像手写数字的图像
* 鉴别器被给予**真实的MNIST图像以及生成的图像（来自发生器）**。
* 接下来，我们计算发电机和鉴别器损耗。
* 然后，我们计算相对于发生器和鉴别器变量（输入）的损耗梯度，并将它们应用于优化器。

**生成图像**

* 经过培训，它有时间生成一些图像！
* 我们首先创建噪声数组作为生成器的输入
* 然后，生成器将噪声转换为手写图像。
* 最后一步是绘制预测和**瞧！**

In [0]:

EPOCHS = 150

noise\_dim = 100

num\_examples\_to\_generate = 16

*# keeping the random vector constant for generation (prediction) so*

*# it will be easier to see the improvement of the gan.*

random\_vector\_for\_generation = tf.random\_normal([num\_examples\_to\_generate,

noise\_dim])

In [0]:

**def** generate\_and\_save\_images(model, epoch, test\_input):

*# make sure the training parameter is set to False because we*

*# don't want to train the batchnorm layer when doing inference.*

predictions = model(test\_input, training=**False**)

fig = plt.figure(figsize=(4,4))

**for** i **in** range(predictions.shape[0]):

plt.subplot(4, 4, i+1)

plt.imshow(predictions[i, :, :, 0] \* 127.5 + 127.5, cmap='gray')

plt.axis('off')

plt.savefig('image\_at\_epoch\_**{:04d}**.png'.format(epoch))

plt.show()

In [0]:

**def** train(dataset, epochs, noise\_dim):

**for** epoch **in** range(epochs):

start = time.time()

**for** images **in** dataset:

*# generating noise from a uniform distribution*

noise = tf.random\_normal([BATCH\_SIZE, noise\_dim])

**with** tf.GradientTape() **as** gen\_tape, tf.GradientTape() **as** disc\_tape:

generated\_images = generator(noise, training=**True**)

real\_output = discriminator(images, training=**True**)

generated\_output = discriminator(generated\_images, training=**True**)

gen\_loss = generator\_loss(generated\_output)

disc\_loss = discriminator\_loss(real\_output, generated\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.variables)

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.variables)

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.variables))

**if** epoch % 1 == 0:

display.clear\_output(wait=**True**)

generate\_and\_save\_images(generator,

epoch + 1,

random\_vector\_for\_generation)

*# saving (checkpoint) the model every 15 epochs*

**if** (epoch + 1) % 15 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)

print ('Time taken for epoch **{}** is **{}** sec'.format(epoch + 1,

time.time()-start))

*# generating after the final epoch*

display.clear\_output(wait=**True**)

generate\_and\_save\_images(generator,

epochs,

random\_vector\_for\_generation)

In [0]:

train(train\_dataset, EPOCHS, noise\_dim)

## 恢复最新的检查点

In [0]:

*# restoring the latest checkpoint in checkpoint\_dir*

checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

## 使用纪元号显示图像

In [0]:

**def** display\_image(epoch\_no):

**return** PIL.Image.open('image\_at\_epoch\_**{:04d}**.png'.format(epoch\_no))

In [0]:

display\_image(EPOCHS)

## 生成所有已保存图像的GIF。

In [0]:

**with** imageio.get\_writer('dcgan.gif', mode='I') **as** writer:

filenames = glob.glob('image\*.png')

filenames = sorted(filenames)

last = -1

**for** i,filename **in** enumerate(filenames):

frame = 2\*(i\*\*0.5)

**if** round(frame) > round(last):

last = frame

**else**:

**continue**

image = imageio.imread(filename)

writer.append\_data(image)

image = imageio.imread(filename)

writer.append\_data(image)

*# this is a hack to display the gif inside the notebook*

os.system('cp dcgan.gif dcgan.gif.png')

In [0]:

display.Image(filename="dcgan.gif.png")

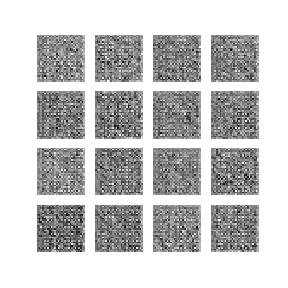
要从Colab下载动画，请取消注释以下代码：

In [0]:

*#from google.colab import files*

*#files.download('dcgan.gif')*

# 卷积VAE：tf.keras和eager的一个例子[¶](https://render.githubusercontent.com/view/ipynb?commit=c19e29306ce1777456b2dbb3a14f511edf7883a8&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f74656e736f72666c6f772f74656e736f72666c6f772f633139653239333036636531373737343536623264626233613134663531316564663738383361382f74656e736f72666c6f772f636f6e747269622f65616765722f707974686f6e2f6578616d706c65732f67656e657261746976655f6578616d706c65732f637661652e6970796e62&nwo=tensorflow%2Ftensorflow&path=tensorflow%2Fcontrib%2Feager%2Fpython%2Fexamples%2Fgenerative_examples%2Fcvae.ipynb&repository_id=45717250&repository_type=Repository#Convolutional-VAE:-An-example-with-tf.keras-and-eager)



这个笔记本演示了如何使用[tf.keras](https://www.tensorflow.org/programmers_guide/keras)生成手写数字的图像，并通过训练变分自动编码器来[执行急切的执行](https://www.tensorflow.org/programmers_guide/eager)。（VAE，[[1]](https://arxiv.org/abs/1312.6114)，[[2]](https://arxiv.org/abs/1401.4082)）。

In [0]:

*# to generate gifs*

!pip install imageio

## 导入TensorFlow并启用Eager执行

In [0]:

**from** **\_\_future\_\_** **import** absolute\_import, division, print\_function

*# Import TensorFlow >= 1.9 and enable eager execution*

**import** **tensorflow** **as** **tf**

tfe = tf.contrib.eager

tf.enable\_eager\_execution()

**import** **os**

**import** **time**

**import** **numpy** **as** **np**

**import** **glob**

**import** **matplotlib.pyplot** **as** **plt**

**import** **PIL**

**import** **imageio**

**from** **IPython** **import** display

## 加载MNIST数据集

每个MNIST图像最初是784个整数的矢量，每个整数在0-255之间并且表示像素的强度。我们在模型中使用伯努利分布对每个像素进行建模，并对数据集进行静态二值化。

In [0]:

(train\_images, \_), (test\_images, \_) = tf.keras.datasets.mnist.load\_data()

In [0]:

train\_images = train\_images.reshape(train\_images.shape[0], 28, 28, 1).astype('float32')

test\_images = test\_images.reshape(test\_images.shape[0], 28, 28, 1).astype('float32')

*# Normalizing the images to the range of [0., 1.]*

train\_images /= 255.

test\_images /= 255.

*# Binarization*

train\_images[train\_images >= .5] = 1.

train\_images[train\_images < .5] = 0.

test\_images[test\_images >= .5] = 1.

test\_images[test\_images < .5] = 0.

In [0]:

TRAIN\_BUF = 60000

BATCH\_SIZE = 100

TEST\_BUF = 10000

## 使用tf.data创建批处理并对数据集进行混洗

In [0]:

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(TRAIN\_BUF).batch(BATCH\_SIZE)

test\_dataset = tf.data.Dataset.from\_tensor\_slices(test\_images).shuffle(TEST\_BUF).batch(BATCH\_SIZE)

## 用tf.keras连接生成和推理网络。顺序

在我们的VAE示例中，我们使用两个小型ConvNets用于生成和推理网络。由于这些神经网络很小，我们tf.keras.Sequential用来简化代码。让和分别表示在下面描述中的观察和潜在变量。

### 生成网络

这定义了生成模型，其采用潜在编码作为输入，并输出用于观察的条件分布的参数，即。另外，我们对潜在变量使用单位高斯先验。

### 推理网络[¶](https://render.githubusercontent.com/view/ipynb?commit=c19e29306ce1777456b2dbb3a14f511edf7883a8&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f74656e736f72666c6f772f74656e736f72666c6f772f633139653239333036636531373737343536623264626233613134663531316564663738383361382f74656e736f72666c6f772f636f6e747269622f65616765722f707974686f6e2f6578616d706c65732f67656e657261746976655f6578616d706c65732f637661652e6970796e62&nwo=tensorflow%2Ftensorflow&path=tensorflow%2Fcontrib%2Feager%2Fpython%2Fexamples%2Fgenerative_examples%2Fcvae.ipynb&repository_id=45717250&repository_type=Repository#Inference-Network)

这定义了近似后验分布，其将观察作为输入并输出用于潜在表示的条件分布的一组参数。在这个例子中，我们简单地将这个分布建模为对角高斯分布。在这种情况下，推理网络输出分解高斯的均值和对数方差参数（对数方差而不是方差直接用于数值稳定性）。

### 重新参数化技巧

在优化过程中，我们可以通过首先从单位高斯采样，然后乘以标准偏差并加上平均值来进行采样。这确保了梯度可以通过样本传递到推理网络参数。

### 网络架构

对于推理网络，我们使用两个卷积层，然后是完全连接的层。在生成网络中，我们通过使用完全连接的层，然后是三个卷积转置层（在某些上下文中也称为反卷积层）来镜像这种架构。请注意，这是常见的做法，以避免使用批标准化培训VAES时，由于额外的随机性，由于使用mini-批次可以从抽样的随机性的顶部加剧不稳定。

In [0]:

**class** **CVAE**(tf.keras.Model):

**def** \_\_init\_\_(self, latent\_dim):

super(CVAE, self).\_\_init\_\_()

self.latent\_dim = latent\_dim

self.inference\_net = tf.keras.Sequential(

[

tf.keras.layers.InputLayer(input\_shape=(28, 28, 1)),

tf.keras.layers.Conv2D(

filters=32, kernel\_size=3, strides=(2, 2), activation=tf.nn.relu),

tf.keras.layers.Conv2D(

filters=64, kernel\_size=3, strides=(2, 2), activation=tf.nn.relu),

tf.keras.layers.Flatten(),

*# No activation*

tf.keras.layers.Dense(latent\_dim + latent\_dim),

]

)

self.generative\_net = tf.keras.Sequential(

[

tf.keras.layers.InputLayer(input\_shape=(latent\_dim,)),

tf.keras.layers.Dense(units=7\*7\*32, activation=tf.nn.relu),

tf.keras.layers.Reshape(target\_shape=(7, 7, 32)),

tf.keras.layers.Conv2DTranspose(

filters=64,

kernel\_size=3,

strides=(2, 2),

padding="SAME",

activation=tf.nn.relu),

tf.keras.layers.Conv2DTranspose(

filters=32,

kernel\_size=3,

strides=(2, 2),

padding="SAME",

activation=tf.nn.relu),

*# No activation*

tf.keras.layers.Conv2DTranspose(

filters=1, kernel\_size=3, strides=(1, 1), padding="SAME"),

]

)

**def** sample(self, eps=**None**):

**if** eps **is** **None**:

eps = tf.random\_normal(shape=(100, self.latent\_dim))

**return** self.decode(eps, apply\_sigmoid=**True**)

**def** encode(self, x):

mean, logvar = tf.split(self.inference\_net(x), num\_or\_size\_splits=2, axis=1)

**return** mean, logvar

**def** reparameterize(self, mean, logvar):

eps = tf.random\_normal(shape=mean.shape)

**return** eps \* tf.exp(logvar \* .5) + mean

**def** decode(self, z, apply\_sigmoid=**False**):

logits = self.generative\_net(z)

**if** apply\_sigmoid:

probs = tf.sigmoid(logits)

**return** probs

**return** logits

## 定义损失函数和优化器

VAE通过最大化边际对数似然下的证据下界（ELBO）进行训练：



在实践中，我们优化了单个样本蒙特卡罗对此期望的估计：



**注意**：我们也可以分析计算KL项，但是为了简单起见，我们在蒙特卡罗估计中将所有三个项合并。  
In [0]:

**def** log\_normal\_pdf(sample, mean, logvar, raxis=1):

log2pi = tf.log(2. \* np.pi)

**return** tf.reduce\_sum(

-.5 \* ((sample - mean) \*\* 2. \* tf.exp(-logvar) + logvar + log2pi),

axis=raxis)

**def** compute\_loss(model, x):

mean, logvar = model.encode(x)

z = model.reparameterize(mean, logvar)

x\_logit = model.decode(z)

cross\_ent = tf.nn.sigmoid\_cross\_entropy\_with\_logits(logits=x\_logit, labels=x)

logpx\_z = -tf.reduce\_sum(cross\_ent, axis=[1, 2, 3])

logpz = log\_normal\_pdf(z, 0., 0.)

logqz\_x = log\_normal\_pdf(z, mean, logvar)

**return** -tf.reduce\_mean(logpx\_z + logpz - logqz\_x)

**def** compute\_gradients(model, x):

**with** tf.GradientTape() **as** tape:

loss = compute\_loss(model, x)

**return** tape.gradient(loss, model.trainable\_variables), loss

optimizer = tf.train.AdamOptimizer(1e-4)

**def** apply\_gradients(optimizer, gradients, variables, global\_step=**None**):

optimizer.apply\_gradients(zip(gradients, variables), global\_step=global\_step)

**Training**

* 我们首先迭代数据集
* 在每次迭代期间，我们将图像传递给编码器以获得近似后验的一组均值和对数方差参数
* 然后我们将*重新参数化技巧*应用于样本
* 最后，我们将重新参数化的样本传递给解码器以获得生成分布的logits
* **注意：**由于我们使用keras加载的数据集，训练集中有60k数据点，测试集中有10k数据点，因此我们在测试集上得到的ELBO略高于使用Larochelle的MNIST动态二值化的文献中的报告结果。

**生成图像**

* 训练结束后，是时候生成一些图像了
* 我们首先从单位高斯先验分布中采样一组潜在向量
* 然后，生成器将潜在样本转换为观察的对数，给出分布
* 在这里，我们绘制伯努利分布的概率

In [0]:

epochs = 100

latent\_dim = 50

num\_examples\_to\_generate = 16

*# keeping the random vector constant for generation (prediction) so*

*# it will be easier to see the improvement.*

random\_vector\_for\_generation = tf.random\_normal(

shape=[num\_examples\_to\_generate, latent\_dim])

model = CVAE(latent\_dim)

In [0]:

**def** generate\_and\_save\_images(model, epoch, test\_input):

predictions = model.sample(test\_input)

fig = plt.figure(figsize=(4,4))

**for** i **in** range(predictions.shape[0]):

plt.subplot(4, 4, i+1)

plt.imshow(predictions[i, :, :, 0], cmap='gray')

plt.axis('off')

*# tight\_layout minimizes the overlap between 2 sub-plots*

plt.savefig('image\_at\_epoch\_**{:04d}**.png'.format(epoch))

plt.show()

In [0]:

generate\_and\_save\_images(model, 0, random\_vector\_for\_generation)

**for** epoch **in** range(1, epochs + 1):

start\_time = time.time()

**for** train\_x **in** train\_dataset:

gradients, loss = compute\_gradients(model, train\_x)

apply\_gradients(optimizer, gradients, model.trainable\_variables)

end\_time = time.time()

**if** epoch % 1 == 0:

loss = tfe.metrics.Mean()

**for** test\_x **in** test\_dataset.make\_one\_shot\_iterator():

loss(compute\_loss(model, test\_x))

elbo = -loss.result()

display.clear\_output(wait=**False**)

print('Epoch: **{}**, Test set ELBO: **{}**, '

'time elapse for current epoch **{}**'.format(epoch,

elbo,

end\_time - start\_time))

generate\_and\_save\_images(

model, epoch, random\_vector\_for\_generation)

### 使用纪元号显示图像

In [0]:

**def** display\_image(epoch\_no):

**return** PIL.Image.open('image\_at\_epoch\_**{:04d}**.png'.format(epoch\_no))

In [0]:

display\_image(epochs) *# Display images*

### 生成所有已保存图像的GIF。

In [0]:

**with** imageio.get\_writer('cvae.gif', mode='I') **as** writer:

filenames = glob.glob('image\*.png')

filenames = sorted(filenames)

last = -1

**for** i,filename **in** enumerate(filenames):

frame = 2\*(i\*\*0.5)

**if** round(frame) > round(last):

last = frame

**else**:

**continue**

image = imageio.imread(filename)

writer.append\_data(image)

image = imageio.imread(filename)

writer.append\_data(image)

*# this is a hack to display the gif inside the notebook*

os.system('cp cvae.gif cvae.gif.png')

In [0]:

display.Image(filename="cvae.gif.png")

要从Colab下载动画，请取消注释以下代码：

In [0]:

*#from google.colab import files*

*#files.download('cvae.gif')*

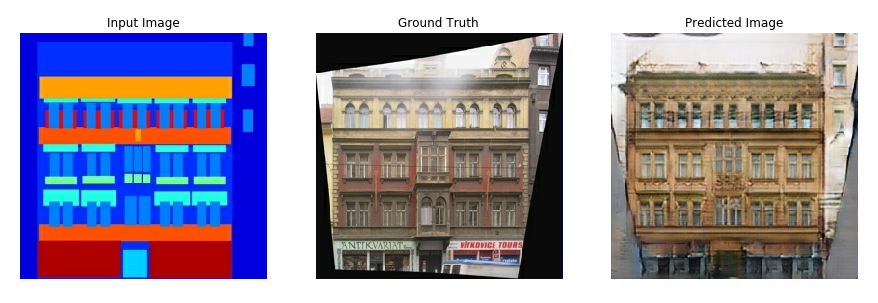
# Pix2Pix：tf.keras和eager的一个例子

该笔记本使用条件GAN演示图像到图像的转换，如[使用条件对抗网络的](https://arxiv.org/abs/1611.07004)图像到图像转换中所述。使用这种技术，我们可以着色黑白照片，将谷歌地图转换为谷歌地球等。在这里，我们将建筑物外墙转换为真实建筑物。我们使用[tf.keras](https://www.tensorflow.org/programmers_guide/keras)和[eager执行](https://www.tensorflow.org/programmers_guide/eager)来实现这一目标。

在例子中，我们将使用[CMP门面数据库](http://cmp.felk.cvut.cz/~tylecr1/facade/)，通过提供有益[中心的机器感知](http://cmp.felk.cvut.cz/)在[布拉格捷克技术大学](https://www.cvut.cz/)。为了简化我们的示例，我们将使用由上述[论文](https://arxiv.org/abs/1611.07004)的作者创建的该数据集的预处理[副本](https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/)。

单个P100 GPU上的每个纪元大约需要58秒。

以下是训练200个时期模型后生成的输出。



## 导入TensorFlow并启用急切执行

In [0]:

*# Import TensorFlow >= 1.10 and enable eager execution*

**import** **tensorflow** **as** **tf**

tf.enable\_eager\_execution()

**import** **os**

**import** **time**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**import** **PIL**

**from** **IPython.display** **import** clear\_output

## 加载数据集

您可以从[此处](https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets)下载此数据集和类似数据集。如[本文](https://arxiv.org/abs/1611.07004)所述，我们将随机抖动和镜像应用于训练数据集。

* 在随机抖动中，图像被调整大小286 x 286，然后随机裁剪为256 x 256
* 在随机镜像中，图像随机水平翻转，即从左到右。

In [0]:

path\_to\_zip = tf.keras.utils.get\_file('facades.tar.gz',

cache\_subdir=os.path.abspath('.'),

origin='https://people.eecs.berkeley.edu/~tinghuiz/projects/pix2pix/datasets/facades.tar.gz',

extract=**True**)

PATH = os.path.join(os.path.dirname(path\_to\_zip), 'facades/')

In [0]:

BUFFER\_SIZE = 400

BATCH\_SIZE = 1

IMG\_WIDTH = 256

IMG\_HEIGHT = 256

In [0]:

**def** load\_image(image\_file, is\_train):

image = tf.read\_file(image\_file)

image = tf.image.decode\_jpeg(image)

w = tf.shape(image)[1]

w = w // 2

real\_image = image[:, :w, :]

input\_image = image[:, w:, :]

input\_image = tf.cast(input\_image, tf.float32)

real\_image = tf.cast(real\_image, tf.float32)

**if** is\_train:

*# random jittering*

*# resizing to 286 x 286 x 3*

input\_image = tf.image.resize\_images(input\_image, [286, 286],

align\_corners=**True**,

method=tf.image.ResizeMethod.NEAREST\_NEIGHBOR)

real\_image = tf.image.resize\_images(real\_image, [286, 286],

align\_corners=**True**,

method=tf.image.ResizeMethod.NEAREST\_NEIGHBOR)

*# randomly cropping to 256 x 256 x 3*

stacked\_image = tf.stack([input\_image, real\_image], axis=0)

cropped\_image = tf.random\_crop(stacked\_image, size=[2, IMG\_HEIGHT, IMG\_WIDTH, 3])

input\_image, real\_image = cropped\_image[0], cropped\_image[1]

**if** np.random.random() > 0.5:

*# random mirroring*

input\_image = tf.image.flip\_left\_right(input\_image)

real\_image = tf.image.flip\_left\_right(real\_image)

**else**:

input\_image = tf.image.resize\_images(input\_image, size=[IMG\_HEIGHT, IMG\_WIDTH],

align\_corners=**True**, method=2)

real\_image = tf.image.resize\_images(real\_image, size=[IMG\_HEIGHT, IMG\_WIDTH],

align\_corners=**True**, method=2)

*# normalizing the images to [-1, 1]*

input\_image = (input\_image / 127.5) - 1

real\_image = (real\_image / 127.5) - 1

**return** input\_image, real\_image

## 使用tf.data创建批处理，映射（执行预处理）和随机数据集

In [0]:

train\_dataset = tf.data.Dataset.list\_files(PATH+'train/\*.jpg')

train\_dataset = train\_dataset.shuffle(BUFFER\_SIZE)

train\_dataset = train\_dataset.map(**lambda** x: load\_image(x, **True**))

train\_dataset = train\_dataset.batch(1)

In [0]:

test\_dataset = tf.data.Dataset.list\_files(PATH+'test/\*.jpg')

test\_dataset = test\_dataset.map(**lambda** x: load\_image(x, **False**))

test\_dataset = test\_dataset.batch(1)

## 写发生器和鉴别器模型

* **发电机**
  + 发电机的结构是改进的U-Net。
  + 编码器中的每个块都是（Conv - > Batchnorm - > Leaky ReLU）
  + 解码器中的每个块是（Transposed Conv - > Batchnorm - > Dropout（应用于前3个块） - > ReLU）
  + 编码器和解码器之间存在跳过连接（如在U-Net中）。
* **鉴别**
  + 鉴别器是PatchGAN。
  + 鉴别器中的每个块都是（Conv - > BatchNorm - > Leaky ReLU）
  + 最后一层之后的输出形状为（batch\_size，30,30,1）
  + 输出的每个30x30补丁分类输入图像的70x70部分（这种架构称为PatchGAN）。
  + 鉴别器接收2个输入。
    - 输入图像和目标图像，它应该归类为真实的。
    - 输入图像和生成的图像（生成器的输出），它应该归类为伪造的。
    - 我们在代码（tf.concat([inp, tar], axis=-1)）中将这两个输入连接在一起
* 通过发生器和鉴别器的输入的形状在代码中的注释中。

要了解有关体系结构和超参数的更多信息，请参阅[本文](https://arxiv.org/abs/1611.07004)。

In [0]:

OUTPUT\_CHANNELS = 3

In [0]:

**class** **Downsample**(tf.keras.Model):

**def** \_\_init\_\_(self, filters, size, apply\_batchnorm=**True**):

super(Downsample, self).\_\_init\_\_()

self.apply\_batchnorm = apply\_batchnorm

initializer = tf.random\_normal\_initializer(0., 0.02)

self.conv1 = tf.keras.layers.Conv2D(filters,

(size, size),

strides=2,

padding='same',

kernel\_initializer=initializer,

use\_bias=**False**)

**if** self.apply\_batchnorm:

self.batchnorm = tf.keras.layers.BatchNormalization()

**def** call(self, x, training):

x = self.conv1(x)

**if** self.apply\_batchnorm:

x = self.batchnorm(x, training=training)

x = tf.nn.leaky\_relu(x)

**return** x

**class** **Upsample**(tf.keras.Model):

**def** \_\_init\_\_(self, filters, size, apply\_dropout=**False**):

super(Upsample, self).\_\_init\_\_()

self.apply\_dropout = apply\_dropout

initializer = tf.random\_normal\_initializer(0., 0.02)

self.up\_conv = tf.keras.layers.Conv2DTranspose(filters,

(size, size),

strides=2,

padding='same',

kernel\_initializer=initializer,

use\_bias=**False**)

self.batchnorm = tf.keras.layers.BatchNormalization()

**if** self.apply\_dropout:

self.dropout = tf.keras.layers.Dropout(0.5)

**def** call(self, x1, x2, training):

x = self.up\_conv(x1)

x = self.batchnorm(x, training=training)

**if** self.apply\_dropout:

x = self.dropout(x, training=training)

x = tf.nn.relu(x)

x = tf.concat([x, x2], axis=-1)

**return** x

**class** **Generator**(tf.keras.Model):

**def** \_\_init\_\_(self):

super(Generator, self).\_\_init\_\_()

initializer = tf.random\_normal\_initializer(0., 0.02)

self.down1 = Downsample(64, 4, apply\_batchnorm=**False**)

self.down2 = Downsample(128, 4)

self.down3 = Downsample(256, 4)

self.down4 = Downsample(512, 4)

self.down5 = Downsample(512, 4)

self.down6 = Downsample(512, 4)

self.down7 = Downsample(512, 4)

self.down8 = Downsample(512, 4)

self.up1 = Upsample(512, 4, apply\_dropout=**True**)

self.up2 = Upsample(512, 4, apply\_dropout=**True**)

self.up3 = Upsample(512, 4, apply\_dropout=**True**)

self.up4 = Upsample(512, 4)

self.up5 = Upsample(256, 4)

self.up6 = Upsample(128, 4)

self.up7 = Upsample(64, 4)

self.last = tf.keras.layers.Conv2DTranspose(OUTPUT\_CHANNELS,

(4, 4),

strides=2,

padding='same',

kernel\_initializer=initializer)

@tf.contrib.eager.defun

**def** call(self, x, training):

*# x shape == (bs, 256, 256, 3)*

x1 = self.down1(x, training=training) *# (bs, 128, 128, 64)*

x2 = self.down2(x1, training=training) *# (bs, 64, 64, 128)*

x3 = self.down3(x2, training=training) *# (bs, 32, 32, 256)*

x4 = self.down4(x3, training=training) *# (bs, 16, 16, 512)*

x5 = self.down5(x4, training=training) *# (bs, 8, 8, 512)*

x6 = self.down6(x5, training=training) *# (bs, 4, 4, 512)*

x7 = self.down7(x6, training=training) *# (bs, 2, 2, 512)*

x8 = self.down8(x7, training=training) *# (bs, 1, 1, 512)*

x9 = self.up1(x8, x7, training=training) *# (bs, 2, 2, 1024)*

x10 = self.up2(x9, x6, training=training) *# (bs, 4, 4, 1024)*

x11 = self.up3(x10, x5, training=training) *# (bs, 8, 8, 1024)*

x12 = self.up4(x11, x4, training=training) *# (bs, 16, 16, 1024)*

x13 = self.up5(x12, x3, training=training) *# (bs, 32, 32, 512)*

x14 = self.up6(x13, x2, training=training) *# (bs, 64, 64, 256)*

x15 = self.up7(x14, x1, training=training) *# (bs, 128, 128, 128)*

x16 = self.last(x15) *# (bs, 256, 256, 3)*

x16 = tf.nn.tanh(x16)

**return** x16

In [0]:

**class** **DiscDownsample**(tf.keras.Model):

**def** \_\_init\_\_(self, filters, size, apply\_batchnorm=**True**):

super(DiscDownsample, self).\_\_init\_\_()

self.apply\_batchnorm = apply\_batchnorm

initializer = tf.random\_normal\_initializer(0., 0.02)

self.conv1 = tf.keras.layers.Conv2D(filters,

(size, size),

strides=2,

padding='same',

kernel\_initializer=initializer,

use\_bias=**False**)

**if** self.apply\_batchnorm:

self.batchnorm = tf.keras.layers.BatchNormalization()

**def** call(self, x, training):

x = self.conv1(x)

**if** self.apply\_batchnorm:

x = self.batchnorm(x, training=training)

x = tf.nn.leaky\_relu(x)

**return** x

**class** **Discriminator**(tf.keras.Model):

**def** \_\_init\_\_(self):

super(Discriminator, self).\_\_init\_\_()

initializer = tf.random\_normal\_initializer(0., 0.02)

self.down1 = DiscDownsample(64, 4, **False**)

self.down2 = DiscDownsample(128, 4)

self.down3 = DiscDownsample(256, 4)

*# we are zero padding here with 1 because we need our shape to*

*# go from (batch\_size, 32, 32, 256) to (batch\_size, 31, 31, 512)*

self.zero\_pad1 = tf.keras.layers.ZeroPadding2D()

self.conv = tf.keras.layers.Conv2D(512,

(4, 4),

strides=1,

kernel\_initializer=initializer,

use\_bias=**False**)

self.batchnorm1 = tf.keras.layers.BatchNormalization()

*# shape change from (batch\_size, 31, 31, 512) to (batch\_size, 30, 30, 1)*

self.zero\_pad2 = tf.keras.layers.ZeroPadding2D()

self.last = tf.keras.layers.Conv2D(1,

(4, 4),

strides=1,

kernel\_initializer=initializer)

@tf.contrib.eager.defun

**def** call(self, inp, tar, training):

*# concatenating the input and the target*

x = tf.concat([inp, tar], axis=-1) *# (bs, 256, 256, channels\*2)*

x = self.down1(x, training=training) *# (bs, 128, 128, 64)*

x = self.down2(x, training=training) *# (bs, 64, 64, 128)*

x = self.down3(x, training=training) *# (bs, 32, 32, 256)*

x = self.zero\_pad1(x) *# (bs, 34, 34, 256)*

x = self.conv(x) *# (bs, 31, 31, 512)*

x = self.batchnorm1(x, training=training)

x = tf.nn.leaky\_relu(x)

x = self.zero\_pad2(x) *# (bs, 33, 33, 512)*

*# don't add a sigmoid activation here since*

*# the loss function expects raw logits.*

x = self.last(x) *# (bs, 30, 30, 1)*

**return** x

In [0]:

*# The call function of Generator and Discriminator have been decorated*

*# with tf.contrib.eager.defun()*

*# We get a performance speedup if defun is used (~25 seconds per epoch)*

generator = Generator()

discriminator = Discriminator()

## 定义损失函数和优化器

* **判别者损失**
  + 鉴别器丢失功能需要2个输入; **真实图像，生成图像**
  + real\_loss是**真实图像**的sigmoid交叉熵损失和一个**数组（因为这些是真实的图像）**
  + generated\_loss是**生成的图像**的sigmoid交叉熵丢失和**零**的**数组（因为这些是伪图像）**
  + 然后total\_loss是real\_loss和generated\_loss的总和
* **发电机损耗**
  + 它是生成的图像的sigmoid交叉熵损失和一**组数组**。
  + 该[论文](https://arxiv.org/abs/1611.07004)还包括L1损失，其是生成的图像和目标图像之间的MAE（平均绝对误差）。
  + 这允许生成的图像在结构上与目标图像相似。
  + 式来计算总发电机损失= gan\_loss + LAMBDA \* l1\_loss，其中LAMBDA = 100。这个值是由的作者决定[纸](https://arxiv.org/abs/1611.07004)。

In [0]:

LAMBDA = 100

In [0]:

**def** discriminator\_loss(disc\_real\_output, disc\_generated\_output):

real\_loss = tf.losses.sigmoid\_cross\_entropy(multi\_class\_labels = tf.ones\_like(disc\_real\_output),

logits = disc\_real\_output)

generated\_loss = tf.losses.sigmoid\_cross\_entropy(multi\_class\_labels = tf.zeros\_like(disc\_generated\_output),

logits = disc\_generated\_output)

total\_disc\_loss = real\_loss + generated\_loss

**return** total\_disc\_loss

In [0]:

**def** generator\_loss(disc\_generated\_output, gen\_output, target):

gan\_loss = tf.losses.sigmoid\_cross\_entropy(multi\_class\_labels = tf.ones\_like(disc\_generated\_output),

logits = disc\_generated\_output)

*# mean absolute error*

l1\_loss = tf.reduce\_mean(tf.abs(target - gen\_output))

total\_gen\_loss = gan\_loss + (LAMBDA \* l1\_loss)

**return** total\_gen\_loss

In [0]:

generator\_optimizer = tf.train.AdamOptimizer(2e-4, beta1=0.5)

discriminator\_optimizer = tf.train.AdamOptimizer(2e-4, beta1=0.5)

## 检查点（基于对象的保存）

In [0]:

checkpoint\_dir = './training\_checkpoints'

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

checkpoint = tf.train.Checkpoint(generator\_optimizer=generator\_optimizer,

discriminator\_optimizer=discriminator\_optimizer,

generator=generator,

discriminator=discriminator)

**Training**

* 我们首先迭代数据集
* 生成器获取输入图像，我们得到生成的输出。
* 鉴别器接收input\_image和生成的图像作为第一输入。第二个输入是input\_image和target\_image。
* 接下来，我们计算发电机和鉴别器损耗。
* 然后，我们计算相对于发生器和鉴别器变量（输入）的损耗梯度，并将它们应用于优化器。

**生成图像**

* 经过培训，它有时间生成一些图像！
* 我们将图像从测试数据集传递到生成器。
* 然后，生成器将输入图像转换为我们期望的输出。
* 最后一步是绘制预测和**瞧！**

In [0]:

EPOCHS = 200

In [0]:

**def** generate\_images(model, test\_input, tar):

*# the training=True is intentional here since*

*# we want the batch statistics while running the model*

*# on the test dataset. If we use training=False, we will get*

*# the accumulated statistics learned from the training dataset*

*# (which we don't want)*

prediction = model(test\_input, training=**True**)

plt.figure(figsize=(15,15))

display\_list = [test\_input[0], tar[0], prediction[0]]

title = ['Input Image', 'Ground Truth', 'Predicted Image']

**for** i **in** range(3):

plt.subplot(1, 3, i+1)

plt.title(title[i])

*# getting the pixel values between [0, 1] to plot it.*

plt.imshow(display\_list[i] \* 0.5 + 0.5)

plt.axis('off')

plt.show()

In [0]:

**def** train(dataset, epochs):

**for** epoch **in** range(epochs):

start = time.time()

**for** input\_image, target **in** dataset:

**with** tf.GradientTape() **as** gen\_tape, tf.GradientTape() **as** disc\_tape:

gen\_output = generator(input\_image, training=**True**)

disc\_real\_output = discriminator(input\_image, target, training=**True**)

disc\_generated\_output = discriminator(input\_image, gen\_output, training=**True**)

gen\_loss = generator\_loss(disc\_generated\_output, gen\_output, target)

disc\_loss = discriminator\_loss(disc\_real\_output, disc\_generated\_output)

generator\_gradients = gen\_tape.gradient(gen\_loss,

generator.variables)

discriminator\_gradients = disc\_tape.gradient(disc\_loss,

discriminator.variables)

generator\_optimizer.apply\_gradients(zip(generator\_gradients,

generator.variables))

discriminator\_optimizer.apply\_gradients(zip(discriminator\_gradients,

discriminator.variables))

**if** epoch % 1 == 0:

clear\_output(wait=**True**)

**for** inp, tar **in** test\_dataset.take(1):

generate\_images(generator, inp, tar)

*# saving (checkpoint) the model every 20 epochs*

**if** (epoch + 1) % 20 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)

print ('Time taken for epoch **{}** is **{}** sec**\n**'.format(epoch + 1,

time.time()-start))

In [0]:

train(train\_dataset, EPOCHS)

## 恢复最新的检查点和测试

In [0]:

*# restoring the latest checkpoint in checkpoint\_dir*

checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

## 测试整个测试数据集

In [0]:

*# Run the trained model on the entire test dataset*

**for** inp, tar **in** test\_dataset:

generate\_images(generator, inp, tar)

# 用tf.keras进行神经风格转移

## Overview

在本教程中，我们将学习如何使用深度学习以另一种图像的风格组合图像（希望你能像毕加索或梵高一样画画？）。这被称为**神经风格转移**！这是[Leon A. Gatys的论文“艺术风格的神经算法”中](https://arxiv.org/abs/1508.06576)概述的一种技术，这是一本很好的读物，你一定要看看它。

但是，什么是神经风格转移？

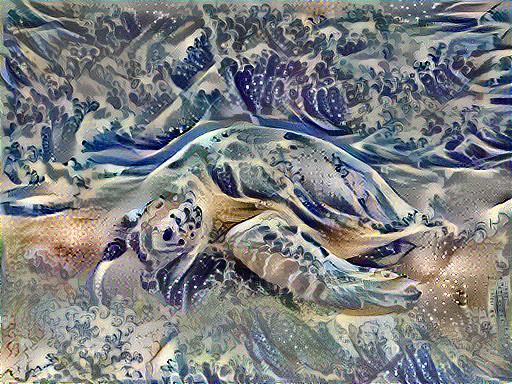
神经风格转移是一种优化技术，用于拍摄三个图像，一个**内容**图像，一个**风格参考**图像（如着名画家的作品），以及您想要设计样式的**输入**图像 - 并将它们混合在一起以便输入图像被转换为​​看起来像内容图像，但在风格图像的样式中“绘制”。

例如，让我们来看看这只乌龟的形象和Katsushika Hokusai的神奈川的大浪：





如果Hokusai决定用这种风格专门描绘这只海龟的照片怎么样呢？像这样的东西？



这是神奇还是只是深度学习？幸运的是，这并不涉及任何巫术：风格转移是一种有趣而有趣的技术，展示了神经网络的能力和内部表征。

神经风格转移的原理是定义两个距离函数，一个描述两个图像的内容有多么不同，一个描述两个图像之间的风格差异，。然后，给定三个图像，期望的样式图像，期望的内容图像和输入图像（用内容图像初始化），我们尝试变换输入图像以最小化与内容图像的内容距离及其与样式距离。风格的形象。总之，我们将采用基本输入图像，我们想要匹配的内容图像以及我们想要匹配的样式图像。我们将通过使用反向传播最小化内容和样式距离（损失）来转换基本输入图像，从而创建与内容图像的内容和样式图像的样式匹配的图像。

### 将涵盖的具体概念：

在此过程中，我们将围绕以下概念构建实践经验并发展直觉

* **急切执行** - 使用TensorFlow的命令式编程环境，立即评估操作
  + [了解有关急切执行的更多信息](https://www.tensorflow.org/programmers_guide/eager)
  + [看到它在行动](https://www.tensorflow.org/get_started/eager)
* **使用**[**Functional API**](https://keras.io/getting-started/functional-api-guide/)**定义模型** - 我们将构建**模型的**一个子集，使我们能够使用Functional API访问必要的中间激活
* **利用预训练模型的特征图** - 了解如何使用预训练模型及其特征图
* **创建自定义训练循环** - 我们将研究如何设置优化器以最小化输入参数的给定损失

### 我们将按照一般步骤进行样式转换：

1. 可视化数据
2. 基本预处理/准备我们的数据
3. 设置损失功能
4. 创建模型
5. 优化损失功能

**观众：**这篇文章面向熟悉基本机器学习概念的中级用户。为了充分利用这篇文章，你应该：

* 阅读[Gatys的论文](https://arxiv.org/abs/1508.06576) - 我们将在此过程中进行解释，但本文将对该任务提供更透彻的理解
* [通过梯度下降了解减少损失](https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

**预计时间**：30分钟

## Setup

### 下载图片[¶](https://render.githubusercontent.com/view/ipynb?commit=f2b702a056ba08a2f2344425f116a673a302abdd&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f74656e736f72666c6f772f6d6f64656c732f663262373032613035366261303861326632333434343235663131366136373361333032616264642f72657365617263682f6e73745f626c6f67706f73742f345f4e657572616c5f5374796c655f5472616e736665725f776974685f45616765725f457865637574696f6e2e6970796e62&nwo=tensorflow%2Fmodels&path=research%2Fnst_blogpost%2F4_Neural_Style_Transfer_with_Eager_Execution.ipynb&repository_id=51117837&repository_type=Repository#Download-Images)

In [0]:

**import** **os**

img\_dir = '/tmp/nst'

**if** **not** os.path.exists(img\_dir):

os.makedirs(img\_dir)

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/d/d7/Green\_Sea\_Turtle\_grazing\_seagrass.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/0/0a/The\_Great\_Wave\_off\_Kanagawa.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/b/b4/Vassily\_Kandinsky%2C\_1913\_-\_Composition\_7.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/0/00/Tuebingen\_Neckarfront.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/6/68/Pillars\_of\_creation\_2014\_HST\_WFC3-UVIS\_full-res\_denoised.jpg

!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/Van\_Gogh\_-\_Starry\_Night\_-\_Google\_Art\_Project.jpg/1024px-Van\_Gogh\_-\_Starry\_Night\_-\_Google\_Art\_Project.jpg

### 导入和配置模块

In [0]:

**import** **matplotlib.pyplot** **as** **plt**

**import** **matplotlib** **as** **mpl**

mpl.rcParams['figure.figsize'] = (10,10)

mpl.rcParams['axes.grid'] = **False**

**import** **numpy** **as** **np**

**from** **PIL** **import** Image

**import** **time**

**import** **functools**

In [0]:

**import** **tensorflow** **as** **tf**

**import** **tensorflow.contrib.eager** **as** **tfe**

**from** **tensorflow.python.keras.preprocessing** **import** image **as** kp\_image

**from** **tensorflow.python.keras** **import** models

**from** **tensorflow.python.keras** **import** losses

**from** **tensorflow.python.keras** **import** layers

**from** **tensorflow.python.keras** **import** backend **as** K

We’ll begin by enabling [eager execution](https://www.tensorflow.org/guide/eager). Eager execution allows us to work through this technique in the clearest and most readable way.

In [0]:

tf.enable\_eager\_execution()

print("Eager execution: **{}**".format(tf.executing\_eagerly()))

In [0]:

*# Set up some global values here*

content\_path = '/tmp/nst/Green\_Sea\_Turtle\_grazing\_seagrass.jpg'

style\_path = '/tmp/nst/The\_Great\_Wave\_off\_Kanagawa.jpg'

## 可视化输入

In [0]:

**def** load\_img(path\_to\_img):

max\_dim = 512

img = Image.open(path\_to\_img)

long = max(img.size)

scale = max\_dim/long

img = img.resize((round(img.size[0]\*scale), round(img.size[1]\*scale)), Image.ANTIALIAS)

img = kp\_image.img\_to\_array(img)

*# We need to broadcast the image array such that it has a batch dimension*

img = np.expand\_dims(img, axis=0)

**return** img

In [0]:

**def** imshow(img, title=**None**):

*# Remove the batch dimension*

out = np.squeeze(img, axis=0)

*# Normalize for display*

out = out.astype('uint8')

plt.imshow(out)

**if** title **is** **not** **None**:

plt.title(title)

plt.imshow(out)

这些是输入内容和样式图像。我们希望用我们的内容图像的内容“创建”一个图像，但具有样式图像的风格。

In [0]:

plt.figure(figsize=(10,10))

content = load\_img(content\_path).astype('uint8')

style = load\_img(style\_path).astype('uint8')

plt.subplot(1, 2, 1)

imshow(content, 'Content Image')

plt.subplot(1, 2, 2)

imshow(style, 'Style Image')

plt.show()

## 准备数据

让我们创建允许我们轻松加载和预处理图像的方法。我们根据VGG培训流程执行与预期相同的预处理过程。VGG网络在图像上进行训练，每个信道由mean = [103.939, 116.779, 123.68]信道BGR进行归一化。  
In [0]:

**def** load\_and\_process\_img(path\_to\_img):

img = load\_img(path\_to\_img)

img = tf.keras.applications.vgg19.preprocess\_input(img)

**return** img

为了查看优化的输出，我们需要执行逆预处理步骤。此外，由于我们优化的图像可之间的任何需要它的值在正负无穷之间，我们必须夹从0-255的范围内维护我们的值。In [0]:

**def** deprocess\_img(processed\_img):

x = processed\_img.copy()

**if** len(x.shape) == 4:

x = np.squeeze(x, 0)

**assert** len(x.shape) == 3, ("Input to deprocess image must be an image of "

"dimension [1, height, width, channel] or [height, width, channel]")

**if** len(x.shape) != 3:

**raise** **ValueError**("Invalid input to deprocessing image")

*# perform the inverse of the preprocessiing step*

x[:, :, 0] += 103.939

x[:, :, 1] += 116.779

x[:, :, 2] += 123.68

x = x[:, :, ::-1]

x = np.clip(x, 0, 255).astype('uint8')

**return** x

### 定义内容和样式表示

为了获得图像的内容和样式表示，我们将查看模型中的一些中间层。随着我们深入到模型中，这些中间层代表越来越高阶的特征。在这种情况下，我们使用网络架构VGG19，一个预训练的图像分类网络。这些中间层是从我们的图像定义内容和样式的表示所必需的。对于输入图像，我们将尝试匹配这些中间层的相应样式和内容目标表示。

#### 为什么中间层？

您可能想知道为什么我们的预训练图像分类网络中的这些中间输出允许我们定义样式和内容表示。在高层次上，这种现象可以通过以下事实来解释：为了使网络执行图像分类（我们的网络已被训练过），它必须理解图像。这包括将原始图像作为输入像素并通过转换构建内部表示，该转换将原始图像像素转换为对图像中存在的特征的复杂理解。这也是卷积神经网络能够很好地推广的部分原因：它们能够捕获不变性并定义类别（例如，猫与狗）之间的特征，这些特征与背景噪声和其他麻烦无关。从而，在输入原始图像和输出分类标签之间的某处，模型用作复杂的特征提取器; 因此，通过访问中间层，我们能够描述输入图像的内容和样式。

具体来说，我们将从我们的网络中提取这些中间层：

In [0]:

*# Content layer where will pull our feature maps*

content\_layers = ['block5\_conv2']

*# Style layer we are interested in*

style\_layers = ['block1\_conv1',

'block2\_conv1',

'block3\_conv1',

'block4\_conv1',

'block5\_conv1'

]

num\_content\_layers = len(content\_layers)

num\_style\_layers = len(style\_layers)

## 构建模型

在这种情况下，我们加载[VGG19](https://keras.io/applications/#vgg19)，并将输入张量输入到模型中。这将允许我们提取内容，样式和生成的图像的特征映射（以及随后的内容和样式表示）。

我们使用VGG19，如文中所建议的那样。此外，由于VGG19是一个相对简单的模型（与ResNet，Inception等相比），因此功能图实际上更适合样式传输。

为了访问与我们的样式和内容特征映射相对应的中间层，我们获得相应的输出并使用Keras [**Functional API**](https://keras.io/getting-started/functional-api-guide/)，我们使用所需的输出激活来定义我们的模型。

使用Functional API定义模型只需要定义输入和输出：

model = Model(inputs, outputs)

In [0]:

**def** get\_model():

*""" Creates our model with access to intermediate layers.*

*This function will load the VGG19 model and access the intermediate layers.*

*These layers will then be used to create a new model that will take input image*

*and return the outputs from these intermediate layers from the VGG model.*

*Returns:*

*returns a keras model that takes image inputs and outputs the style and*

*content intermediate layers.*

*"""*

*# Load our model. We load pretrained VGG, trained on imagenet data*

vgg = tf.keras.applications.vgg19.VGG19(include\_top=**False**, weights='imagenet')

vgg.trainable = **False**

*# Get output layers corresponding to style and content layers*

style\_outputs = [vgg.get\_layer(name).output **for** name **in** style\_layers]

content\_outputs = [vgg.get\_layer(name).output **for** name **in** content\_layers]

model\_outputs = style\_outputs + content\_outputs

*# Build model*

**return** models.Model(vgg.input, model\_outputs)

在上面的代码片段中，我们将加载我们的预训练图像分类网络。然后我们抓住我们之前定义的感兴趣层。然后我们通过将模型的输入设置为图像并将输出设置为样式和内容层的输出来定义模型。换句话说，我们创建了一个模型，它将获取输入图像并输出内容和样式中间层！

## 定义并创建我们的损失函数（内容和样式距离）

### 内容丢失

我们的内容丢失定义实际上非常简单。我们将向网络传递所需的内容图像和基本输入图像。这将从我们的模型返回中间层输出（来自上面定义的层）。然后我们简单地取这些图像的两个中间表示之间的欧氏距离。

更正式地说，内容丢失是一种描述内容与输出图像和内容图像的距离的函数。让我们成为一个预训练的深度卷积神经网络。同样，在这种情况下，我们使用[VGG19](https://keras.io/applications/#vgg19)。设任何图像，然后是由X馈送的网络。让并描述具有输入和层的网络的相应中间特征表示。然后我们正式描述内容距离（损失）：

我们以通常的方式进行反向传播，以便最小化这种内容损失。因此，我们改变初始图像，直到它在某个层（在content\_layer中定义）中生成类似的响应作为原始内容图像。

这可以非常简单地实现。同样，它将输入由x，我们的输入图像和p（我们的内容图像）馈送的网络中的层L作为输入，并返回内容距离。

### 计算内容丢失

我们实际上会在每个所需的层添加内容丢失。这样，当我们通过模型提供输入图像时，每次迭代（简单地说都是model(input\_image)！）通过模型的所有内容损失都将被正确计算，并且因为我们急切地执行所有梯度将被计算。

In [0]:

**def** get\_content\_loss(base\_content, target):

**return** tf.reduce\_mean(tf.square(base\_content - target))

## 风格损失

计算风格损失涉及更多，但遵循相同的原则，这次给我们的网络提供基本输入图像和样式图像。但是，我们不是比较基本输入图像和样式图像的原始中间输出，而是比较两个输出的Gram矩阵。

在数学上，我们描述了基本输入图像的样式损失$ X $，以及样式图像$ A $，作为这些图像的样式表示（语法矩阵）之间的距离。我们将图像的样式表示描述为由Gram矩阵给出的不同滤波器响应之间的相关性 $ G ^升$，其中$ G ^ L\_ {IJ} $是矢量化特征图$ I $和附加$ J $图层之间的内积大号$ $。我们可以看到，$ G ^ L\_ {IJ} $在给定图像的特征映射上生成的代表了特征映射$ I $与特征映射之间的相关性附加$ J $。

为了生成基本输入图像的样式，我们从内容图像执行渐变下降，将其转换为与原始图像的样式表示相匹配的图像。我们通过最小化样式图像的特征相关性图和输入图像之间的均方距离来实现。每个层对总风格损失的贡献描述如下 $$ E\_l = \ frac {1} {4N\_l ^ 2M\_l ^ 2} \ sum\_ {i，j}（G ^ l\_ {ij} - A ^ l\_ {ij}）^ 2 $$

其中$ G ^ L\_ {IJ} $和$ A ^ {L\_ IJ} $是在层相应的式表示大号$ $的$ X $和$ A $。$ N\_l $描述每个尺寸的特征图的数量$ M\_l =高度\*宽度$。因此，每层的总样式损失是 $$ L\_ {style}（a，x）= \ sum\_ {l \ in L} w\_l E\_l $$ 我们通过某种因素来加权每层损失的贡献$ W\_L $。在我们的例子中，我们平等地对每一层加权（$ w\_l = \ frac {1} {| L |} $）

### 计算风格损失

同样，我们将损失作为距离度量来实现。

In [0]:

**def** gram\_matrix(input\_tensor):

*# We make the image channels first*

channels = int(input\_tensor.shape[-1])

a = tf.reshape(input\_tensor, [-1, channels])

n = tf.shape(a)[0]

gram = tf.matmul(a, a, transpose\_a=**True**)

**return** gram / tf.cast(n, tf.float32)

**def** get\_style\_loss(base\_style, gram\_target):

*"""Expects two images of dimension h, w, c"""*

*# height, width, num filters of each layer*

*# We scale the loss at a given layer by the size of the feature map and the number of filters*

height, width, channels = base\_style.get\_shape().as\_list()

gram\_style = gram\_matrix(base\_style)

**return** tf.reduce\_mean(tf.square(gram\_style - gram\_target))*# / (4. \* (channels \*\* 2) \* (width \* height) \*\* 2)*

## 将样式转移应用于我们的图像

### 运行梯度下降

如果你不熟悉梯度下降/反向传播或需要复习，你一定要看看这个[很棒的资源](https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)。

在这种情况下，我们使用[Adam](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam) \*优化器来减少损失。我们迭代地更新输出图像，以便最大限度地减少损失：我们不更新与网络相关的权重，而是训练输入图像以最小化损失。为了做到这一点，我们必须知道我们如何计算损失和梯度。

\*请注意，如果您熟悉此算法，建议不要使用L-BFGS，因为本教程背后的主要动机是用热切的执行来说明最佳实践，并且通过使用Adam，我们可以演示autograd / gradient磁带功能，带有自定义训练循环。

我们将定义一个小辅助函数，它将加载我们的内容和样式图像，通过我们的网络向前馈送它们，然后从我们的模型输出内容和样式特征表示。

In [0]:

**def** get\_feature\_representations(model, content\_path, style\_path):

*"""Helper function to compute our content and style feature representations.*

*This function will simply load and preprocess both the content and style*

*images from their path. Then it will feed them through the network to obtain*

*the outputs of the intermediate layers.*

*Arguments:*

*model: The model that we are using.*

*content\_path: The path to the content image.*

*style\_path: The path to the style image*

*Returns:*

*returns the style features and the content features.*

*"""*

*# Load our images in*

content\_image = load\_and\_process\_img(content\_path)

style\_image = load\_and\_process\_img(style\_path)

*# batch compute content and style features*

style\_outputs = model(style\_image)

content\_outputs = model(content\_image)

*# Get the style and content feature representations from our model*

style\_features = [style\_layer[0] **for** style\_layer **in** style\_outputs[:num\_style\_layers]]

content\_features = [content\_layer[0] **for** content\_layer **in** content\_outputs[num\_style\_layers:]]

**return** style\_features, content\_features

### 计算损失和梯度

这里我们使用[**tf.GradientTape**](https://www.tensorflow.org/programmers_guide/eager#computing_gradients)来计算梯度。它允许我们通过跟踪操作来利用可用的自动区分，以便稍后计算梯度。它记录前向传递期间的操作，然后能够计算我们的损失函数相对于向后传递的输入图像的梯度。

In [0]:

**def** compute\_loss(model, loss\_weights, init\_image, gram\_style\_features, content\_features):

*"""This function will compute the loss total loss.*

*Arguments:*

*model: The model that will give us access to the intermediate layers*

*loss\_weights: The weights of each contribution of each loss function.*

*(style weight, content weight, and total variation weight)*

*init\_image: Our initial base image. This image is what we are updating with*

*our optimization process. We apply the gradients wrt the loss we are*

*calculating to this image.*

*gram\_style\_features: Precomputed gram matrices corresponding to the*

*defined style layers of interest.*

*content\_features: Precomputed outputs from defined content layers of*

*interest.*

*Returns:*

*returns the total loss, style loss, content loss, and total variational loss*

*"""*

style\_weight, content\_weight = loss\_weights

*# Feed our init image through our model. This will give us the content and*

*# style representations at our desired layers. Since we're using eager*

*# our model is callable just like any other function!*

model\_outputs = model(init\_image)

style\_output\_features = model\_outputs[:num\_style\_layers]

content\_output\_features = model\_outputs[num\_style\_layers:]

style\_score = 0

content\_score = 0

*# Accumulate style losses from all layers*

*# Here, we equally weight each contribution of each loss layer*

weight\_per\_style\_layer = 1.0 / float(num\_style\_layers)

**for** target\_style, comb\_style **in** zip(gram\_style\_features, style\_output\_features):

style\_score += weight\_per\_style\_layer \* get\_style\_loss(comb\_style[0], target\_style)

*# Accumulate content losses from all layers*

weight\_per\_content\_layer = 1.0 / float(num\_content\_layers)

**for** target\_content, comb\_content **in** zip(content\_features, content\_output\_features):

content\_score += weight\_per\_content\_layer\* get\_content\_loss(comb\_content[0], target\_content)

style\_score \*= style\_weight

content\_score \*= content\_weight

*# Get total loss*

loss = style\_score + content\_score

**return** loss, style\_score, content\_score

然后计算渐变很容易：

In [0]:

**def** compute\_grads(cfg):

**with** tf.GradientTape() **as** tape:

all\_loss = compute\_loss(\*\*cfg)

*# Compute gradients wrt input image*

total\_loss = all\_loss[0]

**return** tape.gradient(total\_loss, cfg['init\_image']), all\_loss

### 优化循环

In [0]:

**import** **IPython.display**

**def** run\_style\_transfer(content\_path,

style\_path,

num\_iterations=1000,

content\_weight=1e3,

style\_weight=1e-2):

*# We don't need to (or want to) train any layers of our model, so we set their*

*# trainable to false.*

model = get\_model()

**for** layer **in** model.layers:

layer.trainable = **False**

*# Get the style and content feature representations (from our specified intermediate layers)*

style\_features, content\_features = get\_feature\_representations(model, content\_path, style\_path)

gram\_style\_features = [gram\_matrix(style\_feature) **for** style\_feature **in** style\_features]

*# Set initial image*

init\_image = load\_and\_process\_img(content\_path)

init\_image = tfe.Variable(init\_image, dtype=tf.float32)

*# Create our optimizer*

opt = tf.train.AdamOptimizer(learning\_rate=5, beta1=0.99, epsilon=1e-1)

*# For displaying intermediate images*

iter\_count = 1

*# Store our best result*

best\_loss, best\_img = float('inf'), **None**

*# Create a nice config*

loss\_weights = (style\_weight, content\_weight)

cfg = {

'model': model,

'loss\_weights': loss\_weights,

'init\_image': init\_image,

'gram\_style\_features': gram\_style\_features,

'content\_features': content\_features

}

*# For displaying*

num\_rows = 2

num\_cols = 5

display\_interval = num\_iterations/(num\_rows\*num\_cols)

start\_time = time.time()

global\_start = time.time()

norm\_means = np.array([103.939, 116.779, 123.68])

min\_vals = -norm\_means

max\_vals = 255 - norm\_means

imgs = []

**for** i **in** range(num\_iterations):

grads, all\_loss = compute\_grads(cfg)

loss, style\_score, content\_score = all\_loss

opt.apply\_gradients([(grads, init\_image)])

clipped = tf.clip\_by\_value(init\_image, min\_vals, max\_vals)

init\_image.assign(clipped)

end\_time = time.time()

**if** loss < best\_loss:

*# Update best loss and best image from total loss.*

best\_loss = loss

best\_img = deprocess\_img(init\_image.numpy())

**if** i % display\_interval== 0:

start\_time = time.time()

*# Use the .numpy() method to get the concrete numpy array*

plot\_img = init\_image.numpy()

plot\_img = deprocess\_img(plot\_img)

imgs.append(plot\_img)

IPython.display.clear\_output(wait=**True**)

IPython.display.display\_png(Image.fromarray(plot\_img))

print('Iteration: **{}**'.format(i))

print('Total loss: **{:.4e}**, '

'style loss: **{:.4e}**, '

'content loss: **{:.4e}**, '

'time: **{:.4f}**s'.format(loss, style\_score, content\_score, time.time() - start\_time))

print('Total time: **{:.4f}**s'.format(time.time() - global\_start))

IPython.display.clear\_output(wait=**True**)

plt.figure(figsize=(14,4))

**for** i,img **in** enumerate(imgs):

plt.subplot(num\_rows,num\_cols,i+1)

plt.imshow(img)

plt.xticks([])

plt.yticks([])

**return** best\_img, best\_loss

In [0]:

best, best\_loss = run\_style\_transfer(content\_path,

style\_path, num\_iterations=1000)

In [0]:

Image.fromarray(best)

要从Colab下载图像，请取消注释以下代码：

In [0]:

*#from google.colab import files*

*#files.download('wave\_turtle.png')*

## 可视化输出

我们“处理”输出图像以删除应用于它的处理。

In [0]:

**def** show\_results(best\_img, content\_path, style\_path, show\_large\_final=**True**):

plt.figure(figsize=(10, 5))

content = load\_img(content\_path)

style = load\_img(style\_path)

plt.subplot(1, 2, 1)

imshow(content, 'Content Image')

plt.subplot(1, 2, 2)

imshow(style, 'Style Image')

**if** show\_large\_final:

plt.figure(figsize=(10, 10))

plt.imshow(best\_img)

plt.title('Output Image')

plt.show()

In [0]:

show\_results(best, content\_path, style\_path)

## 在其他图像上试试

### Starry night + Tuebingen

In [0]:

best\_starry\_night, best\_loss = run\_style\_transfer('/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/1024px-Van\_Gogh\_-\_Starry\_Night\_-\_Google\_Art\_Project.jpg')

In [0]:

show\_results(best\_starry\_night, '/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/1024px-Van\_Gogh\_-\_Starry\_Night\_-\_Google\_Art\_Project.jpg')

### Pillars of Creation + Tuebingen

In [0]:

best\_poc\_tubingen, best\_loss = run\_style\_transfer('/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/Pillars\_of\_creation\_2014\_HST\_WFC3-UVIS\_full-res\_denoised.jpg')

In [0]:

show\_results(best\_poc\_tubingen,

'/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/Pillars\_of\_creation\_2014\_HST\_WFC3-UVIS\_full-res\_denoised.jpg')

### Kandinsky Composition 7 + Tuebingen

In [0]:

best\_kandinsky\_tubingen, best\_loss = run\_style\_transfer('/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/Vassily\_Kandinsky,\_1913\_-\_Composition\_7.jpg')

In [0]:

show\_results(best\_kandinsky\_tubingen,

'/tmp/nst/Tuebingen\_Neckarfront.jpg',

'/tmp/nst/Vassily\_Kandinsky,\_1913\_-\_Composition\_7.jpg')

### Pillars of Creation + Sea Turtle

In [0]:

best\_poc\_turtle, best\_loss = run\_style\_transfer('/tmp/nst/Green\_Sea\_Turtle\_grazing\_seagrass.jpg',

'/tmp/nst/Pillars\_of\_creation\_2014\_HST\_WFC3-UVIS\_full-res\_denoised.jpg')

In [0]:

show\_results(best\_poc\_turtle,

'/tmp/nst/Green\_Sea\_Turtle\_grazing\_seagrass.jpg',

'/tmp/nst/Pillars\_of\_creation\_2014\_HST\_WFC3-UVIS\_full-res\_denoised.jpg')

**关键要点**

**我们涵盖的内容：**

* 我们构建了几种不同的损耗函数，并使用反向传播来转换输入图像，以最大限度地减少这些损失
  + 为了做到这一点，我们必须加载一个**预训练模型，**并使用其学习的特征图来描述我们图像的内容和样式表示。
    - 我们的主要损失函数主要是根据这些不同的表示来计算距离
* 我们使用自定义模型和**热切执行**实现了这一点
  + 我们使用Functional API构建了自定义模型
  + 热切的执行允许我们使用自然的python控制流动态地使用张量
  + 我们直接操纵张量，这使得调试和使用张量更容易。
* 我们通过使用**tf.gradient**应用我们的优化器更新规则来迭代更新我们的图像。优化器最小化了输入图像的给定损失。

# 图像分割使用tf.keras

在本教程中，我们将学习如何分割图像。**分割**是生成逐像素**分割**的过程，给出在每个像素处可见的对象类。例如，我们可以识别图像中人的位置和边界，或者从图像中识别细胞核。形式上，图像分割指的是将图像划分为我们想要识别（我们的目标）和背景的一组像素的过程。

具体来说，在本教程中，我们将使用[Kaggle Carvana Image Masking Challenge Dataset](https://www.kaggle.com/c/carvana-image-masking-challenge)。

该数据集包含大量汽车图像，每辆汽车从不同角度拍摄。此外，对于每个汽车图像，我们都有一个相关的手动切割面罩; 我们的任务是为看不见的数据自动创建这些剪切蒙版。

## 将涵盖的具体概念：

在此过程中，我们将围绕以下概念构建实践经验并发展直觉：

* [**功能API**](https://keras.io/getting-started/functional-api-guide/) - 我们将实现UNet，一种经典用于使用Functional API进行生物医学图像分割的卷积网络模型。
  + 该模型具有需要多个输入/输出的层。这需要使用功能API
  + 查看原始[论文](https://arxiv.org/abs/1505.04597)，U-Net：Olaf Ronneberger的生物医学图像分割卷积网络！
* **自定义损失函数和指标** - 我们将使用二进制[**交叉熵**](https://developers.google.com/machine-learning/glossary/#cross-entropy)和**骰子损失**实现自定义损失函数。我们还将实施**骰子系数**（用于我们的损失）和**平均交叉结合**，这将有助于我们监控我们的培训过程并判断我们的表现如何。
* **保存和加载keras模型** - 我们将保存最佳模型到磁盘。当我们想要进行推理/评估我们的模型时，我们将从磁盘加载模型。

### 我们将遵循一般工作流程：

1. 可视化数据/执行一些探索性数据分析
2. 设置数据管道和预处理
3. 建立模型
4. 火车模型
5. 评估模型
6. 重复

**观众：**这篇文章面向熟悉基本机器学习概念的中级用户。请注意，如果您希望运行此笔记本，强烈建议您使用GPU。

**预计时间**：60分钟

作者：Raymond Yuan，软件工程实习生

In [0]:

!pip install kaggle

In [0]:

**import** **os**

**import** **glob**

**import** **zipfile**

**import** **functools**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**import** **matplotlib** **as** **mpl**

mpl.rcParams['axes.grid'] = **False**

mpl.rcParams['figure.figsize'] = (12,12)

**from** **sklearn.model\_selection** **import** train\_test\_split

**import** **matplotlib.image** **as** **mpimg**

**import** **pandas** **as** **pd**

**from** **PIL** **import** Image

In [0]:

**import** **tensorflow** **as** **tf**

**import** **tensorflow.contrib** **as** **tfcontrib**

**from** **tensorflow.python.keras** **import** layers

**from** **tensorflow.python.keras** **import** losses

**from** **tensorflow.python.keras** **import** models

**from** **tensorflow.python.keras** **import** backend **as** K

## 获取所有文件

由于本教程将使用Kaggle的数据集，因此需要为您的Kaggle帐户[创建API令牌](https://github.com/Kaggle/kaggle-api#api-credentials)并上传它。

In [0]:

**import** **os**

*# Upload the API token.*

**def** get\_kaggle\_credentials():

token\_dir = os.path.join(os.path.expanduser("~"),".kaggle")

token\_file = os.path.join(token\_dir, "kaggle.json")

**if** **not** os.path.isdir(token\_dir):

os.mkdir(token\_dir)

**try**:

**with** open(token\_file,'r') **as** f:

**pass**

**except** **IOError** **as** no\_file:

**try**:

**from** **google.colab** **import** files

**except** **ImportError**:

**raise** no\_file

uploaded = files.upload()

**if** "kaggle.json" **not** **in** uploaded:

**raise** **ValueError**("You need an API key! see: "

"https://github.com/Kaggle/kaggle-api#api-credentials")

**with** open(token\_file, "wb") **as** f:

f.write(uploaded["kaggle.json"])

os.chmod(token\_file, 600)

get\_kaggle\_credentials()

仅在添加凭据后导入kaggle。

In [0]:

**import** **kaggle**

### 我们将从Kaggle下载数据

注意，提前大量下载 - 下载所有文件将需要14GB的磁盘空间。

In [0]:

competition\_name = 'carvana-image-masking-challenge'

In [0]:

*# Download data from Kaggle and unzip the files of interest.*

**def** load\_data\_from\_zip(competition, file):

**with** zipfile.ZipFile(os.path.join(competition, file), "r") **as** zip\_ref:

unzipped\_file = zip\_ref.namelist()[0]

zip\_ref.extractall(competition)

**def** get\_data(competition):

kaggle.api.competition\_download\_files(competition, competition)

load\_data\_from\_zip(competition, 'train.zip')

load\_data\_from\_zip(competition, 'train\_masks.zip')

load\_data\_from\_zip(competition, 'train\_masks.csv.zip')

您必须在下载数据之前[接受竞争规则](https://www.kaggle.com/c/carvana-image-masking-challenge/rules)。

In [0]:

get\_data(competition\_name)

In [0]:

img\_dir = os.path.join(competition\_name, "train")

label\_dir = os.path.join(competition\_name, "train\_masks")

In [0]:

df\_train = pd.read\_csv(os.path.join(competition\_name, 'train\_masks.csv'))

ids\_train = df\_train['img'].map(**lambda** s: s.split('.')[0])

In [0]:

x\_train\_filenames = []

y\_train\_filenames = []

**for** img\_id **in** ids\_train:

x\_train\_filenames.append(os.path.join(img\_dir, "**{}**.jpg".format(img\_id)))

y\_train\_filenames.append(os.path.join(label\_dir, "**{}**\_mask.gif".format(img\_id)))

In [0]:

x\_train\_filenames, x\_val\_filenames, y\_train\_filenames, y\_val\_filenames = \

train\_test\_split(x\_train\_filenames, y\_train\_filenames, test\_size=0.2, random\_state=42)

In [0]:

num\_train\_examples = len(x\_train\_filenames)

num\_val\_examples = len(x\_val\_filenames)

print("Number of training examples: **{}**".format(num\_train\_examples))

print("Number of validation examples: **{}**".format(num\_val\_examples))

### 这是路径的样子

In [0]:

x\_train\_filenames[:10]

In [0]:

y\_train\_filenames[:10]

## 可视化

我们来看看数据集中不同图像的一些示例。

In [0]:

display\_num = 5

r\_choices = np.random.choice(num\_train\_examples, display\_num)

plt.figure(figsize=(10, 15))

**for** i **in** range(0, display\_num \* 2, 2):

img\_num = r\_choices[i // 2]

x\_pathname = x\_train\_filenames[img\_num]

y\_pathname = y\_train\_filenames[img\_num]

plt.subplot(display\_num, 2, i + 1)

plt.imshow(mpimg.imread(x\_pathname))

plt.title("Original Image")

example\_labels = Image.open(y\_pathname)

label\_vals = np.unique(example\_labels)

plt.subplot(display\_num, 2, i + 2)

plt.imshow(example\_labels)

plt.title("Masked Image")

plt.suptitle("Examples of Images and their Masks")

plt.show()

## 设置

让我们从设置一些参数开始。我们将标准化并调整图像的所有形状。我们还将设置一些训练参数：

In [0]:

img\_shape = (256, 256, 3)

batch\_size = 3

epochs = 5

使用这些完全相同的参数可能对您的硬件来说计算量太大，因此请相应地调整参数。此外，重要的是要注意，由于我们的UNet版本的架构，图像的大小必须可以被32整除，因为我们下调每个空间分辨率2倍MaxPooling2Dlayer。

如果您的机器可以支持它，您将使用更高分辨率的输入图像（例如512 x 512）获得更好的性能，因为这样可以在编码过程中实现更精确的定位和更少的信息丢失。此外，您还可以使模型更深入。

或者，如果您的机器不支持它，请降低图像分辨率和/或批量大小。请注意，降低图像分辨率会降低性能，降低批量会增加培训时间。

## 使用构建输入管道tf.data

由于我们从文件名开始，我们需要构建一个健壮且可扩展的数据管道，它将与我们的模型很好地配合。如果你不熟悉**tf.data，**你应该看看我介绍这个概念的其他教程！

### 我们的输入管道将包含以下步骤：

1. 从文件名中读取文件的字节 - 对于图像和标签。回想一下，我们的标签实际上是每个像素注释为汽车或背景的图像（1,0）。
2. 将字节解码为图像格式
3. 应用图像转换:(可选，根据输入参数）
   * resize - 将图像大小调整为标准大小（由eda或计算/内存限制确定）
     + 这是可选的原因是U-Net是完全卷积网络（例如没有完全连接的单元），因此不依赖于输入大小。但是，如果您选择不调整图像大小，则必须使用批量大小为1，因为您无法将可变图像大小一起批量处理
     + 或者，您也可以将图像拼凑在一起，并按照小批量调整大小，以避免调整图像大小，因为调整大小可能会通过插值等影响您的性能。
   * hue\_delta - 通过随机因子调整RGB图像的色调。这仅适用于实际图像（不是我们的标签图像）。在hue\_delta必须在间隔[0, 0.5]
   * horizontal\_flip - 以0.5概率沿中心轴水平翻转图像。此转换必须应用于标签和实际图像。
   * width\_shift\_range并且height\_shift\_range是在水平或垂直方向上随机平移图像的范围（作为总宽度或高度的一部分）。此转换必须应用于标签和实际图像。
   * rescale - 按特定因子重新缩放图像，例如1/255。
4. 对数据进行混洗，重复数据（因此我们可以跨时期迭代多次），批量处理数据，然后预取批次（为了提高效率）。

请务必注意，数据管道中发生的这些转换必须是符号转换。

#### 为什么我们要进行这些图像转换？

这称为**数据增强**。数据增加通过大量随机变换来增加训练数据的数量。在训练期间，我们的模型永远不会看到完全相同图片的两倍。这有助于防止[过度拟合，](https://developers.google.com/machine-learning/glossary/#overfitting)并有助于模型更好地概括为看不见的数据。

## 处理每个路径名[¶](https://render.githubusercontent.com/view/ipynb?commit=f2b702a056ba08a2f2344425f116a673a302abdd&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f74656e736f72666c6f772f6d6f64656c732f663262373032613035366261303861326632333434343235663131366136373361333032616264642f73616d706c65732f6f757472656163682f626c6f67732f7365676d656e746174696f6e5f626c6f67706f73742f696d6167655f7365676d656e746174696f6e2e6970796e62&nwo=tensorflow%2Fmodels&path=samples%2Foutreach%2Fblogs%2Fsegmentation_blogpost%2Fimage_segmentation.ipynb&repository_id=51117837&repository_type=Repository#Processing-each-pathname)

In [0]:

**def** \_process\_pathnames(fname, label\_path):

*# We map this function onto each pathname pair*

img\_str = tf.read\_file(fname)

img = tf.image.decode\_jpeg(img\_str, channels=3)

label\_img\_str = tf.read\_file(label\_path)

*# These are gif images so they return as (num\_frames, h, w, c)*

label\_img = tf.image.decode\_gif(label\_img\_str)[0]

*# The label image should only have values of 1 or 0, indicating pixel wise*

*# object (car) or not (background). We take the first channel only.*

label\_img = label\_img[:, :, 0]

label\_img = tf.expand\_dims(label\_img, axis=-1)

**return** img, label\_img

## 移动图像

In [0]:

**def** shift\_img(output\_img, label\_img, width\_shift\_range, height\_shift\_range):

*"""This fn will perform the horizontal or vertical shift"""*

**if** width\_shift\_range **or** height\_shift\_range:

**if** width\_shift\_range:

width\_shift\_range = tf.random\_uniform([],

-width\_shift\_range \* img\_shape[1],

width\_shift\_range \* img\_shape[1])

**if** height\_shift\_range:

height\_shift\_range = tf.random\_uniform([],

-height\_shift\_range \* img\_shape[0],

height\_shift\_range \* img\_shape[0])

*# Translate both*

output\_img = tfcontrib.image.translate(output\_img,

[width\_shift\_range, height\_shift\_range])

label\_img = tfcontrib.image.translate(label\_img,

[width\_shift\_range, height\_shift\_range])

**return** output\_img, label\_img

## 随机翻转图像

In [0]:

**def** flip\_img(horizontal\_flip, tr\_img, label\_img):

**if** horizontal\_flip:

flip\_prob = tf.random\_uniform([], 0.0, 1.0)

tr\_img, label\_img = tf.cond(tf.less(flip\_prob, 0.5),

**lambda**: (tf.image.flip\_left\_right(tr\_img), tf.image.flip\_left\_right(label\_img)),

**lambda**: (tr\_img, label\_img))

**return** tr\_img, label\_img

## 将我们的转换组装到我们的扩充函数中

In [0]:

**def** \_augment(img,

label\_img,

resize=**None**, *# Resize the image to some size e.g. [256, 256]*

scale=1, *# Scale image e.g. 1 / 255.*

hue\_delta=0, *# Adjust the hue of an RGB image by random factor*

horizontal\_flip=**False**, *# Random left right flip,*

width\_shift\_range=0, *# Randomly translate the image horizontally*

height\_shift\_range=0): *# Randomly translate the image vertically*

**if** resize **is** **not** **None**:

*# Resize both images*

label\_img = tf.image.resize\_images(label\_img, resize)

img = tf.image.resize\_images(img, resize)

**if** hue\_delta:

img = tf.image.random\_hue(img, hue\_delta)

img, label\_img = flip\_img(horizontal\_flip, img, label\_img)

img, label\_img = shift\_img(img, label\_img, width\_shift\_range, height\_shift\_range)

label\_img = tf.to\_float(label\_img) \* scale

img = tf.to\_float(img) \* scale

**return** img, label\_img

In [0]:

**def** get\_baseline\_dataset(filenames,

labels,

preproc\_fn=functools.partial(\_augment),

threads=5,

batch\_size=batch\_size,

shuffle=**True**):

num\_x = len(filenames)

*# Create a dataset from the filenames and labels*

dataset = tf.data.Dataset.from\_tensor\_slices((filenames, labels))

*# Map our preprocessing function to every element in our dataset, taking*

*# advantage of multithreading*

dataset = dataset.map(\_process\_pathnames, num\_parallel\_calls=threads)

**if** preproc\_fn.keywords **is** **not** **None** **and** 'resize' **not** **in** preproc\_fn.keywords:

**assert** batch\_size == 1, "Batching images must be of the same size"

dataset = dataset.map(preproc\_fn, num\_parallel\_calls=threads)

**if** shuffle:

dataset = dataset.shuffle(num\_x)

*# It's necessary to repeat our data for all epochs*

dataset = dataset.repeat().batch(batch\_size)

**return** dataset

## 设置列车和验证数据集

请注意，我们将图像增强应用于训练数据集，但不应用于我们的验证数据集。

Note that we apply image augmentation to our training dataset but not our validation dataset.

In [0]:

tr\_cfg = {

'resize': [img\_shape[0], img\_shape[1]],

'scale': 1 / 255.,

'hue\_delta': 0.1,

'horizontal\_flip': **True**,

'width\_shift\_range': 0.1,

'height\_shift\_range': 0.1

}

tr\_preprocessing\_fn = functools.partial(\_augment, \*\*tr\_cfg)

In [0]:

val\_cfg = {

'resize': [img\_shape[0], img\_shape[1]],

'scale': 1 / 255.,

}

val\_preprocessing\_fn = functools.partial(\_augment, \*\*val\_cfg)

In [0]:

train\_ds = get\_baseline\_dataset(x\_train\_filenames,

y\_train\_filenames,

preproc\_fn=tr\_preprocessing\_fn,

batch\_size=batch\_size)

val\_ds = get\_baseline\_dataset(x\_val\_filenames,

y\_val\_filenames,

preproc\_fn=val\_preprocessing\_fn,

batch\_size=batch\_size)

## 让我们看看我们的图像增强器数据管道是否产生了预期的结果

In [0]:

temp\_ds = get\_baseline\_dataset(x\_train\_filenames,

y\_train\_filenames,

preproc\_fn=tr\_preprocessing\_fn,

batch\_size=1,

shuffle=**False**)

*# Let's examine some of these augmented images*

data\_aug\_iter = temp\_ds.make\_one\_shot\_iterator()

next\_element = data\_aug\_iter.get\_next()

**with** tf.Session() **as** sess:

batch\_of\_imgs, label = sess.run(next\_element)

*# Running next element in our graph will produce a batch of images*

plt.figure(figsize=(10, 10))

img = batch\_of\_imgs[0]

plt.subplot(1, 2, 1)

plt.imshow(img)

plt.subplot(1, 2, 2)

plt.imshow(label[0, :, :, 0])

plt.show()

## 构建模型

我们将构建U-Net模型。U-Net特别适用于分段任务，因为它可以很好地定位以提供高分辨率分段掩码。此外，它适用于小型数据集，并且对于过度拟合相对稳健，因为训练数据是根据图像内的补丁数量，这远远大于训练图像本身的数量。与原始模型不同，我们将为每个块添加批量标准化。

Unet由编码器部分和解码器部分构成。编码器部分由一个线性堆叠的[Conv](https://developers.google.com/machine-learning/glossary/#convolution)，BatchNorm以及[Relu](https://developers.google.com/machine-learning/glossary/#ReLU)随后是操作[MaxPool](https://developers.google.com/machine-learning/glossary/#pooling)。每个都MaxPool将我们的特征图的空间分辨率降低2倍。当我们用解码器部分提供这些高分辨率特征图时，我们跟踪每个块的输出。解码器部分由UpSampling2D，Conv，BatchNorm和Relus组成。请注意，我们在解码器端连接相同大小的特征映射。最后，我们添加了一个最终的Conv操作，该操作沿着通道为每个单独的像素（内核大小为（1,1））执行卷积，以灰度输出我们的最终分割掩码。

## 硬功能

当您拥有多输入/输出模型，共享层等时，将使用Keras功能API。它是一个功能强大的API，允许您轻松操纵张量并使用交织在一起的数据流构建复杂图形。此外，它使得**图层**和**模型**都可以在张量上调用。

* 要查看更多示例，请查看[入门指南](https://keras.io/getting-started/functional-api-guide/)。

我们将构建这些辅助函数，使我们能够轻松简单地集成模块操作。

In [0]:

**def** conv\_block(input\_tensor, num\_filters):

encoder = layers.Conv2D(num\_filters, (3, 3), padding='same')(input\_tensor)

encoder = layers.BatchNormalization()(encoder)

encoder = layers.Activation('relu')(encoder)

encoder = layers.Conv2D(num\_filters, (3, 3), padding='same')(encoder)

encoder = layers.BatchNormalization()(encoder)

encoder = layers.Activation('relu')(encoder)

**return** encoder

**def** encoder\_block(input\_tensor, num\_filters):

encoder = conv\_block(input\_tensor, num\_filters)

encoder\_pool = layers.MaxPooling2D((2, 2), strides=(2, 2))(encoder)

**return** encoder\_pool, encoder

**def** decoder\_block(input\_tensor, concat\_tensor, num\_filters):

decoder = layers.Conv2DTranspose(num\_filters, (2, 2), strides=(2, 2), padding='same')(input\_tensor)

decoder = layers.concatenate([concat\_tensor, decoder], axis=-1)

decoder = layers.BatchNormalization()(decoder)

decoder = layers.Activation('relu')(decoder)

decoder = layers.Conv2D(num\_filters, (3, 3), padding='same')(decoder)

decoder = layers.BatchNormalization()(decoder)

decoder = layers.Activation('relu')(decoder)

decoder = layers.Conv2D(num\_filters, (3, 3), padding='same')(decoder)

decoder = layers.BatchNormalization()(decoder)

decoder = layers.Activation('relu')(decoder)

**return** decoder

In [0]:

inputs = layers.Input(shape=img\_shape)

*# 256*

encoder0\_pool, encoder0 = encoder\_block(inputs, 32)

*# 128*

encoder1\_pool, encoder1 = encoder\_block(encoder0\_pool, 64)

*# 64*

encoder2\_pool, encoder2 = encoder\_block(encoder1\_pool, 128)

*# 32*

encoder3\_pool, encoder3 = encoder\_block(encoder2\_pool, 256)

*# 16*

encoder4\_pool, encoder4 = encoder\_block(encoder3\_pool, 512)

*# 8*

center = conv\_block(encoder4\_pool, 1024)

*# center*

decoder4 = decoder\_block(center, encoder4, 512)

*# 16*

decoder3 = decoder\_block(decoder4, encoder3, 256)

*# 32*

decoder2 = decoder\_block(decoder3, encoder2, 128)

*# 64*

decoder1 = decoder\_block(decoder2, encoder1, 64)

*# 128*

decoder0 = decoder\_block(decoder1, encoder0, 32)

*# 256*

outputs = layers.Conv2D(1, (1, 1), activation='sigmoid')(decoder0)

## 定义你的模型

使用功能API，您必须通过指定与模型关联的输入和输出来定义模型。

In [0]:

model = models.Model(inputs=[inputs], outputs=[outputs])

## 定义自定义指标和损失函数

使用Keras定义损失和度量函数很简单。只需定义一个函数，该函数既可以获取给定示例的True标签，也可以获取同一给定示例的Predicted标签。

骰子丢失是衡量重叠的指标。有关优化骰子系数（我们的骰子损失）的更多信息可以在引入它的[论文中](http://campar.in.tum.de/pub/milletari2016Vnet/milletari2016Vnet.pdf)找到。

我们在这里使用骰子丢失，因为它在设计上在类不平衡问题上表现更好。此外，最大化骰子系数和IoU指标是我们分割任务的实际目标和目标。使用交叉熵更像是一种更容易最大化的代理。相反，我们直接最大化我们的目标。

In [0]:

**def** dice\_coeff(y\_true, y\_pred):

smooth = 1.

*# Flatten*

y\_true\_f = tf.reshape(y\_true, [-1])

y\_pred\_f = tf.reshape(y\_pred, [-1])

intersection = tf.reduce\_sum(y\_true\_f \* y\_pred\_f)

score = (2. \* intersection + smooth) / (tf.reduce\_sum(y\_true\_f) + tf.reduce\_sum(y\_pred\_f) + smooth)

**return** score

In [0]:

**def** dice\_loss(y\_true, y\_pred):

loss = 1 - dice\_coeff(y\_true, y\_pred)

**return** loss

在这里，我们将使用一个专门的损失函数，它结合了二进制交叉熵和我们的骰子损失。这是基于[在本次比赛中参赛的个人凭经验获得更好的成绩](https://www.kaggle.com/c/carvana-image-masking-challenge/discussion/40199)。尝试自己的自定义损失来衡量性能（例如bce + log（dice\_loss），只有bce等）！

In [0]:

**def** bce\_dice\_loss(y\_true, y\_pred):

loss = losses.binary\_crossentropy(y\_true, y\_pred) + dice\_loss(y\_true, y\_pred)

**return** loss

## 编译你的模型

我们使用自定义损失函数来最小化。此外，我们还会指定在培训时要跟踪的指标。请注意，在培训过程中实际上并未使用度量来调整参数，而是用于衡量培训过程的性能。

In [0]:

model.compile(optimizer='adam', loss=bce\_dice\_loss, metrics=[dice\_loss])

model.summary()

## 训练你的模型

训练模型tf.data涉及简单地将模型的fit功能与训练/验证数据集，步数和时期一起提供。

我们还包括一个模型回调，[ModelCheckpoint](https://keras.io/callbacks/#modelcheckpoint)它将在每个纪元后将模型保存到磁盘。我们对其进行配置，使其仅保存我们性能最高的模型。请注意，保存模型捕获的不仅仅是模型的权重：默认情况下，它会保存模型体系结构，权重以及有关培训过程的信息，例如优化程序的状态等。

In [0]:

save\_model\_path = '/tmp/weights.hdf5'

cp = tf.keras.callbacks.ModelCheckpoint(filepath=save\_model\_path, monitor='val\_dice\_loss', save\_best\_only=**True**, verbose=1)

不要忘记在fit函数调用中指定我们的模型回调。

In [0]:

history = model.fit(train\_ds,

steps\_per\_epoch=int(np.ceil(num\_train\_examples / float(batch\_size))),

epochs=epochs,

validation\_data=val\_ds,

validation\_steps=int(np.ceil(num\_val\_examples / float(batch\_size))),

callbacks=[cp])

## 可视化培训过程

In [0]:

dice = history.history['dice\_loss']

val\_dice = history.history['val\_dice\_loss']

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs\_range = range(epochs)

plt.figure(figsize=(16, 8))

plt.subplot(1, 2, 1)

plt.plot(epochs\_range, dice, label='Training Dice Loss')

plt.plot(epochs\_range, val\_dice, label='Validation Dice Loss')

plt.legend(loc='upper right')

plt.title('Training and Validation Dice Loss')

plt.subplot(1, 2, 2)

plt.plot(epochs\_range, loss, label='Training Loss')

plt.plot(epochs\_range, val\_loss, label='Validation Loss')

plt.legend(loc='upper right')

plt.title('Training and Validation Loss')

plt.show()

即使只有5个时代，我们也看到了强劲的表现。

## 可视化实际性能

我们将在验证集上可视化我们的性能。

请注意，在实际设置（竞争，部署等）中，我们将在具有完整图像分辨率的测试集上进行评估。

要加载我们的模型，我们有两个选择：

1. 由于我们的模型架构已经在内存中，我们可以简单地调用 load\_weights(save\_model\_path)
2. 如果您想从头开始加载模型（在不同的设置中，而内存中没有模型架构），我们只需调用

model = models.load\_model(save\_model\_path, custom\_objects={'bce\_dice\_loss': bce\_dice\_loss, 'dice\_loss': dice\_loss})，具体说明我们用来训练模型的必要自定义对象，损失和指标。

如果您想查看更多示例，请查看我们的[keras指南](https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model)！

In [0]:

*# Alternatively, load the weights directly: model.load\_weights(save\_model\_path)*

model = models.load\_model(save\_model\_path, custom\_objects={'bce\_dice\_loss': bce\_dice\_loss,

'dice\_loss': dice\_loss})

In [0]:

*# Let's visualize some of the outputs*

data\_aug\_iter = val\_ds.make\_one\_shot\_iterator()

next\_element = data\_aug\_iter.get\_next()

*# Running next element in our graph will produce a batch of images*

plt.figure(figsize=(10, 20))

**for** i **in** range(5):

batch\_of\_imgs, label = tf.keras.backend.get\_session().run(next\_element)

img = batch\_of\_imgs[0]

predicted\_label = model.predict(batch\_of\_imgs)[0]

plt.subplot(5, 3, 3 \* i + 1)

plt.imshow(img)

plt.title("Input image")

plt.subplot(5, 3, 3 \* i + 2)

plt.imshow(label[0, :, :, 0])

plt.title("Actual Mask")

plt.subplot(5, 3, 3 \* i + 3)

plt.imshow(predicted\_label[:, :, 0])

plt.title("Predicted Mask")

plt.suptitle("Examples of Input Image, Label, and Prediction")

plt.show()

## 关键要点

在本教程中，我们学习了如何训练网络以自动检测和创建图像中的汽车切口！

**我们将涉及的具体概念：**

在此过程中，我们希望建立一些实践经验并围绕以下概念发展直觉

* [**功能API**](https://keras.io/getting-started/functional-api-guide/) - 我们使用Functional API实现了UNet。Functional API提供类似lego的API，允许我们构建几乎任何网络。
* **自定义损失和指标** - 我们实施了自定义指标，使我们能够准确了解培训期间的需求。此外，我们编写了一个特别适合我们任务的自定义损失函数。
* **保存并加载我们的模型** - 我们根据指定的指标保存了我们遇到的最佳模型。在想用最好的模型进行推理的时候，从磁盘加载了。请注意，保存模型捕获的不仅仅是模型的权重：默认情况下，它会保存模型体系结构，权重以及有关培训过程的信息，例如优化程序的状态等。

# 图像识别

我们的大脑让视觉变得容易。人类不需要做任何努力来分辨狮子和美洲虎，阅读标志或识别人类的脸。但这些实际上是用计算机解决的难题：它们看似简单，因为我们的大脑非常善于理解图像。

在过去几年中，机器学习领域在解决这些难题方面取得了巨大进步。特别地，我们发现一种称为深度[卷积神经网络](https://colah.github.io/posts/2014-07-Conv-Nets-Modular/)的模型 可以在硬视觉识别任务上实现合理的性能 - 在某些领域中匹配或超过人类表现。

研究人员通过验证他们对[ImageNet](http://www.image-net.org/)的工作证明了计算机视觉的稳步发展 - [ImageNet](http://www.image-net.org/)是计算机视觉的学术基准。连续模型继续显示出改进，每次都实现了最新的结果： [QuocNet](https://static.googleusercontent.com/media/research.google.com/en/archive/unsupervised_icml2012.pdf)，[AlexNet](https://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)，[Inception（GoogLeNet）](https://arxiv.org/abs/1409.4842)，[BN-Inception-v2](https://arxiv.org/abs/1502.03167)。谷歌内部和外部的研究人员已经发表了描述所有这些模型的论文，但结果仍然难以重现。我们现在正在采取下一步措施，在我们的最新型号[Inception-v3](https://arxiv.org/abs/1512.00567)上发布用于运行图像识别的代码。

Inception-v3 使用2012年的数据为[ImageNet](http://image-net.org/)大型视觉识别挑战赛进行了培训。这是计算机视觉的标准任务，模型试图将整个图像分为[1000个类别](http://image-net.org/challenges/LSVRC/2014/browse-synsets)，如“斑马”，“达尔马提亚”和“洗碗机” ”。例如，以下是[AlexNet](https://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)对某些图像进行分类的结果：

为了比较模型，我们检查模型无法预测正确答案的频率，作为他们的前5个猜测之一 - 称为“前5个错误率”。 [AlexNet](https://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)通过在2012年验证数据集上设置前15个错误率15.3％来实现; [成立（GoogLeNet）](https://arxiv.org/abs/1409.4842)达到6.67％; [BN-Inception-v2](https://arxiv.org/abs/1502.03167)达到4.9％; [初始-v3](https://arxiv.org/abs/1512.00567)达到3.46％。

人类在ImageNet Challenge上的表现如何？Andrej Karpathy 的[博客文章](https://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/)试图衡量自己的表现。他的前5错误率达到5.1％。

本教程将教您如何使用[Inception-v3](https://arxiv.org/abs/1512.00567)。您将学习如何使用Python或C ++ 将图像分类为[1000个类](http://image-net.org/challenges/LSVRC/2014/browse-synsets)。我们还将讨论如何从该模型中提取更高级别的特征，这些特征可以重用于其他视觉任务。

我们很高兴看到社区将对此模型做些什么。

## 用于Python API

classify\_image.py从tensorflow.org 程序第一次运行时下载训练的模型。您的硬盘上需要大约200M的可用空间。

首先从GitHub 克隆[TensorFlow模型repo](https://github.com/tensorflow/models)。运行以下命令：

cd models/tutorials/image/imagenet  
python classify\_image.py

上面的命令将对熊猫的提供图像进行分类。

如果模型正确运行，脚本将生成以下输出：

giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score = 0.88493)  
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00878)  
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score = 0.00317)  
custard apple (score = 0.00149)  
earthstar (score = 0.00127)

如果您希望提供其他JPEG图像，可以通过编辑--image\_file参数来完成。

如果将模型数据下载到其他目录，则需要指向--model\_dir 使用的目录。

## 使用C ++ API

您可以在C ++中运行相同的[Inception-v3](https://arxiv.org/abs/1512.00567)模型，以便在生产环境中使用。您可以下载包含定义模型的GraphDef的存档（从TensorFlow存储库的根目录运行）：

curl -L "https://storage.googleapis.com/download.tensorflow.org/models/inception\_v3\_2016\_08\_28\_frozen.pb.tar.gz" |  
  tar -C tensorflow/examples/label\_image/data -xz

接下来，我们需要编译包含加载和运行图形的代码的C ++二进制文件。如果您按照 [说明](https://tensorflow.google.cn/install/source) 为您的平台[下载TensorFlow的源安装](https://tensorflow.google.cn/install/source)，您应该能够通过从shell终端运行此命令来构建示例：

bazel build tensorflow/examples/label\_image/...

这应该创建一个二进制可执行文件，然后您可以像这样运行：

bazel-bin/tensorflow/examples/label\_image/label\_image

这使用框架附带的默认示例图像，并应输出类似于此的内容：

I tensorflow/examples/label\_image/main.cc:206] military uniform (653): 0.834306  
I tensorflow/examples/label\_image/main.cc:206] mortarboard (668): 0.0218692  
I tensorflow/examples/label\_image/main.cc:206] academic gown (401): 0.0103579  
I tensorflow/examples/label\_image/main.cc:206] pickelhaube (716): 0.00800814  
I tensorflow/examples/label\_image/main.cc:206] bulletproof vest (466): 0.00535088

在这种情况下，我们使用[Admiral Grace Hopper](https://en.wikipedia.org/wiki/Grace_Hopper)的默认图像 ，你可以看到网络正确识别她穿着军装，得分高达0.8。

接下来，通过提供--image =参数，例如，在您自己的图像上试一试

bazel-bin/tensorflow/examples/label\_image/label\_image --image=my\_image.png

如果您查看[tensorflow/examples/label\_image/main.cc](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc) 文件内部，您可以了解它是如何工作的。我们希望此代码可以帮助您将TensorFlow集成到您自己的应用程序中，因此我们将逐步完成主要功能：

命令行标志控制文件的加载位置和输入图像的属性。该模型期望获得299x299 RGB图像，因此这些是input\_width 和input\_height标志。我们还需要将像素值从0到255之间的整数缩放到图形操作的浮点值。我们使用input\_mean和input\_std标志控制缩放：我们首先input\_mean从每个像素值中减去 ，然后除以input\_std。

这些值可能看起来有点神奇，但它们只是由原始模型作者根据他/她想要用作训练的输入图像来定义。如果您有自己训练的图表，则只需调整值以匹配您在训练过程中使用的任何值。

您可以看到它们如何应用于[ReadTensorFromImageFile()](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L88) 函数中的图像 。

// Given an image file name, read in the data, try to decode it as an image,  
// resize it to the requested size, and then scale the values as desired.  
Status ReadTensorFromImageFile(string file\_name, const int input\_height,  
                               const int input\_width, const float input\_mean,  
                               const float input\_std,  
                               std::vector<Tensor>\* out\_tensors) {  
  tensorflow::GraphDefBuilder b;

我们首先创建一个GraphDefBuilder，这是一个我们可以用来指定要运行或加载的模型的对象。

  string input\_name = "file\_reader";  
  string output\_name = "normalized";  
  tensorflow::Node\* file\_reader =  
      tensorflow::ops::ReadFile(tensorflow::ops::Const(file\_name, b.opts()),  
                                b.opts().WithName(input\_name));

然后，我们开始为我们想要运行的小模型创建节点，以加载，调整大小和缩放像素值，以获得主模型期望作为其输入的结果。我们创建的第一个节点只是一个Constop，它包含一个张量，其中包含我们要加载的图像的文件名。然后将其作为ReadFileop 的第一个输入传递。您可能会注意到我们将b.opts()作为所有op创建函数的最后一个参数传递。该参数确保将节点添加到保存在的模型定义中GraphDefBuilder。我们也ReadFile 通过WithName()拨打电话来命名运营商b.opts()。这给了节点一个名称，这不是绝对必要的，因为如果不这样做，将分配自动名称，但它确实使调试更容易。

  // Now try to figure out what kind of file it is and decode it.  
  const int wanted\_channels = 3;  
  tensorflow::Node\* image\_reader;  
  if (tensorflow::StringPiece(file\_name).ends\_with(".png")) {  
    image\_reader = tensorflow::ops::DecodePng(  
        file\_reader,  
        b.opts().WithAttr("channels", wanted\_channels).WithName("png\_reader"));  
  } else {  
    // Assume if it's not a PNG then it must be a JPEG.  
    image\_reader = tensorflow::ops::DecodeJpeg(  
        file\_reader,  
        b.opts().WithAttr("channels", wanted\_channels).WithName("jpeg\_reader"));  
  }  
  // Now cast the image data to float so we can do normal math on it.  
  tensorflow::Node\* float\_caster = tensorflow::ops::Cast(  
      image\_reader, tensorflow::DT\_FLOAT, b.opts().WithName("float\_caster"));  
  // The convention for image ops in TensorFlow is that all images are expected  
  // to be in batches, so that they're four-dimensional arrays with indices of  
  // [batch, height, width, channel]. Because we only have a single image, we  
  // have to add a batch dimension of 1 to the start with ExpandDims().  
  tensorflow::Node\* dims\_expander = tensorflow::ops::ExpandDims(  
      float\_caster, tensorflow::ops::Const(0, b.opts()), b.opts());  
  // Bilinearly resize the image to fit the required dimensions.  
  tensorflow::Node\* resized = tensorflow::ops::ResizeBilinear(  
      dims\_expander, tensorflow::ops::Const({input\_height, input\_width},  
                                            b.opts().WithName("size")),  
      b.opts());  
  // Subtract the mean and divide by the scale.  
  tensorflow::ops::Div(  
      tensorflow::ops::Sub(  
          resized, tensorflow::ops::Const({input\_mean}, b.opts()), b.opts()),  
      tensorflow::ops::Const({input\_std}, b.opts()),  
      b.opts().WithName(output\_name));

然后我们继续添加更多节点，将文件数据解码为图像，将整数转换为浮点值，调整大小，然后最终对像素值运行减法和除法运算。

  // This runs the GraphDef network definition that we've just constructed, and  
  // returns the results in the output tensor.  
  tensorflow::GraphDef graph;  
  TF\_RETURN\_IF\_ERROR(b.ToGraphDef(&graph));

最后，我们有一个存储在b变量中的模型定义，我们将其转换为带有该ToGraphDef()函数的完整图形定义。

  std::unique\_ptr<tensorflow::Session> session(  
      tensorflow::NewSession(tensorflow::SessionOptions()));  
  TF\_RETURN\_IF\_ERROR(session->Create(graph));  
  TF\_RETURN\_IF\_ERROR(session->Run({}, {output\_name}, {}, out\_tensors));  
  return Status::OK();

然后我们创建一个[tf.Session](https://tensorflow.google.cn/api_docs/python/tf/Session) 对象，它是实际运行图形的接口，并运行它，指定我们想要从哪个节点获取输出，以及放置输出数据的位置。

这给了我们一个Tensor对象的向量，在这种情况下，我们知道只有一个对象。Tensor在这种情况下，您可以将a 视为一个多维数组，它可以保持299像素高，299像素宽，3通道图像作为浮点值。如果您的产品中已经有自己的图像处理框架，那么只要在将图像输入主图之前应用相同的转换，您就应该可以使用它。

这是在C ++中动态创建小型TensorFlow图的简单示例，但对于预先训练的Inception模型，我们希望从文件中加载更大的定义。您可以在LoadGraph()函数中看到我们如何做到这一点。

// Reads a model graph definition from disk, and creates a session object you  
// can use to run it.  
Status LoadGraph(string graph\_file\_name,  
                 std::unique\_ptr<tensorflow::Session>\* session) {  
  tensorflow::GraphDef graph\_def;  
  Status load\_graph\_status =  
      ReadBinaryProto(tensorflow::Env::Default(), graph\_file\_name, &graph\_def);  
  if (!load\_graph\_status.ok()) {  
    return tensorflow::errors::NotFound("Failed to load compute graph at '",  
                                        graph\_file\_name, "'");  
  }

如果您查看了图像加载代码，很多术语应该很熟悉。我们不是使用a GraphDefBuilder来生成GraphDef对象，而是加载一个直接包含的protobuf文件GraphDef。

  session->reset(tensorflow::NewSession(tensorflow::SessionOptions()));  
  Status session\_create\_status = (\*session)->Create(graph\_def);  
  if (!session\_create\_status.ok()) {  
    return session\_create\_status;  
  }  
  return Status::OK();  
}

然后我们从中创建一个Session对象GraphDef并将其传递给调用者，以便他们以后可以运行它。

该GetTopLabels()函数很像图像加载，除了在这种情况下我们想要获取运行主图的结果，并将其转换为得分最高的标签的排序列表。就像图像加载器一样，它创建一个GraphDefBuilder，为它添加几个节点，然后运行短图形以获得一对输出张量。在这种情况下，它们代表最高结果的排序分数和索引位置。

// Analyzes the output of the Inception graph to retrieve the highest scores and  
// their positions in the tensor, which correspond to categories.  
Status GetTopLabels(const std::vector<Tensor>& outputs, int how\_many\_labels,  
                    Tensor\* indices, Tensor\* scores) {  
  tensorflow::GraphDefBuilder b;  
  string output\_name = "top\_k";  
  tensorflow::ops::TopK(tensorflow::ops::Const(outputs[0], b.opts()),  
                        how\_many\_labels, b.opts().WithName(output\_name));  
  // This runs the GraphDef network definition that we've just constructed, and  
  // returns the results in the output tensors.  
  tensorflow::GraphDef graph;  
  TF\_RETURN\_IF\_ERROR(b.ToGraphDef(&graph));  
  std::unique\_ptr<tensorflow::Session> session(  
      tensorflow::NewSession(tensorflow::SessionOptions()));  
  TF\_RETURN\_IF\_ERROR(session->Create(graph));  
  // The TopK node returns two outputs, the scores and their original indices,  
  // so we have to append :0 and :1 to specify them both.  
  std::vector<Tensor> out\_tensors;  
  TF\_RETURN\_IF\_ERROR(session->Run({}, {output\_name + ":0", output\_name + ":1"},  
                                  {}, &out\_tensors));  
  \*scores = out\_tensors[0];  
  \*indices = out\_tensors[1];  
  return Status::OK();

该PrintTopLabels()函数采用这些排序结果，并以友好的方式打印出来。该CheckTopLabel()函数非常相似，但只是确保顶部标签是我们期望的，用于调试目的。

最后，[main()](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L252) 将所有这些呼叫联系在一起。

int main(int argc, char\* argv[]) {  
  // We need to call this to set up global state for TensorFlow.  
  tensorflow::port::InitMain(argv[0], &argc, &argv);  
  Status s = tensorflow::ParseCommandLineFlags(&argc, argv);  
  if (!s.ok()) {  
    LOG(ERROR) << "Error parsing command line flags: " << s.ToString();  
    return -1;  
  }  
  
  // First we load and initialize the model.  
  std::unique\_ptr<tensorflow::Session> session;  
  string graph\_path = tensorflow::io::JoinPath(FLAGS\_root\_dir, FLAGS\_graph);  
  Status load\_graph\_status = LoadGraph(graph\_path, &session);  
  if (!load\_graph\_status.ok()) {  
    LOG(ERROR) << load\_graph\_status;  
    return -1;  
  }

我们加载主图。

  // Get the image from disk as a float array of numbers, resized and normalized  
  // to the specifications the main graph expects.  
  std::vector<Tensor> resized\_tensors;  
  string image\_path = tensorflow::io::JoinPath(FLAGS\_root\_dir, FLAGS\_image);  
  Status read\_tensor\_status = ReadTensorFromImageFile(  
      image\_path, FLAGS\_input\_height, FLAGS\_input\_width, FLAGS\_input\_mean,  
      FLAGS\_input\_std, &resized\_tensors);  
  if (!read\_tensor\_status.ok()) {  
    LOG(ERROR) << read\_tensor\_status;  
    return -1;  
  }  
  const Tensor& resized\_tensor = resized\_tensors[0];

加载，调整大小和处理输入图像。

  // Actually run the image through the model.  
  std::vector<Tensor> outputs;  
  Status run\_status = session->Run({ {FLAGS\_input\_layer, resized\_tensor}},  
                                   {FLAGS\_output\_layer}, {}, &outputs);  
  if (!run\_status.ok()) {  
    LOG(ERROR) << "Running model failed: " << run\_status;  
    return -1;  
  }

这里我们运行加载的图形，并将图像作为输入。

  // This is for automated testing to make sure we get the expected result with  
  // the default settings. We know that label 866 (military uniform) should be  
  // the top label for the Admiral Hopper image.  
  if (FLAGS\_self\_test) {  
    bool expected\_matches;  
    Status check\_status = CheckTopLabel(outputs, 866, &expected\_matches);  
    if (!check\_status.ok()) {  
      LOG(ERROR) << "Running check failed: " << check\_status;  
      return -1;  
    }  
    if (!expected\_matches) {  
      LOG(ERROR) << "Self-test failed!";  
      return -1;  
    }  
  }

出于测试目的，我们可以检查以确保获得我们期望的输出。

  // Do something interesting with the results we've generated.  
  Status print\_status = PrintTopLabels(outputs, FLAGS\_labels);

最后，我们打印出我们找到的标签。

  if (!print\_status.ok()) {  
    LOG(ERROR) << "Running print failed: " << print\_status;  
    return -1;  
  }

这里的错误处理是使用TensorFlow的Status 对象，这非常方便，因为它可以让您知道ok()检查器是否发生任何错误，然后可以打印出来以提供可读的错误消息。

在这种情况下，我们正在演示对象识别，但您应该能够在各种域中使用与您自己找到或训练过的其他模型非常相似的代码。我们希望这个小例子能为您提供有关如何在自己的产品中使用TensorFlow的一些想法。

**练习**：转移学习的理念是，如果你知道如何很好地解决任务，你应该能够将一些理解转移到解决相关问题上。执行转移学习的一种方式是移除网络的最终分类层并提取[CNN的倒数第二层](https://arxiv.org/abs/1310.1531)，在这种情况下是2048维向量。

## 更多学习资源

要了解一般的神经网络，Michael Nielsen的 [免费在线书籍](http://neuralnetworksanddeeplearning.com/chap1.html) 是一个很好的资源。特别是对于卷积神经网络，克里斯奥拉有一些 [不错的博客文章](https://colah.github.io/posts/2014-07-Conv-Nets-Modular/)，迈克尔尼尔森的书有一个 [很好的章节](http://neuralnetworksanddeeplearning.com/chap6.html) 涵盖他们。

要了解有关实现卷积神经网络的更多信息，您可以跳转到TensorFlow [深层卷积网络教程](https://tensorflow.google.cn/tutorials/images/deep_cnn)，或者使用我们的[Estimator MNIST教程](https://tensorflow.google.cn/tutorials/estimators/cnn)更轻松地开始。最后，如果您想加快这方面的研究速度，您可以阅读本教程中引用的所有论文的最新工作。

# 如何为新类别重新划分图像分类器

现代图像识别模型具有数百万个参数。从头开始训练需要大量标记的训练数据和大量计算能力（数百小时GPU或更多）。转移学习是一种通过采用已经在相关任务上训练并在新模型中重复使用的模型的一部分来快速实现这一目标的技术。在本教程中，我们将重用ImageNet上训练的强大图像分类器的特征提取功能，并简单地在顶部训练新的分类层。有关该方法的更多信息，您可以[在Decaf上](https://arxiv.org/abs/1310.1531)看到[这篇论文](https://arxiv.org/abs/1310.1531)。

虽然它不如训练整个模型那么好，但对于许多应用来说这是非常有效的，适用于适量的训练数据（数千，而不是数百万标记的图像），并且可以在笔记本电脑上运行30分钟而不需要一个GPU。本教程将向您展示如何在您自己的图像上运行示例脚本，并将解释您有助于控制培训过程的一些选项。

**注意：**本教程的一个版本也可 [作为codelab使用](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets/#0)。

本教程使用[TensorFlow Hub](https://tensorflow.google.cn/hub/index)来调用预先训练过的模型或模块。对于初学者，我们将使用 具有在ImageNet上训练的Inception V3架构的[图像特征提取模块](https://tensorflow.google.cn/hub/modules/google/imagenet/inception_v3/feature_vector/1)，并[稍后](https://tensorflow.google.cn/hub/tutorials/image_retraining#other_architectures)返回其他选项，包括 [NASNet](https://research.googleblog.com/2017/11/automl-for-large-scale-image.html)/ PNASNet，以及 [MobileNet V1](https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html)和 [V2](https://research.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html)。

在开始之前，您需要安装PIP包tensorflow-hub以及最新版本的TensorFlow。有关详细信息，请参阅TensorFlow Hub的[安装说明](https://tensorflow.google.cn/hub/installation)。

## 花卉培训

[图片来自Kelly Sikkema](https://www.flickr.com/photos/95072945@N05/9922116524/)

在开始任何培训之前，您需要一组图像来向网络传授您想要识别的新类。后面的部分介绍了如何准备自己的图像，但为了方便起见，我们创建了一个创建公共许可花卉照片的存档，以便最初使用。要获取花卉照片集，请运行以下命令：

cd ~  
curl -LO http://download.tensorflow.org/example\_images/flower\_photos.tgz  
tar xzf flower\_photos.tgz

获得图像后，可以从GitHub下载示例代码（它不是库安装的一部分）：

mkdir ~/example\_code  
cd ~/example\_code  
curl -LO https://github.com/tensorflow/hub/raw/master/examples/image\_retraining/retrain.py

在最简单的情况下，训练师可以这样运行（大约需要半小时）：

python retrain.py --image\_dir ~/flower\_photos

该脚本还有许多其他选项。您可以通过以下方式获得完整列表：

python retrain.py -h

此脚本加载预先训练的模块，并在您下载的花卉照片的顶部训练新的分类器。在完整的网络训练过程中，没有一种花卉种类属于原始的ImageNet类。转移学习的神奇之处在于，经过训练以区分某些对象的较低层可以重复用于许多识别任务而无需任何改动。

## 瓶颈

根据机器的速度，脚本可能需要30分钟或更长时间才能完成。第一阶段分析磁盘上的所有映像，并计算和缓存每个映像的瓶颈值。'瓶颈'是一个非正式术语，我们经常在实际进行分类的最终输出层之前使用该层。（TensorFlow Hub将其称为“图像特征向量”。）此倒数第二层已经过训练，可以输出一组足够好的值，分类器可以用它来区分要求识别的所有类。这意味着它必须是一个有意义且紧凑的图像摘要，因为它必须包含足够的信息，以便分类器在一组非常小的值中做出正确的选择。

因为每个图像在训练期间多次重复使用并且计算每个瓶颈需要花费大量时间，所以它会加速将这些瓶颈值缓存到磁盘上，因此不必反复重新计算它们。默认情况下，它们存储在/tmp/bottleneck目录中，如果重新运行脚本，它们将被重用，因此您不必再次等待此部分。

## 训练

一旦瓶颈完成，就开始对网络顶层进行实际培训。您将看到一系列步骤输出，每个步骤输出显示训练准确性，验证准确性和交叉熵。训练准确性显示当前训练批次中使用的图像百分比标记为正确的类别。验证准确度是来自不同组的随机选择的图像组的精度。关键的区别在于训练精度基于网络能够学习的图像，因此网络可以过度拟合训练数据中的噪声。衡量网络性能的真正标准是测量其在训练数据中未包含的数据集上的性能 - 这是通过验证准确度来衡量的。如果列车精度高但验证精度仍然很低，则意味着网络过度拟合并记住训练图像中的特定特征，这些特征对于更一般无用。交叉熵是一种损失函数，可以让我们一瞥学习过程的进展情况。培训的目标是尽可能减少损失，因此您可以通过关注损失是否保持向下趋势，忽略短期噪音来判断学习是否有效。

默认情况下，此脚本将运行4,000个训练步骤。每个步骤从训练集中随机选择十个图像，从缓存中找到它们的瓶颈，并将它们输入到最后一层以获得预测。然后将这些预测与实际标签进行比较，以通过反向传播过程更新最终层的权重。随着过程的继续，您应该看到报告的准确度得到改善，并且在完成所有步骤之后，对与训练和验证图片分开的一组图像运行最终测试准确度评估。该测试评估是训练模型将如何在分类任务上执行的最佳估计。您应该看到准确度值介于90％和95％之间，但是由于训练过程中的随机性，准确值会因批次不同而不同。

## 使用TensorBoard可视化再培训

该脚本包含TensorBoard摘要，可以更容易理解，调试和优化再培训。例如，您可以可视化图形和统计数据，例如在训练期间权重或准确度如何变化。

要启动TensorBoard，请在重新训练期间或之后运行此命令：

tensorboard --logdir /tmp/retrain\_logs

TensorBoard运行后，导航您的Web浏览器 [localhost:6006](http://localhost:6006/)以查看TensorBoard。

该retrain.py脚本/tmp/retrain\_logs 默认会将TensorBoard摘要记录到。您可以使用--summaries\_dir标志更改目录。

[TensorBoard的GitHub存储库](https://github.com/tensorflow/tensorboard) 提供了有关TensorBoard使用的更多信息，包括提示和技巧以及调试信息。

## 使用再培训模型

该脚本将写出在您的类别上训练的新模型 /tmp/output\_graph.pb，以及包含标签的文本文件/tmp/output\_labels.txt。新模型包含内嵌的TF-Hub模块和新的分类层。这两个文件都采用[C ++和Python图像分类示例](https://tensorflow.google.cn/tutorials/image_recognition) 可以读入的格式，因此您可以立即开始使用新模型。由于您已经替换了顶层，因此您需要在脚本中指定新名称，例如，--output\_layer=final\_result如果您使用label\_image，则使用该标志。

以下是如何使用重新训练的图形运行label\_image示例的示例。按照惯例，所有TensorFlow Hub模块都接受具有固定范围[0,1]中颜色值的图像输入，因此您无需设置 --input\_mean或--input\_std标记。

curl -LO https://github.com/tensorflow/tensorflow/raw/master/tensorflow/examples/label\_image/label\_image.py  
python label\_image.py \  
--graph=/tmp/output\_graph.pb --labels=/tmp/output\_labels.txt \  
--input\_layer=Placeholder \  
--output\_layer=final\_result \  
--image=$HOME/flower\_photos/daisy/21652746\_cc379e0eea\_m.jpg

你应该看到一个花卉标签列表，在大多数情况下顶部有菊花（虽然每个重新训练的模型可能会略有不同）。您可以将--image参数替换为 您自己的图像以尝试这些参数。

如果您想在自己的Python程序中使用重新训练的模型，那么上面的label\_image脚本是一个合理的起点。该label\_image 目录还包含C ++代码，您可以将其用作模板以将tensorflow与您自己的应用程序集成。

如果您发现默认的Inception V3模块对于您的应用程序来说太大或太慢，请查看 下面的“ [其他模型架构”部分](https://tensorflow.google.cn/hub/tutorials/image_retraining#other_architectures)，了解加快和缩小网络的选项。

## 训练你自己的类别

如果您已经设法让脚本处理花卉示例图像，您可以开始考虑教它来识别您关心的类别。理论上，您需要做的就是将其指向一组子文件夹，每个子文件夹以您的一个类别命名，并且仅包含该类别的图像。如果你这样做并将子目录的根文件夹作为参数传递给 --image\_dir，那么脚本应该像对花一样训练。

以下是鲜花存档的文件夹结构，为您提供脚本所需布局类型的示例：

在实践中，可能需要一些工作来获得您想要的准确性。我将尝试引导您解决下面可能遇到的一些常见问题。

## 创建一套训练图像

首先要看的是你收集的图像，因为我们通过培训看到的最常见的问题来自于被输入的数据。

为了使训练更好地运作，您应该收集至少一百张您想要识别的每种物体的照片。您收集的越多，您训练的模型的准确性就越高。您还需要确保照片很好地代表了您的应用程序实际遇到的内容。例如，如果您将所有照片都放在室内空白墙上​​并且用户试图在户外识别物体，则部署时可能看不到好的结果。

要避免的另一个缺陷是，学习过程会对标记图像彼此相同的任何内容产生影响，如果你不小心，那可能是没用的东西。例如，如果您在蓝色房间中拍摄一种物体，而另一种物体在绿色物体中拍摄，则模型最终会根据背景颜色进行预测，而不是您实际关注的物体的特征。为避免这种情况，请尝试在不同时间和不同设备上尽可能多地拍摄照片。

您可能还想考虑您使用的类别。将大量不同物理形式的大类别划分为更具视觉冲突力的小类别可能是值得的。例如，您可以使用“汽车”，“摩托车”和“卡车”代替“车辆”。同样值得思考的是你是否有“封闭世界”或“开放世界”问题。在一个封闭的世界中，你唯一要求分类的东西就是你所知道的对象类别。这可能适用于您知道用户可能正在拍摄花卉照片的植物识别应用程序，因此您所要做的就是决定使用哪种物种。相比之下，漫游机器人可以通过其相机在世界各地漫游时看到各种不同的东西。在那种情况下你' d希望分类器报告它是否不确定它看到了什么。这可能很难做到，但通常如果您收集大量典型的“背景”照片而其中没有相关对象，您可以将它们添加到图像文件夹中的额外“未知”类。

还需要检查以确保所有图像都标记正确。用户生成的标签通常不能用于我们的目的。例如：标记的图片#daisy可能还包括名为Daisy的人物和角色。如果您浏览图像并清除任何错误，它可以为您的整体准确性创造奇迹。

## 培训步骤

如果您对图像感到满意，可以通过更改学习过程的详细信息来了解改善结果的方法。最简单的尝试是 --how\_many\_training\_steps。默认为4,000，但如果将其增加到8,000，它将训练两倍。准确度的提高速度会减慢你训练的时间，并且在某些时候会完全停止（甚至因过度拟合而下降），但你可以尝试看看什么最适合你的模型。

## 扭曲

改善图像训练结果的常用方法是以随机方式使训练输入变形，裁剪或增亮。由于相同图像的所有可能变化，这具有扩展训练数据的有效大小的优点，并且倾向于帮助网络学习应对将在分类器的实际使用中发生的所有失真。在我们的脚本中启用这些失真的最大缺点是瓶颈缓存不再有用，因为输入图像永远不会被完全重用。这意味着训练过程需要更长的时间（许多小时），因此建议您尝试将此作为一种抛光模型的方法，只有在您拥有一个相当满意的模型之后。

您可以通过将启用这些扭曲--random\_crop，--random\_scale并 --random\_brightness给脚本。这些都是控制每个图像应用了多少失真的百分比值。从每个值的5或10开始是合理的，然后试验看哪些值对您的应用有帮助。--flip\_left\_right将水平地随机镜像一半图像，只要这些反转可能在您的应用程序中发生，这是有道理的。例如，如果你试图识别字母，这不是一个好主意，因为翻转它们会破坏它们的含义。

## 超参数

您可以尝试调整其他几个参数，看看它们是否有助于您的结果。该--learning\_rate控件更新到最后一层的训练过程中的大小。直观地，如果这个小于学习将花费更长时间，但它最终可以帮助整体精确度。但情况并非总是这样，所以你需要仔细研究，看看哪种方法适合你的情况。的--train\_batch\_size多少图像中的每个训练步骤被检查控制来估计更新的最终层。

## 培训，验证和测试集

当您将脚本指向图像文件夹时，脚本所做的一件事就是将它们分成三组。最大的通常是训练集，它们是训练期间馈入网络的所有图像，其结果用于更新模型的权重。您可能想知道为什么我们不使用所有图像进行培训？当我们进行机器学习时，一个很大的潜在问题是我们的模型可能只是记住训练图像的不相关细节，以得出正确的答案。例如，您可以想象一个网络在其显示的每张照片的背景中记住一个图案，并使用它来匹配标签与对象。它可以在训练前看到的所有图像上产生良好的效果，但是因为它'

这个问题被称为过度拟合，为了避免它，我们将一些数据保留在训练过程之外，这样模型就无法记住它们。然后我们使用这些图像作为检查以确保不会发生过度拟合，因为如果我们看到它们具有良好的准确性，则表明网络没有过度拟合是一个好兆头。通常的分割是将80％的图像放入主训练集中，保留10％以备在训练期间经常作为验证运行，然后将最终的10％作为测试集用于预测实际情况世界表现的分类器。可以使用--testing\_percentage和--validation\_percentage标志来控制这些比率 。一般情况下，您应该能够将这些值保留为默认值，因为通常不会发现培训调整它们的任何优势。

请注意，该脚本使用图像文件名（而不是完全随机的函数）在训练，验证和测试集之间划分图像。这样做是为了确保图像不会在不同运行的训练集和测试集之间移动，因为如果用于训练模型的图像随后在验证集中使用，那么这可能是一个问题。

您可能会注意到验证准确度在迭代之间波动。大部分这种波动源于这样的事实：为每个验证精度测量选择验证集的随机子集。通过选择--validation\_batch\_size=-1使用整个验证集进行每次精度计算，可以大大减少波动，代价是培训时间有所增加 。

培训完成后，您可能会发现检查测试集中错误分类的图像非常有见地。这可以通过添加标志来完成 --print\_misclassified\_test\_images。这可以帮助您了解哪种类型的图像最容易混淆模型，以及哪些类别最难以区分。例如，您可能会发现特定类别的某些子类型或某些不寻常的照片角度特别难以识别，这可能会鼓励您添加该子类型的更多训练图像。通常，检查错误分类的图像也可能指向输入数据集中的错误，例如错误标记，低质量或模糊图像。然而，通常应该避免在测试集中修正个别错误，因为它们可能仅仅反映（更大）训练集中的更一般的问题。

## 其他模型架构

默认情况下，脚本使用带有Inception V3体系结构预训练实例的图像特征提取模块。这是一个很好的起点，因为它为再训练脚本提供了高精度的结果和适中的运行时间。但现在让我们来看看[TensorFlow Hub模块的其他选项](https://tensorflow.google.cn/hub/modules/image)。

在一方面，该名单显示更近，强大的架构，如 [NASNet](https://research.googleblog.com/2017/11/automl-for-large-scale-image.html) （特别是 [nasnet\_large](https://tensorflow.google.cn/hub/modules/google/imagenet/nasnet_large/feature_vector/1) 和[pnasnet\_large](https://tensorflow.google.cn/hub/modules/google/imagenet/pnasnet_large/feature_vector/2)），这可能给你一些额外的精度。

另一方面，如果您打算在移动设备或其他资源受限的环境中部署模型，您可能希望以更小的文件大小或更快的速度（也在培训中）交换一点精度。为此，尝试 实现[MobileNet V1](https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html) 或 [V2](https://research.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html) 架构的不同 [模块](https://tensorflow.google.cn/hub/modules/image#mobilenet)，或者也可以 。[nasnet\_mobile](https://tensorflow.google.cn/hub/modules/google/imagenet/nasnet_mobile/feature_vector/1)

使用不同模块进行培训很简单：只需--tfhub\_module 使用模块URL 传递标志，例如：

python retrain.py \  
    --image\_dir ~/flower\_photos \  
    --tfhub\_module https://tfhub.dev/google/imagenet/mobilenet\_v2\_100\_224/feature\_vector/2

这将创建一个9 MB模型文件，/tmp/output\_graph.pb其中包含使用MobileNet V2基准版本的模型。在浏览器中打开模块URL将转到模块文档。

如果你只是想让它快一点，你可以将输入图像（第二个数字）的大小从'224'减小到'192'，'160'或'128'像素的平方，甚至'96' （仅适用于V2）。为了更积极的节省，您可以选择百分比（第一个数字）'100'，'075'，'050'或'035'（V1的'025'）来控制“特征深度”或每个神经元的数量位置。权重的数量（以及文件大小和速度）随着该分数的平方而缩小。该[MobileNet V1的博文](https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html) ，并[在GitHub上MobileNet V2页](https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet) 上Imagenet分类相应的权衡报告。

Mobilenet V2不会将功能深度百分比应用于瓶颈层。Mobilenet V1做到了，这使得分类层的工作对于小深度更难。是否有助于欺骗和使用原始1001 ImageNet类的分数而不是严格的瓶颈？你可以简单地通过更换尝试 mobilenet\_v1.../feature\_vector与mobilenet\_v1.../classification 在模块的名字。

和以前一样，您可以使用所有重新训练的模型label\_image.py。您需要指定模型所需的图像大小，例如：

python label\_image.py \  
--graph=/tmp/output\_graph.pb --labels=/tmp/output\_labels.txt \  
--input\_layer=Placeholder \  
--output\_layer=final\_result \  
--input\_height=224 --input\_width=224 \  
--image=$HOME/flower\_photos/daisy/21652746\_cc379e0eea\_m.jpg

有关将重新训练的模型部署到移动设备的更多信息，请参阅 本教程的[codelab版本](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets/#0)，尤其是[第2部分](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets-2-tflite/#0)，其中介绍了 [TensorFlow Lite](https://tensorflow.google.cn/mobile/tflite/)及其提供的其他优化（包括模型权重的量化）。

# 先进的卷积神经网络

## 概观

CIFAR-10分类是机器学习中常见的基准问题。问题是将RGB 32x32像素图像分为10类：

airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

有关更多详细信息，请参阅[CIFAR-10页面](https://www.cs.toronto.edu/~kriz/cifar.html) 和 Alex Krizhevsky 的[技术报告](https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf)。

### 目标

本教程的目标是构建一个相对较小的[卷积神经网络](https://en.wikipedia.org/wiki/Convolutional_neural_network)（CNN）来识别图像。在这个过程中，本教程：

1. 突出显示网络架构，培训和评估的规范组织。
2. 提供用于构建更大和更复杂模型的模板。

选择CIFAR-10的原因在于它足够复杂，可以运用大部分TensorFlow扩展到大型模型的能力。同时，该模型足够小，可以快速训练，这是尝试新想法和尝试新技术的理想选择。

### 教程的亮点

CIFAR-10教程演示了在TensorFlow中设计更大和更复杂模型的几个重要结构：

* 核心数学组件包括[tf.nn.conv2d](https://tensorflow.google.cn/api_docs/python/tf/nn/conv2d) （[wiki](https://en.wikipedia.org/wiki/Convolution)）， [tf.nn.relu](https://tensorflow.google.cn/api_docs/python/tf/nn/relu) （[wiki](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))）， [tf.nn.max\_pool](https://tensorflow.google.cn/api_docs/python/tf/nn/max_pool)（[wiki](https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer)）和[tf.nn.local\_response\_normalization](https://tensorflow.google.cn/api_docs/python/tf/nn/local_response_normalization) （[AlexNet论文中的](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf)第3.3章 ）。
* 在培训期间[可视化](https://tensorflow.google.cn/guide/summaries_and_tensorboard)网络活动，包括输入图像，激活和梯度的损失和分布。
* 用于计算[tf.train.ExponentialMovingAverage](https://tensorflow.google.cn/api_docs/python/tf/train/ExponentialMovingAverage) 学习参数并在评估期间使用这些平均值来提高预测性能的例程 。
* [tf.train.exponential\_decay](https://tensorflow.google.cn/api_docs/python/tf/train/exponential_decay) 随着时间的推移系统地减少的实施 。
* 预取[tf.train.shuffle\_batch](https://tensorflow.google.cn/api_docs/python/tf/train/shuffle_batch) 输入数据以隔离模型与磁盘延迟和昂贵的图像预处理。

我们还提供了 该模型的[多GPU版本](https://tensorflow.google.cn/tutorials/images/deep_cnn#training_a_model_using_multiple_gpu_cards)，演示了：

* 配置模型以并行训练多个GPU卡。
* 在多个GPU之间共享和更新变量。

我们希望本教程提供一个启动点，用于在TensorFlow上构建用于视觉任务的更大的CNN。

### 模型架构

该CIFAR-10教程中的模型是一个由交替卷积和非线性组成的多层架构。这些层之后是完全连接的层，通向softmax分类器。该模型遵循[Alex Krizhevsky](https://code.google.com/p/cuda-convnet/)描述的架构 ，前几个层次有一些差异。

该模型在GPU上的几小时训练时间内实现了约86％准确度的峰值性能。请参阅[下面](https://tensorflow.google.cn/tutorials/images/deep_cnn#evaluating_a_model)的代码和详细信息。它由1,068,298个可学习参数组成，需要大约19.5M乘法加法运算来计算单个图像的推断。

## 代码组织

本教程的代码位于 [models/tutorials/image/cifar10/](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/)。

| 文件 | 目的 |
| --- | --- |
| [cifar10\_input.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_input.py) | 读取本机CIFAR-10二进制文件格式。 |
| [cifar10.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10.py) | 构建CIFAR-10模型。 |
| [cifar10\_train.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_train.py) | 在CPU或GPU上训练CIFAR-10模型。 |
| [cifar10\_multi\_gpu\_train.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_multi_gpu_train.py) | 在多个GPU上训练CIFAR-10模型。 |
| [cifar10\_eval.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_eval.py) | 评估CIFAR-10模型的预测性能。 |

## CIFAR-10型号

CIFAR-10网络主要包含在 [cifar10.py](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10.py)。完整的培训图包含大约765个操作。我们发现通过使用以下模块构建图形，我们可以使代码最具可重用性：

1. [**模型输入：**](https://tensorflow.google.cn/tutorials/images/deep_cnn#model_inputs) inputs()并分别distorted\_inputs()添加读取和预处理CIFAR图像以进行评估和培训的操作。
2. [**模型预测：**](https://tensorflow.google.cn/tutorials/images/deep_cnn#model_prediction) inference() 添加对提供的图像执行推理（即分类）的操作。
3. [**模型训练：**](https://tensorflow.google.cn/tutorials/images/deep_cnn#model_training) loss()并train() 添加计算损失，梯度，变量更新和可视化摘要的操作。

### 模型输入

模型的输入部分由函数构建，inputs()并 distorted\_inputs()从CIFAR-10二进制数据文件中读取图像。这些文件包含固定的字节长度记录，因此我们使用[tf.FixedLengthRecordReader](https://tensorflow.google.cn/api_docs/python/tf/FixedLengthRecordReader)。请参阅[阅读数据](https://tensorflow.google.cn/api_guides/python/reading_data#reading-from-files)以了解有关Reader该类如何工作的更多信息。

图像处理如下：

* 它们被裁剪为24 x 24像素，集中用于评估或 [tf.random\_crop](https://tensorflow.google.cn/api_docs/python/tf/random_crop)培训。
* 它们是[tf.image.per\_image\_standardization](https://tensorflow.google.cn/api_docs/python/tf/image/per_image_standardization) 为了使模型对动态范围不敏感。

对于培训，我们还应用一系列随机失真来人为地增加数据集大小：

* [tf.image.random\_flip\_left\_right](https://tensorflow.google.cn/api_docs/python/tf/image/random_flip_left_right) 图像从左到右。
* 随机扭曲了[tf.image.random\_brightness](https://tensorflow.google.cn/api_docs/python/tf/image/random_brightness)。
* 随机扭曲了[tf.image.random\_contrast](https://tensorflow.google.cn/api_docs/python/tf/image/random_contrast)。

请参阅[图像](https://tensorflow.google.cn/api_guides/python/image)页面以获取可用的扭曲列表。我们还附加了一个 [tf.summary.image](https://tensorflow.google.cn/api_docs/python/tf/summary/image)图像，以便我们可以在[TensorBoard中](https://tensorflow.google.cn/guide/summaries_and_tensorboard)显示它们。这是验证输入是否正确构建的好方法。

从磁盘读取图像并扭曲它们可以使用非常少量的处理时间。为了防止这些操作减慢训练速度，我们在16个单独的线程中运行它们，这些线程不断填充TensorFlow[tf.train.shuffle\_batch](https://tensorflow.google.cn/api_docs/python/tf/train/shuffle_batch)。

### 模型预测

模型的预测部分由inference()添加操作以计算预测的logits的函数构成。模型的这一部分组织如下：

| 图层名称 | 描述 |
| --- | --- |
| conv1 | [tf.nn.conv2d](https://tensorflow.google.cn/api_docs/python/tf/nn/conv2d)和[tf.nn.relu](https://tensorflow.google.cn/api_docs/python/tf/nn/relu)激活。 |
| pool1 | [tf.nn.max\_pool](https://tensorflow.google.cn/api_docs/python/tf/nn/max_pool)。 |
| norm1 | [tf.nn.local\_response\_normalization](https://tensorflow.google.cn/api_docs/python/tf/nn/local_response_normalization)。 |
| conv2 | [tf.nn.conv2d](https://tensorflow.google.cn/api_docs/python/tf/nn/conv2d)和[tf.nn.relu](https://tensorflow.google.cn/api_docs/python/tf/nn/relu)激活。 |
| norm2 | [tf.nn.local\_response\_normalization](https://tensorflow.google.cn/api_docs/python/tf/nn/local_response_normalization)。 |
| pool2 | [tf.nn.max\_pool](https://tensorflow.google.cn/api_docs/python/tf/nn/max_pool)。 |
| local3 | [完全连接的层，具有整流的线性激活](https://tensorflow.google.cn/api_guides/python/nn)。 |
| local4 | [完全连接的层，具有整流的线性激活](https://tensorflow.google.cn/api_guides/python/nn)。 |
| softmax\_linear | 线性转换以产生logits。 |

这是从TensorBoard生成的图，描述了推理操作：

**练习**：输出inference是非规范化的logits。尝试编辑网络体系结构以使用返回标准化预测[tf.nn.softmax](https://tensorflow.google.cn/api_docs/python/tf/nn/softmax)。

这些inputs()和inference()函数提供了执行模型评估所需的所有组件。我们现在将重点转向建立模型培训操作。

**练习：**模型架构inference()与[cuda-convnet中](https://code.google.com/p/cuda-convnet/)指定的CIFAR-10模型略有不同 。特别是，Alex的原始模型的顶层是本地连接的，并没有完全连接。尝试编辑体系结构以准确再现顶层中的本地连接体系结构。

### 模特训练

训练网络执行N路分类的常用方法是 [多项逻辑回归](https://en.wikipedia.org/wiki/Multinomial_logistic_regression)，也就是说。softmax回归。Softmax回归将[tf.nn.softmax](https://tensorflow.google.cn/api_docs/python/tf/nn/softmax)非线性应用于 网络的输出，并计算[tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits](https://tensorflow.google.cn/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits) 归一化预测和标签索引之间的差异。对于正规化，我们还将通常的 [tf.nn.l2\_loss](https://tensorflow.google.cn/api_docs/python/tf/nn/l2_loss)损失应用于所有学习变量。模型的目标函数是交叉熵损失和loss()函数返回的所有这些权重衰减项的总和。

我们在TensorBoard中将其可视化为[tf.summary.scalar](https://tensorflow.google.cn/api_docs/python/tf/summary/scalar)：

我们使用标准[梯度下降](https://en.wikipedia.org/wiki/Gradient_descent) 算法[训练](https://tensorflow.google.cn/api_guides/python/train)模型 （参见其他方法的[训练](https://tensorflow.google.cn/api_guides/python/train)），学习率[tf.train.exponential\_decay](https://tensorflow.google.cn/api_docs/python/tf/train/exponential_decay) 随着时间的推移而变化。

该train()函数通过计算梯度和更新学习变量来添加最小化目标所需的操作（[tf.train.GradientDescentOptimizer](https://tensorflow.google.cn/api_docs/python/tf/train/GradientDescentOptimizer) 有关详细信息，请参阅 参考资料）。它返回一个操作，该操作执行为一批图像训练和更新模型所需的所有计算。

## 启动和培训模型

我们已经构建了模型，现在让我们启动它并使用脚本运行训练操作cifar10\_train.py。

python cifar10\_train.py

**注意：**第一次在CIFAR-10教程中运行任何目标时，将自动下载CIFAR-10数据集。数据集约为160MB，因此您可能需要在第一次运行时获取一杯速溶咖啡。

你应该看到输出：

Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.  
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)  
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)  
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)  
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)  
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)  
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)  
...

该脚本报告每10个步骤的总损失以及处理最后一批数据的速度。一些评论：

* 第一批数据可能非常慢（例如几分钟），因为预处理线程用20,000个处理过的CIFAR图像填充了混洗队列。
* 报告的损失是最近一批的平均损失。请记住，这种损失是交叉熵和所有权重衰减项的总和。
* 密切关注批次的处理速度。上面显示的数字是在Tesla K40c上获得的。如果您在CPU上运行，则期望性能降低。

**练习：**在进行实验时，第一次训练步骤可能需要很长时间，这有时令人讨厌。尝试减少最初填满队列的图像数量。搜索min\_fraction\_of\_examples\_in\_queue 在cifar10\_input.py。

cifar10\_train.py定期使用 [tf.train.Saver](https://tensorflow.google.cn/api_docs/python/tf/train/Saver)，保存所有模型参数的 [检查点文件](https://tensorflow.google.cn/guide/saved_model) ，但它并没有评估模型。将使用检查点文件cifar10\_eval.py来度量预测性能（请参阅下面的[评估模型](https://tensorflow.google.cn/tutorials/images/deep_cnn#evaluating_a_model)）。

如果您按照前面的步骤操作，那么您现在已经开始训练CIFAR-10模型。[恭喜！](https://www.youtube.com/watch?v=9bZkp7q19f0)

返回的终端文本cifar10\_train.py提供了对模型如何训练的最小见解。我们希望在培训期间更深入地了解模型：

* 损失真的在减少还是仅仅是噪音？
* 模型是否提供了合适的图像？
* 渐变，激活和重量是否合理？
* 目前的学习率是多少？

[TensorBoard](https://tensorflow.google.cn/guide/summaries_and_tensorboard)提供此功能，显示cifar10\_train.py通过a 定期导出的数据[tf.summary.FileWriter](https://tensorflow.google.cn/api_docs/python/tf/summary/FileWriter)。

例如，我们可以观察local3在训练过程中特征的激活分布和稀疏程度如何变化：

随着时间的推移，个人损失函数以及总损失特别有趣。然而，由于训练所用的小批量，损失表现出相当大的噪音。在实践中，我们发现除了原始值之外，可视化他们的移动平均线非常有用。了解脚本如何 [tf.train.ExponentialMovingAverage](https://tensorflow.google.cn/api_docs/python/tf/train/ExponentialMovingAverage) 用于此目的。

## 评估模型

现在让我们评估训练模型在保持数据集上的表现。该模型由脚本评估cifar10\_eval.py。它使用该inference()函数构建模型，并在CIFAR-10的评估集中使用所有10,000个图像。它计算精度为1：顶部预测与图像的真实标签匹配的频率。

为了监控模型在培训期间如何改进，评估脚本定期运行由最新的检查点文件创建cifar10\_train.py。

python cifar10\_eval.py

注意不要在同一GPU上运行评估和训练二进制文件，否则可能会耗尽内存。考虑在单独的GPU上运行评估（如果可用），或者在同一GPU上运行评估时暂停训练二进制文件。

你应该看到输出：

2015-11-06 08:30:44.391206: precision @ 1 = 0.860  
...

该脚本仅定期返回精度@ 1 - 在这种情况下，它返回86％的准确性。cifar10\_eval.py还可以导出可在TensorBoard中显示的摘要。这些摘要提供了在评估期间对模型的进一步了解。

训练脚本计算 [tf.train.ExponentialMovingAverage](https://tensorflow.google.cn/api_docs/python/tf/train/ExponentialMovingAverage)所有学习变量。评估脚本用移动平均版本替换所有学习的模型参数。此替换可在评估时提升模型性能。

**练习：**采用平均参数可以通过精度@ 1测量预测性能提高约3％。编辑cifar10\_eval.py不使用模型的平均参数并验证预测性能下降。

## 使用多GPU卡训练模型

现代工作站可能包含多个用于科学计算的GPU。TensorFlow可以利用此环境在多个卡上同时运行训练操作。

以并行，分布式方式培训模型需要协调培训流程。对于以下内容，我们的术语模型副本 是数据子集上的模型训练的一个副本。

天真地使用模型参数的异步更新导致次优的训练性能，因为可以在模型参数的陈旧副本上训练单独的模型副本。相反，采用完全同步更新将与最慢的模型副本一样慢。

在具有多个GPU卡的工作站中，每个GPU将具有相似的速度并包含足够的内存来运行整个CIFAR-10模型。因此，我们选择以下列方式设计我们的培训系统：

* 在每个GPU上放置一个单独的模型副本。
* 通过等待所有GPU完成处理一批数据来同步更新模型参数。

这是这个模型的图表：

请注意，每个GPU都会计算推理以及唯一批量数据的渐变。这种设置有效地允许在GPU上划分更大批量的数据。

此设置要求所有GPU共享模型参数。众所周知的事实是，与GPU之间的数据传输速度非常慢。因此，我们决定在CPU上存储和更新所有模型参数（参见绿色框）。当所有GPU处理新批数据时，一组新的模型参数将传输到GPU。

GPU在运行中同步。所有渐变都从GPU累积并取平均值（参见绿框）。使用在所有模型复制品中平均的梯度更新模型参数。

### 在设备上放置变量和操作

在设备上放置操作和变量需要一些特殊的抽象。

我们需要的第一个抽象是用于计算单个模型副本的推理和渐变的函数。在代码中，我们将这种抽象称为“塔”。我们必须为每个塔设置两个属性：

* 塔内所有操作的唯一名称。 [tf.name\_scope](https://tensorflow.google.cn/api_docs/python/tf/name_scope)通过在范围之前添加此唯一名称。例如，第一塔中的所有操作都在tower\_0例如前面tower\_0/conv1/Conv2D。
* 在塔内运行操作的首选硬件设备。 [tf.device](https://tensorflow.google.cn/api_docs/python/tf/device)指定这个。例如，第一个塔中的所有操作都在device('/device:GPU:0') 范围内，表明它们应该在第一个GPU上运行。

所有变量都固定到CPU并通过访问 [tf.get\_variable](https://tensorflow.google.cn/api_docs/python/tf/get_variable) ，以便在多GPU版本中共享它们。请参阅[共享变量的操作方法](https://tensorflow.google.cn/guide/variables)。

### 在多GPU卡上启动和训练模型

如果您的计算机上安装了多个GPU卡，则可以使用它们通过cifar10\_multi\_gpu\_train.py脚本更快地训练模型。此版本的训练脚本将模型与多个GPU卡并行化。

python cifar10\_multi\_gpu\_train.py --num\_gpus=2

请注意，使用的GPU卡数量默认为1.此外，如果您的计算机上只有1个GPU可用，即使您要求更多，也会对其进行所有计算。

**练习：**默认设置cifar10\_train.py是在批量大小为128 的情况下运行。尝试cifar10\_multi\_gpu\_train.py在批量大小为64的2个GPU上运行并比较训练速度。

## 下一步

如果您现在对开发和培训自己的图像分类系统感兴趣，我们建议您使用本教程并更换组件以解决图像分类问题。

**练习：**下载 [街景房号（SVHN）](http://ufldl.stanford.edu/housenumbers/)数据集。分叉CIFAR-10教程并交换SVHN作为输入数据。尝试调整网络架构以提高预测性能。

# 使用具有急切执行的RNN生成文本

本教程演示了如何使用基于字符的RNN生成文本。我们将使用Andrej Karpathy的[“](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)回归[神经网络的不合理有效性”中](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)的莎士比亚写作数据集。给定来自该数据的一系列字符（“Shakespear”），训练模型以预测序列中的下一个字符（“e”）。通过重复调用模型可以生成更长的文本序列。

**注意：**启用GPU加速以更快地执行此笔记本。在Colab中：运行时>更改运行时类型>硬件加速器> GPU。如果在本地运行，请确保TensorFlow版本> = 1.11.0。

本教程包含使用[tf.keras](https://tensorflow.google.cn/programmers_guide/keras)和[eager执行](https://tensorflow.google.cn/programmers_guide/eager)实现的可运行代码。以下是使用默认设置运行本教程时的示例输出：

QUEENE:  
I had thought thou hadst a Roman; for the oracle,  
Thus by All bids the man against the word,  
Which are so weak of care, by old care done;  
Your children were in your holy love,  
And the precipitation through the bleeding throne.  
  
BISHOP OF ELY:  
Marry, and will, my lord, to weep in such a one were prettiest;  
Yet now I was adopted heir  
Of the world's lamentable day,  
To watch the next way with his father with his face?  
  
ESCALUS:  
The cause why then we are all resolved more sons.  
  
VOLUMNIA:  
O, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, no, it is no sin it should be dead,  
And love and pale as any will to that word.  
  
QUEEN ELIZABETH:  
But how long have I heard the soul for this world,  
And show his hands of life be proved to stand.  
  
PETRUCHIO:  
I say he look'd on, if I must be content  
To stay him from the fatal of our country's bliss.  
His lordship pluck'd from this sentence then for prey,  
And then let us twain, being the moon,  
were she such a case as fills m

虽然有些句子是语法上的，但大多数句子都没有意义。该模型尚未学习单词的含义，但考虑：

* 该模型基于字符。当训练开始时，模型不知道如何拼写英语单词，或者单词甚至是单词文本。
* 输出的结构类似于文本的播放块，通常以说话者名称开头，所有大写字母都类似于数据集。
* 如下所示，模型是针对小批量文本（每个100个字符）进行训练的，并且仍然能够生成具有连贯结构的更长文本序列。

## 建立

### 导入TensorFlow和其他库

import tensorflow as tf  
tf.enable\_eager\_execution()  
  
import numpy as np  
import os  
import time

### 下载莎士比亚数据集

更改以下行以在您自己的数据上运行此代码。

path\_to\_file = tf.keras.utils.get\_file('shakespeare.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

从https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt下载数据

1122304/1115394 [==============================] - 0s 0us / step

### 阅读数据

首先，让我们看一下文字。

text = open(path\_to\_file).read()  
# length of text is the number of characters in it  
print ('Length of text: {} characters'.format(len(text)))

文字长度：1115394个字符

# Take a look at the first 1000 characters in text  
print(text[:1000])

First Citizen:

Before we proceed any further, hear me speak.

All:

Speak, speak.

First Citizen:

You are all resolved rather to die than to famish?

All:

Resolved. resolved.

First Citizen:

First, you know Caius Marcius is chief enemy to the people.

All:

We know't, we know't.

First Citizen:

Let us kill him, and we'll have corn at our own price.

Is't a verdict?

All:

No more talking on't; let it be done: away, away!

Second Citizen:

One word, good citizens.

First Citizen:

We are accounted poor citizens, the patricians good.

What authority surfeits on would relieve us: if they

would yield us but the superfluity, while it were

wholesome, we might guess they relieved us humanely;

but they think we are too dear: the leanness that

afflicts us, the object of our misery, is as an

inventory to particularise their abundance; our

sufferance is a gain to them Let us revenge this with

our pikes, ere we become rakes: for the gods know I

speak this in hunger for bread, not in thirst for revenge.

# The unique characters in the file  
vocab = sorted(set(text))  
print ('{} unique characters'.format(len(vocab)))

65个独特的角色

## 处理文本

### 矢量化文本

在训练之前，我们需要将字符串映射到数字表示。创建两个查找表：一个将字符映射到数字，另一个用于数字到字符。

# Creating a mapping from unique characters to indices  
char2idx = {u:i for i, u in enumerate(vocab)}  
idx2char = np.array(vocab)  
  
text\_as\_int = np.array([char2idx[c] for c in text])

现在我们有一个每个字符的整数表示。请注意，我们将字符映射为从0到的索引len(unique)。

for char,\_ in zip(char2idx, range(20)):  
    print('{:6s} ---> {:4d}'.format(repr(char), char2idx[char]))

'O'---> 27

'Q'---> 29

'我'---> 21

'' ---> 8

“'”---> 5

'N'---> 26

'J'---> 22

'v'---> 60

'：'---> 10

'G'---> 19

'm'---> 51

't'---> 58

'S'---> 31

'x'---> 62

'E'---> 17

'V'---> 34

'我'---> 47

'H'---> 20

'＆'---> 4

'C'---> 15

# Show how the first 13 characters from the text are mapped to integers  
print ('{} ---- characters mapped to int ---- > {}'.format(text[:13], text\_as\_int[:13]))

First Citizen ----映射到int的字符----> [18 47 56 57 58 1 15 47 58 47 64 43 52]

### 预测任务

给定一个字符或一系列字符，最可能的下一个字符是什么？这是我们正在训练模型执行的任务。模型的输入将是一系列字符，我们训练模型以预测输出 - 每个时间步的后续字符。

由于RNN维持一个取决于之前看到的元素的内部状态，给定直到此时计算的所有字符，下一个字符是什么？

### 创建培训示例和目标

将文本划分为训练示例和目标。每个训练示例都包含seq\_length文本中的字符。相应的目标包含相同长度的文本，除了向右移动一个字符。例如，假设seq\_length为4，我们的文本为“Hello”，创建一个训练示例“Hell”，并创建一个目标“ello”。

将文本分成以下块seq\_length+1：

# The maximum length sentence we want for a single input in characters  
seq\_length = 100  
  
# Create training examples / targets  
chunks = tf.data.Dataset.from\_tensor\_slices(text\_as\_int).batch(seq\_length+1, drop\_remainder=True)  
  
for item in chunks.take(5):  
  print(repr(''.join(idx2char[item.numpy()])))

'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '

'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst Citizen:\nFirst, you k'

"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFirst Citizen:\nLet us ki"

"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talking on't; let it be d"

'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citizen:\nWe are accounted poor citi'

接下来，从此块创建输入和目标文本：

def split\_input\_target(chunk):  
    input\_text = chunk[:-1]  
    target\_text = chunk[1:]  
    return input\_text, target\_text  
  
dataset = chunks.map(split\_input\_target)

让我们打印第一个例子的前10个值：

for input\_example, target\_example in  dataset.take(1):  
  print ('Input data: ', repr(''.join(idx2char[input\_example.numpy()])))  
  print ('Target data:', repr(''.join(idx2char[target\_example.numpy()])))

Input data: 'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'

Target data: 'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '

这些矢量的每个索引作为一个时间步骤处理。对于时间步骤0的输入，我们接收映射到数字18的字符并尝试预测映射到数字47的字符。在时间步骤1，做同样的事情，但除了当前字符之外还要考虑前一步骤。

for i, (input\_idx, target\_idx) in enumerate(zip(input\_example[:5], target\_example[:5])):  
    print("Step {:4d}".format(i))  
    print("  input: {} ({:s})".format(input\_idx, repr(idx2char[input\_idx])))  
    print("  expected output: {} ({:s})".format(target\_idx, repr(idx2char[target\_idx])))

第0步

输入：18（'F'）

预期产量：47（'i'）

步骤1

输入：47（'i'）

预期产量：56（'r'）

第2步

输入：56（'r'）

预期产量：57（'s'）

第3步

输入：57（'s'）

预期产量：58（'t'）

第4步

输入：58（'t'）

预期产量：1（''）

### 使用tf.data创建批处理并对其进行混洗

我们使用[tf.data](https://tensorflow.google.cn/guide/datasets)将文本分块。但在将这些数据输入模型之前，我们需要对数据进行混洗并将其打包成批。

# Batch size   
BATCH\_SIZE = 64  
  
# Buffer size to shuffle the dataset  
# (TF data is designed to work with possibly infinite sequences,   
# so it doesn't attempt to shuffle the entire sequence in memory. Instead,   
# it maintains a buffer in which it shuffles elements).  
BUFFER\_SIZE = 10000  
  
dataset = dataset.shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE, drop\_remainder=True)

## 该模型

### 实施模型

使用[模型cubclassing API](https://tensorflow.google.cn/guide/keras)创建模型并根据需要更改它。有三个层用于定义我们的模型：[tf.keras](https://tensorflow.google.cn/api_docs/python/tf/keras)

* [嵌入](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Embedding)层：一个可训练的查找表，它将每个字符的数字映射到具有维度的高维向量embedding\_dim;
* [GRU](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/GRU)层：一种层大小=单位的RNN。（您也可以在此处使用LSTM图层。）
* 带有vocab\_size细胞的[密集](https://tensorflow.google.cn/api_docs/python/tf/keras/layers/Dense)层。

class Model(tf.keras.Model):  
  def \_\_init\_\_(self, vocab\_size, embedding\_dim, units):  
    super(Model, self).\_\_init\_\_()  
    self.units = units  
  
    self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)  
  
    if tf.test.is\_gpu\_available():  
      self.gru = tf.keras.layers.CuDNNGRU(self.units,   
                                          return\_sequences=True,   
                                          recurrent\_initializer='glorot\_uniform',  
                                          stateful=True)  
    else:  
      self.gru = tf.keras.layers.GRU(self.units,   
                                     return\_sequences=True,   
                                     recurrent\_activation='sigmoid',   
                                     recurrent\_initializer='glorot\_uniform',   
                                     stateful=True)  
  
    self.fc = tf.keras.layers.Dense(vocab\_size)  
          
  def call(self, x):  
    embedding = self.embedding(x)  
      
    # output at every time step  
    # output shape == (batch\_size, seq\_length, hidden\_size)   
    output = self.gru(embedding)  
      
    # The dense layer will output predictions for every time\_steps(seq\_length)  
    # output shape after the dense layer == (seq\_length \* batch\_size, vocab\_size)  
    prediction = self.fc(output)  
      
    # states will be used to pass at every step to the model while training  
    return prediction

### 实例化模型，优化器和损失函数

# Length of the vocabulary in chars  
vocab\_size = len(vocab)  
  
# The embedding dimension   
embedding\_dim = 256  
  
# Number of RNN units  
units = 1024  
  
model = Model(vocab\_size, embedding\_dim, units)

我们将使用具有默认参数的[Adam优化器](https://tensorflow.google.cn/api_docs/python/tf/train/AdamOptimizer)和[softmax交叉熵](https://tensorflow.google.cn/api_docs/python/tf/losses/sparse_softmax_cross_entropy)作为损失函数。这种损失函数很重要，因为我们正在训练预测下一个字符，而字符数是一个离散数（类似于分类问题）。

# Using adam optimizer with default arguments  
optimizer = tf.train.AdamOptimizer()  
  
# Using sparse\_softmax\_cross\_entropy so that we don't have to create one-hot vectors  
def loss\_function(real, preds):  
    return tf.losses.sparse\_softmax\_cross\_entropy(labels=real, logits=preds)

### 检查点（基于对象的保存）

使用[tf.train.Checkpoint](https://tensorflow.google.cn/api_docs/python/tf/train/Checkpoint)在几个时期之后保存模型的权重。

# Directory where the checkpoints will be saved  
checkpoint\_dir = './training\_checkpoints'  
# Name of the checkpoint files  
checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")  
# Checkpoint instance  
checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)

### 训练模型

在这里，使用[GradientTape](https://tensorflow.google.cn/api_docs/python/tf/GradientTape)的自定义训练循环。您可以通过阅读[热切的执行指南](https://tensorflow.google.cn/guide/eager)来了解有关此方法的更多信息。

* 首先，用零和shape ==（batch\_size，rnn单位数）初始化模型的隐藏状态。我们通过在创建模型时调用定义的函数来完成此操作。
* 接下来，迭代数据集（逐批）并计算与该输入关联的预测和隐藏状态。
* 培训期间发生了许多有趣的事情：
  + 模型得到隐藏状态（用0初始化），让我们调用那个H0和第一批输入，让我们调用它I0。
  + 然后模型返回预测P1和H1。
  + 对于下一批输入，模型接收I1和H1。
  + 这里有趣的是我们传递H1给模型，模型I1是如何学习的。从批次到批次学习的上下文包含在隐藏状态中。
  + 继续这样做直到数据集耗尽，然后开始一个新的纪元并重复该过程。
* 计算预测后，使用上面定义的损失函数计算损失。然后计算相对于模型变量的损失梯度。
* 最后，在优化器的帮助下使用该apply\_gradients函数向该方向迈出一步。

下面是表示上述过程的图表：

model.build(tf.TensorShape([BATCH\_SIZE, seq\_length]))

model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

图层（类型）输出形状参数＃

================================================== ===============

嵌入（嵌入）多个16640

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

gru（GRU）multiple 3935232

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

密集（密集）多个66625

================================================== ===============

总参数：4,018,497

可训练的参数：4,018,497

不可训练的参数：0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Training step  
EPOCHS = 30  
  
for epoch in range(EPOCHS):  
    start = time.time()  
      
    # initializing the hidden state at the start of every epoch  
    # initally hidden is None  
    hidden = model.reset\_states()  
      
    for (batch, (inp, target)) in enumerate(dataset):  
          with tf.GradientTape() as tape:  
              # feeding the hidden state back into the model  
              # This is the interesting step  
              predictions = model(inp)  
              loss = loss\_function(target, predictions)  
                
          grads = tape.gradient(loss, model.variables)  
          optimizer.apply\_gradients(zip(grads, model.variables))  
  
          if batch % 100 == 0:  
              print ('Epoch {} Batch {} Loss {:.4f}'.format(epoch+1,  
                                                            batch,  
                                                            loss))  
    # saving (checkpoint) the model every 5 epochs  
    if (epoch + 1) % 5 == 0:  
      checkpoint.save(file\_prefix = checkpoint\_prefix)  
  
    print ('Epoch {} Loss {:.4f}'.format(epoch+1, loss))  
    print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

Epoch 1批次0损失4.1750

Epoch 1 Batch 100 Loss 2.3517

Epoch 1损失2.1793

所需的时间为1纪元774.1733202934265秒

Epoch 2 Batch 0 Loss 2.1502

Epoch 2 Batch 100 Loss 1.9351

大纪元2损失1.8191

所需时间为1纪元766.1733911037445秒

Epoch 3批次0损失1.8099

Epoch 3 Batch 100 Loss 1.6994

大纪元3损失1.6409

历时762.7873966693878秒

Epoch 4 Batch 0 Loss 1.6119

Epoch 4 Batch 100 Loss 1.5967

Epoch 4损失1.5099

所需的时间为1纪元765.6999187469482秒

Epoch 5 Batch 0 Loss 1.4498

Epoch 5 Batch 100 Loss 1.4723

Epoch 5损失1.4599

所需的时间为1纪录752.7974932193756秒

Epoch 6 Batch 0 Loss 1.4366

Epoch 6 Batch 100 Loss 1.4062

Epoch 6损失1.3669

所需时间为1纪录754.976331949234秒

Epoch 7 Batch 0 Loss 1.3398

Epoch 7 Batch 100 Loss 1.4046

Epoch 7损失1.3731

所需的时间为1纪录758.0927352905273秒

Epoch 8 Batch 0 Loss 1.2517

Epoch 8 Batch 100 Loss 1.3586

Epoch 8损失1.3172

所需时间为1纪录754.760913848877秒

Epoch 9 Batch 0 Loss 1.2488

Epoch 9 Batch 100 Loss 1.3058

Epoch 9损失1.2892

所需的时间为1纪元777.5504937171936秒

Epoch 10 Batch 0 Loss 1.1952

Epoch 10 Batch 100 Loss 1.2515

大纪元10损失1.2856

所需的时间为1纪元788.7329194545746秒

Epoch 11批次0损失1.1727

Epoch 11 Batch 100 Loss 1.2412

Epoch 11损失1.2362

所需的时间为1纪录784.0863444805145秒

Epoch 12 Batch 0 Loss 1.1200

Epoch 12 Batch 100 Loss 1.1896

Epoch 12损失1.2063

所需的时间为1纪元778.597975730896秒

Epoch 13 Batch 0 Loss 1.1069

Epoch 13 Batch 100 Loss 1.1846

Epoch 13损失1.1936

所需的时间为1纪元766.9924449920654秒

Epoch 14批次0损失1.0375

Epoch 14 Batch 100 Loss 1.1656

Epoch 14损失1.1369

所需的时间为1纪元765.9739792346954秒

Epoch 15批次0损失1.0096

Epoch 15 Batch 100 Loss 1.1050

Epoch 15损失1.1035

所需时间为1纪元759.5643520355225秒

Epoch 16 Batch 0 Loss 0.9848

Epoch 16 Batch 100 Loss 1.0595

Epoch 16损失1.0582

所需时间为1纪元756.1685025691986秒

Epoch 17批次0损失0.9424

Epoch 17 Batch 100 Loss 0.9984

Epoch 17损失1.0181

所需时间为1纪元780.668380022049秒

Epoch 18 Batch 0 Loss 0.8714

Epoch 18 Batch 100 Loss 0.9500

Epoch 18损失1.0319

所需的时间为1个纪元775.13188123703秒

Epoch 19 Batch 0 Loss 0.8481

Epoch 19 Batch 100 Loss 0.9382

Epoch 19损失0.9522

所需的时间为1个纪元764.6832077503204秒

Epoch 20 Batch 0 Loss 0.8036

Epoch 20 Batch 100 Loss 0.9106

Epoch 20 Loss 0.9336

所需的时间为1纪元761.5604221820831秒

Epoch 21批次0损失0.7523

Epoch 21 Batch 100 Loss 0.8460

Epoch 21损失0.9178

所需的时间为1纪元913.5510909557343秒

Epoch 22批次0损失0.7142

Epoch 22 Batch 100 Loss 0.8129

Epoch 22损失0.8635

历时792.1691782474518秒

Epoch 23批次0损失0.6787

Epoch 23 Batch 100 Loss 0.7924

Epoch 23损失0.8688

所需的时间为1纪元1799.7459905147552秒

Epoch 24批次0损失0.6637

Epoch 24 Batch 100 Loss 0.7601

Epoch 24损失0.8224

所需的时间为1个纪元1945.9425377845764秒

Epoch 25 Batch 0 Loss 0.6215

Epoch 25 Batch 100 Loss 0.7543

Epoch 25损失0.7729

所需时间为1纪元753.2542717456818秒

Epoch 26 Batch 0 Loss 0.6098

Epoch 26 Batch 100 Loss 0.7236

Epoch 26损失0.7626

所需时间为1个时期750.8798377513885秒

Epoch 27批次0损失0.5824

Epoch 27 Batch 100 Loss 0.6841

大纪元27损失0.7852

所需时间为1纪录754.4956197738647秒

Epoch 28 Batch 0 Loss 0.5737

Epoch 28 Batch 100 Loss 0.6675

大纪元28损失0.7347

所需时间为1纪元1589.552562713623秒

Epoch 29批次0损失0.5481

Epoch 29 Batch 100 Loss 0.7012

大纪元29损失0.7248

所需的时间为1纪元2138.2978603839874秒

Epoch 30 Batch 0 Loss 0.5630

Epoch 30 Batch 100 Loss 0.6865

Epoch 30损失0.7099

所需的时间为1个纪元1378.361043214798秒

checkpoint.save(file\_prefix = checkpoint\_prefix)

” ./training\_checkpoints/ckpt-7'

### 恢复最新的检查点

该模型仅接受固定的批量大小。要使用相同的权重和不同的模型，我们需要重建模型并从检查点恢复权重。

!ls {checkpoint\_dir}

checkpoint ckpt-4.index

ckpt-1.data-00000-of-00001 ckpt-5.data-00000-of-00001

ckpt-1.index ckpt-5.index

ckpt-2.data-00000-of-00001 ckpt-6.data-00000-of-00001

ckpt-2.index ckpt-6.index

ckpt-3.data-00000-of-00001 ckpt-7.data-00000-of-00001

ckpt-3.index ckpt-7.index

CKPT-4.data 00000-的-00001

tf.train.latest\_checkpoint(checkpoint\_dir)

” ./training\_checkpoints/ckpt-7'

model = Model(vocab\_size, embedding\_dim, units)  
  
checkpoint = tf.train.Checkpoint(model=model)  
checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))  
  
model.build(tf.TensorShape([1, None]))

### 使用我们训练有素的模型生成文本

以下代码块生成文本：

* 首先选择一个起始字符串，初始化隐藏状态并设置要生成的字符数。
* 使用起始字符串和隐藏状态获取预测。
* 然后，使用多项分布计算预测字符的索引 - 使用此预测字符作为模型的下一个输入。
* 模型返回的隐藏状态被反馈到模型中，以便它现在具有更多上下文，而不是仅有一个单词。在预测下一个单词之后，修改后的隐藏状态再次被反馈到模型中，这是它从先前预测的单词获得更多上下文时的学习方式。

查看生成的文本，您将看到模型知道何时大写，制作段落并模仿类似莎士比亚的写作风格。

# Evaluation step (generating text using the learned model)  
  
# Number of characters to generate  
num\_generate = 1000  
  
# You can change the start string to experiment  
start\_string = 'Q'  
  
# Converting our start string to numbers (vectorizing)   
input\_eval = [char2idx[s] for s in start\_string]  
input\_eval = tf.expand\_dims(input\_eval, 0)  
  
# Empty string to store our results  
text\_generated = []  
  
# Low temperatures results in more predictable text.  
# Higher temperatures results in more surprising text.  
# Experiment to find the best setting.  
temperature = 1.0  
  
# Here batch size == 1  
model.reset\_states()  
for i in range(num\_generate):  
    predictions = model(input\_eval)  
    # remove the batch dimension  
    predictions = tf.squeeze(predictions, 0)  
  
    # using a multinomial distribution to predict the word returned by the model  
    predictions = predictions / temperature  
    predicted\_id = tf.multinomial(predictions, num\_samples=1)[-1,0].numpy()  
      
    # We pass the predicted word as the next input to the model  
    # along with the previous hidden state  
    input\_eval = tf.expand\_dims([predicted\_id], 0)  
      
    text\_generated.append(idx2char[predicted\_id])  
  
print (start\_string + ''.join(text\_generated))

QUEEN MARGARET:

Which one rosts and rotten, courtesy,

And satisfy froot madamity of Verona's; n your grace.

LUCIO:

Some servenes of a kinderness of his proce.

This tiggh twelve long ago.

PARIS:

O discharger, and it join'd

With sovereigning Richmond, and am I lend.

ROMEO:

Ay, if I know the better doth he start all talk of her observe,

And prefer you for my mishes, call them back.

HORTENSIO:

Gaunt'st as sweet as a cockle,

Repair to the name unto the world,

And see horrors; seain her your honour!

Clown:

Give me thy hand.

The wit is but a head for 't.

BIONDELLO:

I do love to her hence,

And that's mine own,

That, if the least of Maria?

May of Camillo tackless cater?

PETRUCHIO:

Well, then, I thank you.

ISABELLA:

My best and like a press'd truer brother!'

With six on blood to thee,

And he's dead, here are they despair:

See thee utto rise and pluck'd thee by the

hangman. My entertainment,

In pestoly right:

Thy fortune I have many fear'd of mortals

How sall hope is in duty the poor I'll

您还可以尝试使用不同的起始字符，或尝试添加另一个RNN图层以提高模型的准确性，或者调整温度参数以生成或多或少的随机预测。

# 递归神经网络

* [**目录**](https://tensorflow.google.cn/tutorials/sequences/recurrent#top_of_page)
* [介绍](https://tensorflow.google.cn/tutorials/sequences/recurrent#introduction)
* [语言建模](https://tensorflow.google.cn/tutorials/sequences/recurrent#language_modeling)
* [教程文件](https://tensorflow.google.cn/tutorials/sequences/recurrent#tutorial_files)
* [下载并准备数据](https://tensorflow.google.cn/tutorials/sequences/recurrent#download_and_prepare_the_data)

## 介绍

有关 递归神经网络和[LSTM](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)的介绍，请参阅[了解LSTM网络](https://colah.github.io/posts/2015-08-Understanding-LSTMs/)。

## 语言建模

在本教程中，我们将展示如何在具有挑战性的语言建模任务上训练循环神经网络。该问题的目标是拟合概率模型，该概率模型将概率分配给句子。它通过预测文本中的下一个单词来给出前一个单词的历史记录。为此，我们将使用[宾夕法尼亚树银行](https://catalog.ldc.upenn.edu/ldc99t42) （PTB）数据集，这是衡量这些模型质量的流行基准，同时体积小且训练速度相对较快。

语言建模是许多有趣问题的关键，例如语音识别，机器翻译或图像字幕。这也很有趣 - 看看[这里](https://karpathy.github.io/2015/05/21/rnn-effectiveness/)。

出于本教程的目的，我们将重现[Zaremba等人，2014](https://arxiv.org/abs/1409.2329) （[pdf](https://arxiv.org/pdf/1409.2329.pdf)）的结果，该结果 在PTB数据集上实现了非常好的质量。

## 教程文件

本教程是引用了以下文件models/tutorials/rnn/ptb中[TensorFlow机型回购](https://github.com/tensorflow/models)：

| 文件 | 目的 |
| --- | --- |
| ptb\_word\_lm.py | 用于在PTB数据集上训练语言模型的代码。 |
| reader.py | 用于读取数据集的代码。 |

## 下载并准备数据

本教程所需data/的[数据位于Tomas Mikolov网页](http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz)的[PTB数据集](http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz)目录中 。

数据集已经预处理并含有整体10000个不同的词，包括结束句子的标记和用于罕见词语的特殊符号（<UNK>）。在reader.py，我们将每个单词转换为唯一的整数标识符，以便神经网络可以轻松处理数据。

## 该模型

### LSTM

该模型的核心由一个LSTM单元组成，该单元一次处理一个单词并计算句子中下一个单词的可能值的概率。使用零向量初始化网络的内存状态，并在读取每个单词后进行更新。出于计算原因，我们将以小批量处理数据batch\_size。在这个例子中，重要的是要注意current\_batch\_of\_words不对应于单词的“句子”。批处理中的每个单词都应对应于时间t。TensorFlow将自动为您分配每批的梯度。

例如：

 t=0  t=1    t=2  t=3     t=4  
[The, brown, fox, is,     quick]  
[The, red,   fox, jumped, high]  
  
words\_in\_dataset[0] = [The, The]  
words\_in\_dataset[1] = [brown, red]  
words\_in\_dataset[2] = [fox, fox]  
words\_in\_dataset[3] = [is, jumped]  
words\_in\_dataset[4] = [quick, high]  
batch\_size = 2, time\_steps = 5

基本伪代码如下：

words\_in\_dataset = tf.placeholder(tf.float32, [time\_steps, batch\_size, num\_features])  
lstm = tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
# Initial state of the LSTM memory.  
state = lstm.zero\_state(batch\_size, dtype=tf.float32)  
probabilities = []  
loss = 0.0  
for current\_batch\_of\_words in words\_in\_dataset:  
    # The value of state is updated after processing each batch of words.  
    output, state = lstm(current\_batch\_of\_words, state)  
  
    # The LSTM output can be used to make next word predictions  
    logits = tf.matmul(output, softmax\_w) + softmax\_b  
    probabilities.append(tf.nn.softmax(logits))  
    loss += loss\_function(probabilities, target\_words)

### 截断反向传播

通过设计，递归神经网络（RNN）的输出取决于任意远距离的输入。不幸的是，这使得反向传播计算变得困难。为了使学习过程易于处理，通常的做法是创建网络的“展开”版本，其中包含固定数量（num\_steps）的LSTM输入和输出。然后在RNN的这种有限近似上训练该模型。这可以通过一次馈送长度输入num\_steps并在每个这样的输入块之后执行反向传递来实现。

这是一个简化的代码块，用于创建执行截断反向传播的图形：

# Placeholder for the inputs in a given iteration.  
words = tf.placeholder(tf.int32, [batch\_size, num\_steps])  
  
lstm = tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
# Initial state of the LSTM memory.  
initial\_state = state = lstm.zero\_state(batch\_size, dtype=tf.float32)  
  
for i in range(num\_steps):  
    # The value of state is updated after processing each batch of words.  
    output, state = lstm(words[:, i], state)  
  
    # The rest of the code.  
    # ...  
  
final\_state = state

这是如何在整个数据集上实现迭代：

# A numpy array holding the state of LSTM after each batch of words.  
numpy\_state = initial\_state.eval()  
total\_loss = 0.0  
for current\_batch\_of\_words in words\_in\_dataset:  
    numpy\_state, current\_loss = session.run([final\_state, loss],  
        # Initialize the LSTM state from the previous iteration.  
        feed\_dict={initial\_state: numpy\_state, words: current\_batch\_of\_words})  
    total\_loss += current\_loss

### 输入

在输入LSTM之前，单词ID将嵌入到密集表示中（请参阅 [矢量表示教程](https://tensorflow.google.cn/tutorials/representation/word2vec)）。这允许模型有效地表示关于特定单词的知识。写起来也很容易：

# embedding\_matrix is a tensor of shape [vocabulary\_size, embedding size]  
word\_embeddings = tf.nn.embedding\_lookup(embedding\_matrix, word\_ids)

嵌入矩阵将随机初始化，模型将学习通过查看数据来区分单词的含义。

### 损失函数

我们希望最小化目标词的平均负对数概率：

失利=- 1ñΣ一世=1ñLN⁡p目标一世

实现起来并不是很困难，但功能 sequence\_loss\_by\_example已经可用，所以我们可以在这里使用它。

论文中报告的典型指标是平均每个单词的困惑（通常只称为困惑），等于

Ë- 1ñΣ一世=1ñLN⁡p目标一世=Ë失利

我们将在整个培训过程中监控其价值。

### 堆叠多个LSTM

为了使模型具有更强的表现力，我们可以添加多层LSTM来处理数据。第一层的输出将成为第二层的输入，依此类推。

我们有一个叫做的类MultiRNNCell使得实现无缝：

def lstm\_cell():  
  return tf.contrib.rnn.BasicLSTMCell(lstm\_size)  
stacked\_lstm = tf.contrib.rnn.MultiRNNCell(  
    [lstm\_cell() for \_ in range(number\_of\_layers)])  
  
initial\_state = state = stacked\_lstm.zero\_state(batch\_size, tf.float32)  
for i in range(num\_steps):  
    # The value of state is updated after processing each batch of words.  
    output, state = stacked\_lstm(words[:, i], state)  
  
    # The rest of the code.  
    # ...  
  
final\_state = state

## 运行代码

在运行代码之前，请下载PTB数据集，如本教程开头所述。然后，提取主目录下的PTB数据集，如下所示：

tar xvfz simple-examples.tgz -C $HOME

（注意：在Windows上，您可能需要使用 [*其他工具*](https://wiki.haskell.org/How_to_unpack_a_tar_file_in_Windows)。）

现在，克隆 来自GitHub 的[TensorFlow模型repo](https://github.com/tensorflow/models)。运行以下命令：

cd models/tutorials/rnn/ptb  
python ptb\_word\_lm.py --data\_path=$HOME/simple-examples/data/ --model=small

教程代码中有3种支持的模型配置：“小”，“中”和“大”。它们之间的区别在于LSTM的大小和用于训练的超参数集。

模型越大，它应该得到的结果越好。该small模型应该能够在测试装置上达到低于120且在large低于80的情况下达到困惑，尽管可能需要几个小时来训练。

## 接下来是什么？

有一些我们没有提到的技巧可以使模型更好，包括：

* 降低学习率计划，
* LSTM层之间的丢失。

研究代码并对其进行修改以进一步改进模型。

# 用于绘图分类的递归神经网络

[快点，画画！](http://quickdraw.withgoogle.com/)是一个游戏，玩家被挑战绘制一些对象，看看计算机是否可以识别绘图。

在[Quick，Draw中](http://quickdraw.withgoogle.com/)识别[！](http://quickdraw.withgoogle.com/)由获取用户输入的分类器执行，作为x和y中的点的笔划序列给出，并识别用户试图绘制的对象类别。

在本教程中，我们将展示如何为此问题构建基于RNN的识别器。该模型将使用卷积层，LSTM图层和softmax输出图层的组合来对图纸进行分类：

上图显示了我们将在本教程中构建的模型的结构。输入是一个图形，它被编码为x，y和n中点的笔划序列，其中n表示该点是否是新笔划中的第一个点。

然后，应用一系列1维卷积。然后应用LSTM层，并将所有LSTM步骤的输出之和馈送到softmax层，以在我们知道的图纸类别中做出分类决策。

本教程使用来自实际[Quick，Draw](http://quickdraw.withgoogle.com/)的数据[！](http://quickdraw.withgoogle.com/)[公开的](https://quickdraw.withgoogle.com/data)游戏。该数据集包含345个类别中的50M图纸。

## 运行教程代码

要尝试本教程的代码：

1. 如果你还没有[安装TensorFlow](https://tensorflow.google.cn/install)。
2. 下载[教程代码](https://github.com/tensorflow/models/tree/master/tutorials/rnn/quickdraw/train_model.py)。
3. [下载数据](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#download_the_data)中TFRecord的格式 [在这里](http://download.tensorflow.org/data/quickdraw_tutorial_dataset_v1.tar.gz)和解压。有关[如何获取原始Quick，Draw的](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#optional_download_the_full_quick_draw_data)更多细节[！下面提供了数据](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#optional_download_the_full_quick_draw_data)以及[如何将其转换为TFRecord文件](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#optional_converting_the_data)。
4. 使用以下命令执行教程代码以训练本教程中描述的基于RNN的模型。确保调整路径以指向步骤3中下载的解压缩数据。

  python train\_model.py \  
    --training\_data=rnn\_tutorial\_data/training.tfrecord-?????-of-????? \  
    --eval\_data=rnn\_tutorial\_data/eval.tfrecord-?????-of-????? \  
    --classes\_file=rnn\_tutorial\_data/training.tfrecord.classes

## 教程详细信息

### 下载数据

我们将本教程中使用的数据作为TFRecord包含的文件提供TFExamples。您可以从此处下载数据：[http](http://download.tensorflow.org/data/quickdraw_tutorial_dataset_v1.tar.gz)： [//download.tensorflow.org/data/quickdraw\_tutorial\_dataset\_v1.tar.gz](http://download.tensorflow.org/data/quickdraw_tutorial_dataset_v1.tar.gz)（~1GB）。

或者，您可以ndjson从Google云中下载格式化的原始数据，并将其转换为TFRecord包含您TFExamples 自己的文件，如下一节所述。

### 可选：下载完整的快速绘图数据

完整的[快速，抽奖！](https://quickdraw.withgoogle.com/) [数据集](https://quickdraw.withgoogle.com/data)在Google云端存储上可用作按类别分隔的[ndjson](http://ndjson.org/)文件。您可以 [在Cloud Console中浏览文件列表](https://console.cloud.google.com/storage/quickdraw_dataset)。

要下载数据，我们建议使用 [gsutil](https://cloud.google.com/storage/docs/gsutil_install#install)下载整个数据集。请注意，原始.ndjson文件需要下载~22GB。

然后使用以下命令检查您的gsutil安装是否有效以及您是否可以访问数据桶：

gsutil ls -r "gs://quickdraw\_dataset/full/simplified/\*"

这将输出一长串文件，如下所示：

gs://quickdraw\_dataset/full/simplified/The Eiffel Tower.ndjson  
gs://quickdraw\_dataset/full/simplified/The Great Wall of China.ndjson  
gs://quickdraw\_dataset/full/simplified/The Mona Lisa.ndjson  
gs://quickdraw\_dataset/full/simplified/aircraft carrier.ndjson  
...

然后创建一个文件夹并在那里下载数据集。

mkdir rnn\_tutorial\_data  
cd rnn\_tutorial\_data  
gsutil -m cp "gs://quickdraw\_dataset/full/simplified/\*" .

此下载需要一段时间，并下载超过23GB的数据。

### 可选：转换数据

将ndjson文件 转换为包含的[TFRecord](https://tensorflow.google.cn/api_guides/python/python_io#TFRecords_Format_Details)文件 [tf.train.Example](https://tensorflow.google.cn/api_docs/python/tf/train/Example) protos运行以下命令。

   python create\_dataset.py --ndjson\_path rnn\_tutorial\_data \  
      --output\_path rnn\_tutorial\_data

这将数据存储在10个[TFRecord](https://tensorflow.google.cn/api_guides/python/python_io#TFRecords_Format_Details)文件的分 [片中](https://tensorflow.google.cn/api_guides/python/python_io#TFRecords_Format_Details)，每个类有10000个项目用于训练数据，每个类别有1000个项目作为eval数据。

以下更详细地描述该转换过程。

原始QuickDraw数据被格式化为ndjson文件，其中每行包含一个JSON对象，如下所示：

{"word":"cat",  
 "countrycode":"VE",  
 "timestamp":"2017-03-02 23:25:10.07453 UTC",  
 "recognized":true,  
 "key\_id":"5201136883597312",  
 "drawing":[  
   [  
     [130,113,99,109,76,64,55,48,48,51,59,86,133,154,170,203,214,217,215,208,186,176,162,157,132],  
     [72,40,27,79,82,88,100,120,134,152,165,184,189,186,179,152,131,114,100,89,76,0,31,65,70]  
   ],[  
     [76,28,7],  
     [136,128,128]  
   ],[  
     [76,23,0],  
     [160,164,175]  
   ],[  
     [87,52,37],  
     [175,191,204]  
   ],[  
     [174,220,246,251],  
     [134,132,136,139]  
   ],[  
     [175,255],  
     [147,168]  
   ],[  
     [171,208,215],  
     [164,198,210]  
   ],[  
     [130,110,108,111,130,139,139,119],  
     [129,134,137,144,148,144,136,130]  
   ],[  
     [107,106],  
     [96,113]  
   ]  
 ]  
}

为了构建分类器，我们只关心字段“ word”和“ drawing”。在解析ndjson文件时，我们使用一个函数逐行处理它们，该函数将drawing字段中的笔划转换为[number of points, 3]包含连续点差异的大小张量。此函数还将类名作为字符串返回。

def parse\_line(ndjson\_line):  
  """Parse an ndjson line and return ink (as np array) and classname."""  
  sample = json.loads(ndjson\_line)  
  class\_name = sample["word"]  
  inkarray = sample["drawing"]  
  stroke\_lengths = [len(stroke[0]) for stroke in inkarray]  
  total\_points = sum(stroke\_lengths)  
  np\_ink = np.zeros((total\_points, 3), dtype=np.float32)  
  current\_t = 0  
  for stroke in inkarray:  
    for i in [0, 1]:  
      np\_ink[current\_t:(current\_t + len(stroke[0])), i] = stroke[i]  
    current\_t += len(stroke[0])  
    np\_ink[current\_t - 1, 2] = 1  # stroke\_end  
  # Preprocessing.  
  # 1. Size normalization.  
  lower = np.min(np\_ink[:, 0:2], axis=0)  
  upper = np.max(np\_ink[:, 0:2], axis=0)  
  scale = upper - lower  
  scale[scale == 0] = 1  
  np\_ink[:, 0:2] = (np\_ink[:, 0:2] - lower) / scale  
  # 2. Compute deltas.  
  np\_ink = np\_ink[1:, 0:2] - np\_ink[0:-1, 0:2]  
  return np\_ink, class\_name

由于我们希望数据被混洗用于写入，因此我们以随机顺序从每个类别文件中读取并写入随机分片。

对于训练数据，我们读取每个类的前10000个项目，对于eval数据，我们读取每个类别的下1000个项目。

然后将该数据重新格式化为张量形状[num\_training\_samples, max\_length, 3]。然后我们在屏幕坐标中确定原始图形的边界框并标准​​化尺寸，使得图形具有单位高度。

最后，我们计算连续点之间的差异，并将它们存储VarLenFeature在张量 [流中](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto)ink。[示例](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto) 在密钥下面。此外，我们存储class\_index作为单个条目 FixedLengthFeature和所述shape的ink作为FixedLengthFeature长度为2。

### 定义模型

为了定义模型，我们创建了一个新模型Estimator。如果您想了解更多关于估算器的信息，我们建议您使用[本教程](https://tensorflow.google.cn/guide/custom_estimators)。

为了构建模型，我们：

1. 将输入重新整形为原始形状 - 将迷你批次填充到其内容的最大长度。除了墨迹数据，我们还有每个示例和目标类的长度。这发生在函数中[\_get\_input\_tensors](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#_get_input_tensors)。
2. 将输入传递给一系列卷积层 [\_add\_conv\_layers](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#_add_conv_layers)。
3. 将卷积的输出传递给一系列双向LSTM层[\_add\_rnn\_layers](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#_add_rnn_layers)。最后，将每个时间步长的输出相加，以使输入具有紧凑的固定长度嵌入。
4. 使用softmax层对此嵌入进行分类 [\_add\_fc\_layers](https://tensorflow.google.cn/tutorials/sequences/recurrent_quickdraw#_add_fc_layers)。

在代码中，这看起来像：

inks, lengths, targets = \_get\_input\_tensors(features, targets)  
convolved = \_add\_conv\_layers(inks)  
final\_state = \_add\_rnn\_layers(convolved, lengths)  
logits =\_add\_fc\_layers(final\_state)

### \_get\_input\_tensors

为了获得输入特征，我们首先从特征dict获得形状，然后创建[batch\_size]包含输入序列长度的1D张量。墨水作为SparseTensor存储在特征字典中，我们将其转换为密集张量然后重新形成[batch\_size, ?, 3]。最后，如果目标被传递，我们确保它们存储为1D张量的大小[batch\_size]

在代码中，这看起来像这样：

shapes = features["shape"]  
lengths = tf.squeeze(  
    tf.slice(shapes, begin=[0, 0], size=[params["batch\_size"], 1]))  
inks = tf.reshape(  
    tf.sparse\_tensor\_to\_dense(features["ink"]),  
    [params["batch\_size"], -1, 3])  
if targets is not None:  
  targets = tf.squeeze(targets)

### \_add\_conv\_layers

卷积层的期望数量与所述过滤器的长度是通过参数来配置num\_conv和conv\_len在params 字典。

输入是一个序列，其中每个点都具有维数3.我们将使用1D卷积，其中我们将3个输入要素视为通道。这意味着输入是[batch\_size, length, 3]张量，输出将是 [batch\_size, length, number\_of\_filters]张量。

convolved = inks  
for i in range(len(params.num\_conv)):  
  convolved\_input = convolved  
  if params.batch\_norm:  
    convolved\_input = tf.layers.batch\_normalization(  
        convolved\_input,  
        training=(mode == tf.estimator.ModeKeys.TRAIN))  
  # Add dropout layer if enabled and not first convolution layer.  
  if i > 0 and params.dropout:  
    convolved\_input = tf.layers.dropout(  
        convolved\_input,  
        rate=params.dropout,  
        training=(mode == tf.estimator.ModeKeys.TRAIN))  
  convolved = tf.layers.conv1d(  
      convolved\_input,  
      filters=params.num\_conv[i],  
      kernel\_size=params.conv\_len[i],  
      activation=None,  
      strides=1,  
      padding="same",  
      name="conv1d\_%d" % i)  
return convolved, lengths

### \_add\_rnn\_layers

我们将卷积的输出传递给双向LSTM层，我们使用contrib的辅助函数。

outputs, \_, \_ = contrib\_rnn.stack\_bidirectional\_dynamic\_rnn(  
    cells\_fw=[cell(params.num\_nodes) for \_ in range(params.num\_layers)],  
    cells\_bw=[cell(params.num\_nodes) for \_ in range(params.num\_layers)],  
    inputs=convolved,  
    sequence\_length=lengths,  
    dtype=tf.float32,  
    scope="rnn\_classification")

请参阅代码以获取更多详细信息以及如何使用CUDA加速实现。

为了创建紧凑的固定长度嵌入，我们总结了LSTM的输出。我们首先将批处理中没有数据的区域归零。

mask = tf.tile(  
    tf.expand\_dims(tf.sequence\_mask(lengths, tf.shape(outputs)[1]), 2),  
    [1, 1, tf.shape(outputs)[2]])  
zero\_outside = tf.where(mask, outputs, tf.zeros\_like(outputs))  
outputs = tf.reduce\_sum(zero\_outside, axis=1)

### \_add\_fc\_layers

输入的嵌入被传递到完全连接的层，然后我们将其用作softmax层。

tf.layers.dense(final\_state, params.num\_classes)

### 损失，预测和优化

最后，我们需要添加一个损失，一个训练操作和预测来创建 ModelFn：

cross\_entropy = tf.reduce\_mean(  
    tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(  
        labels=targets, logits=logits))  
# Add the optimizer.  
train\_op = tf.contrib.layers.optimize\_loss(  
    loss=cross\_entropy,  
    global\_step=tf.train.get\_global\_step(),  
    learning\_rate=params.learning\_rate,  
    optimizer="Adam",  
    # some gradient clipping stabilizes training in the beginning.  
    clip\_gradients=params.gradient\_clipping\_norm,  
    summaries=["learning\_rate", "loss", "gradients", "gradient\_norm"])  
predictions = tf.argmax(logits, axis=1)  
return model\_fn\_lib.ModelFnOps(  
    mode=mode,  
    predictions={"logits": logits,  
                 "predictions": predictions},  
    loss=cross\_entropy,  
    train\_op=train\_op,  
    eval\_metric\_ops={"accuracy": tf.metrics.accuracy(targets, predictions)})

### 培训和评估模型

为了训练和评估模型，我们可以依赖EstimatorAPI 的功能，并使用 API轻松运行培训和评估Experiment ：

  estimator = tf.estimator.Estimator(  
      model\_fn=model\_fn,  
      model\_dir=output\_dir,  
      config=config,  
      params=model\_params)  
  # Train the model.  
  tf.contrib.learn.Experiment(  
      estimator=estimator,  
      train\_input\_fn=get\_input\_fn(  
          mode=tf.contrib.learn.ModeKeys.TRAIN,  
          tfrecord\_pattern=FLAGS.training\_data,  
          batch\_size=FLAGS.batch\_size),  
      train\_steps=FLAGS.steps,  
      eval\_input\_fn=get\_input\_fn(  
          mode=tf.contrib.learn.ModeKeys.EVAL,  
          tfrecord\_pattern=FLAGS.eval\_data,  
          batch\_size=FLAGS.batch\_size),  
      min\_eval\_frequency=1000)

请注意，本教程只是一个相对较小的数据集的快速示例，可让您熟悉递归神经网络和估算器的API。如果您在大型数据集上尝试它们，这些模型可能会更强大。

当训练模型进行1M步骤时，您可以期望在前1名候选人上获得大约70％的准确度。请注意，这种准确性足以构建快速抽奖游戏，因为游戏动态可以让用户在准备好之前调整他们的绘图。此外，如果目标类别显示的分数优于固定阈值，则游戏不会仅使用前1候选者，而是接受绘图正确。

# 简单的音频识别

本教程将向您展示如何构建一个识别十个不同单词的基本语音识别网络。重要的是要知道真正的语音和音频识别系统要复杂得多，但像图像的MNIST一样，它应该让您对所涉及的技术有基本的了解。完成本教程后，您将拥有一个模型，尝试将一秒音频剪辑分类为静音，未知单词，“是”，“否”，“向上”，“向下”，“向左” ，“正确”，“开启”，“关闭”，“停止”或“开始”。您还可以使用此模型并在Android应用程序中运行它。

## 制备

您应该确保安装了TensorFlow，并且由于脚本下载超过1GB的训练数据，您需要良好的互联网连接和足够的可用空间。培训过程本身可能需要几个小时，因此请确保您有一台可用的机器。

## 训练

要开始培训过程，请转到TensorFlow源代码树并运行：

python tensorflow/examples/speech\_commands/train.py

该脚本将从下载[Speech Commands数据集开始](https://storage.cloud.google.com/download.tensorflow.org/data/speech_commands_v0.02.tar.gz)，该[数据集](https://storage.cloud.google.com/download.tensorflow.org/data/speech_commands_v0.02.tar.gz)包含超过105,000个WAVE音频文件的人们说三十个不同的单词。这些数据由Google收集并在CC BY许可下发布，您可以通过[贡献五分钟自己的声音](https://aiyprojects.withgoogle.com/open_speech_recording)来帮助改进它。存档超过2GB，因此这部分可能需要一段时间，但您应该看到进度日志，一旦下载，您将不再需要执行此步骤。您可以在此[语音命令文章中](https://arxiv.org/abs/1804.03209)找到有关此数据集的更多信息 。

下载完成后，您将看到如下所示的日志记录信息：

I0730 16:53:44.766740   55030 train.py:176] Training from step: 1  
I0730 16:53:47.289078   55030 train.py:217] Step #1: rate 0.001000, accuracy 7.0%, cross entropy 2.611571

这表明初始化过程已经完成，训练循环已经开始。您将看到它为每个训练步骤输出信息。这是对它意味着什么的细分：

Step #1表明我们正处于训练循环的第一步。在这种情况下，总共将有18,000个步骤，因此您可以查看步骤编号，以了解它与完成的接近程度。

rate 0.001000是控制网络权重更新速度的学习率。早期这是一个相对较高的数字（0.001），但对于以后的训练周期，它将减少10倍，到0.0001。

accuracy 7.0%是在这个训练步骤中正确预测了多少个班级。这个值经常会波动很多，但随着培训的进行，平均值会增加。模型输出一个数字数组，每个标签一个，每个数字是输入的预测可能性。通过选择具有最高分数的条目来挑选预测标签。分数始终在0和1之间，较高的值表示对结果更有信心。

cross entropy 2.611571是我们用来指导培训过程的损失函数的结果。这是通过将当前训练运行的得分矢量与正确标签进行比较而获得的得分，并且这应该在训练期间向下趋势。

经过一百步后，您应该看到如下所示的一行：

I0730 16:54:41.813438 55030 train.py:252] Saving to "/tmp/speech\_commands\_train/conv.ckpt-100"

这样可以将当前训练的权重保存到检查点文件中。如果您的训练脚本被中断，您可以查找上次保存的检查点，然后--start\_checkpoint=/tmp/speech\_commands\_train/conv.ckpt-100使用命令行参数重新启动脚本以 从该点开始。

## 混乱矩阵

经过四百步后，将记录此信息：

I0730 16:57:38.073667   55030 train.py:243] Confusion Matrix:  
 [[258   0   0   0   0   0   0   0   0   0   0   0]  
 [  7   6  26  94   7  49   1  15  40   2   0  11]  
 [ 10   1 107  80  13  22   0  13  10   1   0   4]  
 [  1   3  16 163   6  48   0   5  10   1   0  17]  
 [ 15   1  17 114  55  13   0   9  22   5   0   9]  
 [  1   1   6  97   3  87   1  12  46   0   0  10]  
 [  8   6  86  84  13  24   1   9   9   1   0   6]  
 [  9   3  32 112   9  26   1  36  19   0   0   9]  
 [  8   2  12  94   9  52   0   6  72   0   0   2]  
 [ 16   1  39  74  29  42   0   6  37   9   0   3]  
 [ 15   6  17  71  50  37   0   6  32   2   1   9]  
 [ 11   1   6 151   5  42   0   8  16   0   0  20]]

第一部分是[混淆矩阵](https://tensorflow.google.cn/api_docs/python/tf/confusion_matrix)。要理解它的含义，首先需要知道正在使用的标签，在这种情况下是“ 沉默 ”，“ 未知 ”，“是”，“否”，“向上”，“向下”，“向左”，“右“，”上“，”关“，”停止“，”去“。每列代表一组预测为每个标签的样本，因此第一列代表预测为静音的所有片段，第二列代表所有预测为未知单词的片段，第三列代表“是”，所以上。

每行用正确的地面真实标签表示剪辑。第一行是所有静音剪辑，第二行是未知单词，第三行是“是”等。

这个矩阵比单个精度分数更有用，因为它可以很好地总结网络所犯的错误。在此示例中，您可以看到第一行中的所有条目都是零，除了初始条目。因为第一行是所有实际上是静音的片段，这意味着它们都没有被错误地标记为单词，所以我们没有任何假阴性的沉默。这表明网络已经非常善于区分沉默与文字。

如果我们向下看第一列，我们会看到很多非零值。该列表示预测为静音的所有剪辑，因此第一个单元格之外的正数是错误。这意味着实际口语单词的某些片段实际上被预测为沉默，所以我们确实有很多误报。

一个完美的模型会产生一个混淆矩阵，其中所有条目与穿过中心的对角线相差零。发现与该模式的偏差可以帮助您弄清楚模型最容易混淆的方式，一旦确定了问题，您就可以通过添加更多数据或清理类别来解决这些问题。

## 验证

在混淆矩阵之后，您应该看到如下所示的行：

I0730 16:57:38.073777 55030 train.py:245] Step 400: Validation accuracy = 26.3% (N=3093)

将数据集分为三类是一种很好的做法。最大的（在这种情况下大约80％的数据）用于训练网络，较小的一组（此处为10％，称为“验证”）保留用于评估训练期间的准确性，另一组（最后一组） 10％，“测试”用于在培训完成后评估准确性。

这种分裂的原因是网络总是存在在训练期间开始记忆其输入的危险。通过将验证集保持独立，您可以确保模型使用以前从未见过的数据。测试装置是一个额外的安全措施，可确保您不仅仅是以适合培训和验证集的方式调整模型，而不是更广泛的输入。

训练脚本自动将数据集分为这三个类别，上面的日志记录行显示了在验证集上运行时模型的准确性。理想情况下，这应该非常接近训练准确性。如果训练准确度提高但验证没有增加，则表明过度拟合正在发生，并且您的模型只是学习有关训练片段的内容，而不是更广泛的模式。

## Tensorboard

使用Tensorboard可视化培训进展的一种好方法。默认情况下，脚本将事件保存到/ tmp / retrain\_logs，您可以通过运行以下命令加载这些事件：

tensorboard --logdir /tmp/retrain\_logs

然后在浏览器中导航到[http：// localhost：6006](http://localhost:6006/)，您将看到显示模型进度的图表和图形。

## 训练结束

经过几个小时的培训（取决于您机器的速度），脚本应该完成所有18,000步骤。它将打印出最终的混淆矩阵以及准确度分数，所有这些都在测试集上运行。使用默认设置，您应该看到85％到90％之间的准确度。

由于音频识别在移动设备上特别有用，接下来我们将其导出为易于在这些平台上使用的紧凑格式。为此，请运行以下命令行：

python tensorflow/examples/speech\_commands/freeze.py \  
--start\_checkpoint=/tmp/speech\_commands\_train/conv.ckpt-18000 \  
--output\_file=/tmp/my\_frozen\_graph.pb

创建冻结模型后，您可以使用label\_wav.py 脚本对其进行测试，如下所示：

python tensorflow/examples/speech\_commands/label\_wav.py \  
--graph=/tmp/my\_frozen\_graph.pb \  
--labels=/tmp/speech\_commands\_train/conv\_labels.txt \  
--wav=/tmp/speech\_dataset/left/a5d485dc\_nohash\_0.wav

这应该打印出三个标签：

left (score = 0.81477)  
right (score = 0.14139)  
\_unknown\_ (score = 0.03808)

希望“左”是最高分，因为这是正确的标签，但由于训练是随机的，它可能不是你尝试的第一个文件。尝试使用同一文件夹中的一些其他.wav文件来查看它的效果。

得分在0和1之间，较高的值意味着模型对其预测更有信心。

## 在Android应用程序中运行模型

查看此模型在实际应用程序中的工作方式的最简单方法是下载 [预构建的Android演示应用程序](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#prebuilt-components)并将其安装在手机上。您将看到“TF Speech”出现在您的应用列表中，打开它会显示我们刚刚训练过模型的动作词汇列表，从“是”和“否”开始。一旦您授予应用程序使用麦克风的权限，您应该能够尝试说出这些单词，并在模型识别出其中一个时在UI中突出显示它们。

您也可以自己构建此应用程序，因为它是开源的， [可作为github上TensorFlow存储库的一部分提供](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#building-in-android-studio-using-the-tensorflow-aar-from-jcenter)。默认情况下，它会[从tensorflow.org](http://download.tensorflow.org/models/speech_commands_v0.02.zip)下载[预训练模型](http://download.tensorflow.org/models/speech_commands_v0.02.zip)，但您可以轻松地[将其替换为您自己训练的模型](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android#install-model-files-optional)。如果你这样做，你需要确保在常量[主要SpeechActivity Java源文件](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android/src/org/tensorflow/demo/SpeechActivity.java) 一样SAMPLE\_RATE，并SAMPLE\_DURATION符合你训练时的默认设置所做的任何更改。您还将看到有一个[Java版本的RecognizeCommands模块](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android/src/org/tensorflow/demo/RecognizeCommands.java) 这与本教程中的C ++版本非常相似。如果你已经调整了参数，你也可以在SpeechActivity中更新它们，以获得与服务器测试中相同的结果。

演示应用程序会根据您在冻结图表旁边复制到资产中的标签文本文件自动更新其UI结果列表，这意味着您可以轻松地尝试不同的模型，而无需进行任何代码更改。如果更改路径，则需要更新LABEL\_FILENAME并MODEL\_FILENAME指向已添加的文件。

## 这个模型如何运作？

本教程中使用的体系结构基于“ [小型足迹关键字定位的卷积神经网络](http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf) ”一文中所描述的一些体系结构 。选择它是因为它相对简单，快速训练，易于理解，而不是最先进的技术。构建神经网络模型以处理音频有许多不同的方法，包括 [循环网络](https://svds.com/tensorflow-rnn-tutorial/)或[扩张（萎缩）卷积](https://deepmind.com/blog/wavenet-generative-model-raw-audio/)。本教程基于卷积网络的类型，对于使用图像识别的人来说，这种网络会非常熟悉。尽管如此，这可能看起来令人惊讶，因为音频本质上是跨越时间的一维连续信号，而不是2D空间问题。

我们通过定义我们认为我们的口语单词应该适合的时间窗口，并将该窗口中的音频信号转换为图像来解决该问题。这是通过将输入音频样本分组为短片段（仅几毫秒长）并计算一组频带上的频率强度来完成的。来自段的每组频率强度被视为数字向量，并且这些向量按时间顺序排列以形成二维阵列。然后可以将该数组值视为单通道图像，并且称为 [谱图](https://en.wikipedia.org/wiki/Spectrogram)。如果要查看音频样本生成的图像类型，可以运行`wav\_to\_spectrogram工具：

bazel run tensorflow/examples/wav\_to\_spectrogram:wav\_to\_spectrogram -- \  
--input\_wav=/tmp/speech\_dataset/happy/ab00c4b2\_nohash\_0.wav \  
--output\_image=/tmp/spectrogram.png

如果你打开，/tmp/spectrogram.png你应该看到这样的事情：

由于TensorFlow的内存顺序，此图像中的时间从上到下逐渐增加，频率从左到右，与通常的时间从左到右的光谱图惯例不同。您应该能够看到几个不同的部分，第一个音节“Ha”与“ppy”不同。

由于人耳对某些频率比其他人更敏感，因此在语音识别中传统的做法是对该表示进行进一步处理，将其转换为一组[Mel频率倒谱系数](https://en.wikipedia.org/wiki/Mel-frequency_cepstrum)，简称MFCC。这也是一个二维的单通道表示，因此它也可以像图像一样对待。如果您的目标是一般声音而不是语音，您可能会发现可以跳过此步骤并直接在频谱图上进行操作。

然后将由这些处理步骤产生的图像馈送到多层卷积神经网络中，其具有完全连接的层，最后是softmax。您可以在[tensorflow / examples / speech\_commands / models.py中](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py)看到此部分的定义 。

## 流媒体准确性

大多数音频识别应用程序需要在连续的音频流上运行，而不是在单个剪辑上运行。在此环境中使用模型的典型方法是在不同的时间偏移中重复应用它，并在短窗口内平均结果以生成平滑预测。如果您将输入视为图像，则它会沿时间轴不断滚动。我们想要识别的单词可以随时开始，因此我们需要拍摄一系列快照，以便有机会在我们输入模型的时间窗口中捕获大部分话语。如果我们以足够高的速率进行采样，那么我们很有可能在多个窗口中捕获该单词，因此对结果进行平均可以提高预测的整体可信度。

有关如何在流数据上使用模型的示例，您可以查看 [test\_streaming\_accuracy.cc](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/)。这使用[RecognizeCommands](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/recognize_commands.h) 类来运行长格式输入音频，尝试发现单词，并将这些预测与标签和时间的基本事实列表进行比较。这使其成为随着时间的推移将模型应用于音频信号流的一个很好的例子。

你需要一个很长的音频文件来测试它，以及显示每个单词所在位置的标签。如果您不想自己录制，可以使用该generate\_streaming\_test\_wav 实用程序生成一些综合测试数据。默认情况下，这将创建一个十分钟的.wav文件，其中大约每三秒显示一次单词，以及一个文本文件，其中包含说出每个单词时的基本事实。这些单词是从当前数据集的测试部分中提取的，与背景噪声混合在一起。要运行它，请使用：

bazel run tensorflow/examples/speech\_commands:generate\_streaming\_test\_wav

这将保存.wav文件/tmp/speech\_commands\_train/streaming\_test.wav，以及列出标签的文本文件 /tmp/speech\_commands\_train/streaming\_test\_labels.txt。然后，您可以运行准确度测试：

bazel run tensorflow/examples/speech\_commands:test\_streaming\_accuracy -- \  
--graph=/tmp/my\_frozen\_graph.pb \  
--labels=/tmp/speech\_commands\_train/conv\_labels.txt \  
--wav=/tmp/speech\_commands\_train/streaming\_test.wav \  
--ground\_truth=/tmp/speech\_commands\_train/streaming\_test\_labels.txt \  
--verbose

这将输出有关正确匹配的单词数量，给出错误标签的数量以及未说出真实单词时模型触发的次数的信息。有各种参数可以控制信号平均的工作方式，包括--average\_window\_ms设置平均结果的时间长度，--clip\_stride\_ms这是模型应用之间的时间，--suppression\_ms它会阻止后续的单词检测在初始结果之后的一定时间内触发。发现，并且 --detection\_threshold，它控制了在被认为是可靠的结果之前必须达到多高的平均分数。

您将看到流式传输精度输出三个数字，而不仅仅是训练中使用的一个度量标准。这是因为不同的应用程序有不同的要求，只要找到真实的单词（高召回率），有些人能够容忍频繁的错误结果，而其他应用程序非常注重确保预测的标签很可能是正确的，即使有些人不是检测到t（高精度）。该工具中的数字可让您了解模型在应用程序中的性能，您可以尝试调整信号平均参数以对其进行调整，以提供您想要的性能。要了解适用于您的应用程序的正确参数，您可以查看生成[ROC曲线](https://en.wikipedia.org/wiki/Receiver_operating_characteristic)以帮助您理解权衡。

## RecognizeCommands

流精度工具使用一个名为[RecognizeCommands](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/recognize_commands.h)的小型C ++类中包含的简单解码器 。这个类随着时间的推移输入运行TensorFlow模型的输出，它对信号进行平均，并在有足够证据认为已找到已识别的单词时返回有关标签的信息。实现相当小，只是跟踪最后几个预测并对它们进行平均，因此可以根据需要轻松移植到其他平台和语言。例如，在Android上的Java级别或Raspberry Pi上的Python上执行类似的操作很方便。只要这些实现共享相同的逻辑，您就可以使用流测试工具调整控制平均值的参数，然后将它们传输到您的应用程序以获得类似的结果。

## 高级训练

训练脚本的默认设计旨在在相对较小的文件中生成良好的端到端结果，但您可以更改许多选项以根据自己的要求自定义结果。

### 自定义培训数据

默认情况下，脚本将下载[Speech Commands数据集](https://download.tensorflow.org/data/speech_commands_v0.01.tgz)，但您也可以提供自己的训练数据。要训​​练您自己的数据，您应该确保您想要识别的每种声音至少有几百个录音，并按类别将它们排列到文件夹中。例如，如果你试图识别来自cat miaows的狗吠，你会创建一个名为的根文件夹animal\_sounds，然后在那两个名为barkand的子文件夹中miaow。然后，您可以将音频文件组织到相应的文件夹中。

要将脚本指向新的音频文件，您需要设置--data\_url=为禁用语音命令数据集的下载，以及 --data\_dir=/your/data/folder/查找刚刚创建的文件。

文件本身应该是16位小端PCM编码的WAVE格式。采样率默认为16,000，但只要您的所有音频始终保持相同的速率（脚本不支持重新采样），您可以使用--sample\_rate参数更改此值。剪辑也应该大致相同的持续时间。默认的预期持续时间是一秒，但您可以使用--clip\_duration\_ms标志设置此值。如果您在开始时有可变静音量的剪辑，您可以查看单词对齐工具来标准化它们（[这里也可以使用快速而肮脏的方法](https://petewarden.com/2017/07/17/a-quick-hack-to-align-single-word-audio-recordings/)）。

需要注意的一个问题是，您的数据集中可能会有非常相似的相同声音重复，如果它们分布在您的训练，验证和测试集中，则可能会产生误导性指标。例如，语音命令集让人们多次重复相同的单词。这些重复中的每一个都可能与其他重复非常接近，因此如果训练过度拟合并且记忆过，那么当它在测试集中看到非常相似的副本时，它可能会表现得不切实际。为了避免这种危险，语音命令尝试确保将具有单个人所说的相同单词的所有剪辑放入同一分区。剪辑根据其文件名的哈希值分配给训练集，测试集或验证集，确保即使在添加新剪辑时分配仍保持稳定，并避免任何训练样本迁移到其他集合中。为了确保所有给定发言者的话都在同一个桶中，在计算赋值时， 在“ nohash ” 之后[，散列函数会](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/input_data.py)忽略文件名中的任何内容。这意味着，如果你有一个像文件名pete\_nohash\_0.wav和 pete\_nohash\_1.wav，他们保证在同一组。

### 未知类

您的应用程序可能会听到训练集中没有的声音，并且您希望模型在这些情况下表明它无法识别噪声。为了帮助网络了解要忽略的声音，您需要提供一些既不属于您的类的音频片段。要执行此操作，您需要创建quack，oink和moo子文件夹，并使用您的用户可能遇到的其他动物的噪音填充它们。--wanted\_words脚本的参数定义了您关心的类，子文件夹名称中提到的所有其他\_unknown\_类将用于在训练期间填充类。Speech Commands数据集在其未知类中有20个单词，包括0到9的数字和随机名称，如“Sheila”。

默认情况下，10％的训练样例是从未知类中挑选出来的，但您可以使用该--unknown\_percentage标志来控制它。增加这一点会使模型不太可能将未知单词误认为是被通缉的单词，但是将其设置得太大可能会适得其反，因为模型可能会认为将所有单词归类为未知是最安全的！

### 背景噪音

即使在环境中发生其他不相关的声音，真实应用程序也必须识别音频。为了构建一个对这种干扰具有鲁棒性的模型，我们需要针对具有类似属性的录制音频进行训练。Speech Commands数据集中的文件是由许多不同环境中的用户在各种设备上捕获的，而不是在工作室中捕获的，因此有助于为培训添加一些真实感。要添加更多，您可以将环境音频的随机片段混合到训练输入中。在语音命令集中，有一个特殊的文件夹\_background\_noise\_，其中包含带有白噪声的长达一分钟的WAVE文件以及机器和日常家庭活动的记录。

这些文件的小片段随机选择，并在训练期间以低音量混合到剪辑中。响度也是随机选择的，并由--background\_volume参数控制为0为静音的比例，1为满音量。并非所有剪辑都添加了背景，因此 --background\_frequency标志控制它们混合的比例。

您自己的应用程序可能在其自身环境中使用不同于这些默认值的背景噪音模式，因此您可以在\_background\_noise\_文件夹中提供自己的音频剪辑。这些应该与主数据集的采样率相同，但持续时间要长得多，以便可以从中选择一组好的随机段。

### 安静

在大多数情况下，您关心的声音会断断续续，因此了解何时没有匹配的音频非常重要。为了支持这一点，有一个特殊的\_silence\_标签，表明模型什么时候没有检测到任何有趣的东西。因为在真实环境中永远不会完全沉默，我们实际上必须提供安静且无关的音频示例。为此，我们重复使用\_background\_noise\_也混合到真实剪辑中的文件夹，拉动音频数据的短片段并将其与基础真实类一起提供\_silence\_。默认情况下，10％的训练数据是这样提供的，但是--silence\_percentage可以用来控制比例。与未知单词一样，将其设置得更高可以加重模型结果，有利于沉默的真正积极，代价是对单词的错误否定，但是过大的比例会导致它陷入总是猜测沉默的陷阱。

### 时间转移

增加背景噪声是以实际方式扭曲训练数据以有效增加数据集大小的一种方式，因此提高整体准确度，而时移是另一种方法。这涉及训练样本数据的随机时间偏移，从而切断开始或结束的一小部分，并用零填充相对的部分。这模拟了训练数据中开始时间的自然变化，并通过--time\_shift\_ms标志控制， 标志默认为100ms。增加此值将提供更多变化，但存在切断音频重要部分的风险。使用[时间拉伸和音高缩放](https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling)来增加数据的相关方法是使用[时间拉伸和音高缩放](https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling)，但这超出了本教程的范围。

## 自定义模型

用于此脚本的默认模型非常大，每个推理需要超过8亿FLOP并使用940,000个权重参数。这在台式机或现代手机上以可用的速度运行，但它涉及太多的计算，以便在资源更有限的设备上以交互速度运行。为了支持这些用例，可以使用以下几种方法：

**low\_latency\_conv** 基于[卷积神经网络中针对小占用率关键字定位的纸张中](http://www.isca-speech.org/archive/interspeech_2015/papers/i15_1478.pdf)描述的'cnn-one-fstride4'拓扑。精度略低于'conv'，但重量参数的数量大致相同，并且只需要1100万FLOP来运行一个预测，使其更快。

要使用此模型，请--model\_architecture=low\_latency\_conv在命令行上指定。您还需要更新训练率和步数，因此完整命令将如下所示：

python tensorflow/examples/speech\_commands/train \  
--model\_architecture=low\_latency\_conv \  
--how\_many\_training\_steps=20000,6000 \  
--learning\_rate=0.01,0.001

这要求脚本训练的学习率为0.01，步长为20,000，然后以小于10倍的速率进行6,000步的微调。

**low\_latency\_svdf** 基于[使用秩约束拓扑文件压缩深度神经网络中](https://static.googleusercontent.com/media/research.google.com/en/pubs/archive/43813.pdf)呈现的[拓扑](https://static.googleusercontent.com/media/research.google.com/en/pubs/archive/43813.pdf)。精度也低于'conv'，但它仅使用大约75万个参数，最重要的是，它允许在测试时优化执行（即，当您在应用程序中实际使用它时），从而产生75万FLOP。

要使用此模型，请--model\_architecture=low\_latency\_svdf在命令行上指定，并更新训练速率和步骤数，因此完整命令将如下所示：

python tensorflow/examples/speech\_commands/train \  
--model\_architecture=low\_latency\_svdf \  
--how\_many\_training\_steps=100000,35000 \  
--learning\_rate=0.01,0.005

请注意，尽管需要比前两种拓扑更多的步骤，但减少的计算次数意味着训练应该花费大约相同的时间，并且最终达到大约85％的准确度。您还可以通过更改SVDF层中的这些参数，相当容易地进一步调整拓扑以实现计算和精度：

* rank - 近似的等级（通常更高，但会导致更多的计算）。
* num\_units - 与其他图层类型类似，指定图层中的节点数（更多节点质量更好，计算更多）。

关于运行时，由于该层允许通过缓存某些内部神经网络激活进行优化，因此在冻结图形时以及在流模式下执行模型时都需要确保使用一致的步幅（例如'clip\_stride\_ms'标志） （例如test\_streaming\_accuracy.cc）。

**要自定义的其他参数** 如果要试验自定义模型，最好先从调整光谱图创建参数开始。这具有将输入图像的大小改变为模型的效果，并且[models.py中](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py)的创建代码 将自动调整计算和权重的数量以适合不同的维度。如果您将输入设置得更小，则模型将需要更少的计算来处理它，因此它可以是一种折衷一些准确性以改善延迟的好方法。的--window\_stride\_ms控制分开的各频率分析样品是从以前的多远。如果增加此值，则在给定的持续时间内将采用更少的样本，并且输入的时间轴将缩小。该--dct\_coefficient\_countflag控制有多少桶用于频率计数，因此减少此值将缩小其他维度的输入。该--window\_size\_ms参数不会影响大小，但会控制用于计算每个样本的频率的区域的宽度。--clip\_duration\_ms如果您正在寻找的声音很短，那么减少训练样本的持续时间 也会有所帮助，因为这也会减少输入的时间维度。您需要确保所有训练数据包含剪辑初始部分中的正确音频。

如果你的问题有一个完全不同的模型，你可能会发现你可以将它插入 [models.py](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/speech_commands/models.py) 并让其余的脚本处理所有的预处理和训练机制。您可以添加一个新子句create\_model，查找您的体系结构的名称，然后调用模型创建函数。该函数给出了谱图输入的大小以及其他模型信息，并且期望创建TensorFlow操作以读取其中并产生输出预测向量，并且占位符来控制辍学率。脚本的其余部分将处理将此模型集成到更大的图形中进行输入计算并应用softmax和损失函数来训练它。

当您调整模型和训练超参数时，一个常见的问题是，由于数值精度问题，非数值可能会蔓延。通常，您可以通过减少学习速率和权重初始化函数等数量来解决这些问题，但如果它们是持久的，您可以启用该--check\_nans标志来追踪错误的来源。这将在TensorFlow中的大多数常规操作之间插入检查操作，并在遇到时使用有用的错误消息中止训练过程。

# 神经机器翻译（seq2seq）教程

作者：Thang Luong，Eugene Brevdo，Rui Zhao（[*Google Research Blogpost*](https://research.googleblog.com/2017/07/building-your-own-neural-machine.html)，[*Github*](https://github.com/tensorflow/nmt)）

此版本的教程需要[*TensorFlow Nightly*](https://github.com/tensorflow/tensorflow/#installation)。要使用稳定的TensorFlow版本，请考虑其他分支，例如 [*tf-1.4*](https://github.com/tensorflow/nmt/tree/tf-1.4)。

## 介绍

序列到序列（seq2seq）模型（[Sutskever等人，2014](https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf)， [Cho等人，2014](http://emnlp2014.org/papers/pdf/EMNLP2014179.pdf)）在诸如机器翻译，语音识别和文本摘要的各种任务中取得了巨大成功。本教程让读者全面了解seq2seq模型，并展示如何从头开始构建具有竞争力的seq2seq模型。我们专注于神经机器翻译（NMT）的任务，这是seq2seq模型的第一个测试平台， [取得了巨大的成功](https://research.googleblog.com/2016/09/a-neural-network-for-machine.html)。所包含的代码是轻量级，高质量，生产就绪，并结合了最新的研究思路。我们实现这一目标：

1. 使用最近的解码器/注意包装器 [API](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/seq2seq/python/ops)，TensorFlow 1.2数据迭代器
2. 结合我们在构建recurrent和seq2seq模型方面的强大专业知识
3. 提供建立最佳NMT模型和复制[Google NMT（GNMT）系统的](https://research.google.com/pubs/pub45610.html)提示和技巧 。

我们认为提供人们可以轻松复制的基准非常重要。因此，我们提供了完整的实验结果，并在以下公开数据集上预先训练了我们的模型：

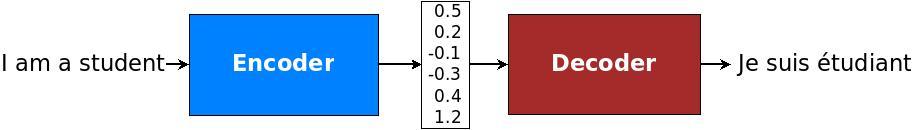
1. 小规模：[IWSLT评估活动](https://sites.google.com/site/iwsltevaluation2015/)提供的英语 - 越南语平行语料库TED讲座（133K句子对） 。
2. 大规模：[WMT评估活动](http://www.statmt.org/wmt16/translation-task.html)提供的德语 - 英语平行语料库（4.5M句子对）。

我们首先建立一些关于NMT的seq2seq模型的基本知识，解释如何构建和训练一个vanilla NMT模型。第二部分将详细介绍建立具有关注机制的竞争性NMT模型。然后，我们讨论了构建最佳NMT模型（速度和翻译质量）的技巧和窍门，例如TensorFlow最佳实践（批处理，分组），双向RNN，波束搜索，以及使用GNMT注意扩展到多个GPU。

## 基本

## 神经机器翻译的背景

回到过去，传统的基于短语的翻译系统通过将源句分成多个块然后逐个词地翻译它们来完成它们的任务。这导致了翻译输出的不流畅，并不像我们人类翻译的那样。我们阅读整个源句，理解其含义，然后产生翻译。神经机器翻译（NMT）模仿了！

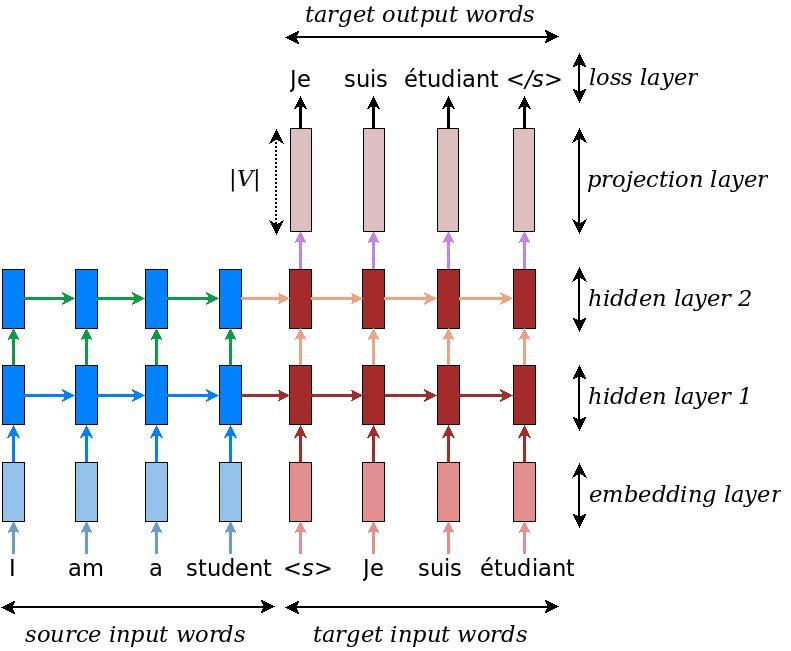
图1. **编码器 - 解码器架构** - NMT的一般方法示例。编码器将源句子转换为“含义”向量，该向量通过*解码器*以产生翻译。

具体地，NMT系统首先使用编码器读取源句子 以构建 [“思想”向量](https://www.theguardian.com/science/2015/may/21/google-a-step-closer-to-developing-machines-with-human-like-intelligence)，即表示句子含义的数字序列; 一个解码器，那么，处理句子矢量以发射平移，如在图1中示出这通常被称为编码器-解码器架构。通过这种方式，NMT解决了传统的基于短语的方法中的本地翻译问题：它可以捕获语言中的长程依赖性，例如性别协议; 语法结构; [谷歌神经机器翻译系统](https://research.googleblog.com/2016/09/a-neural-network-for-machine.html)证明，并提供更流畅的翻译 。

NMT模型的确切架构各不相同。顺序数据的自然选择是大多数NMT模型使用的递归神经网络（RNN）。通常RNN用于编码器和解码器。然而，RNN模型在以下方面有所不同：（a）方向性 - 单向或双向; （b）深度 - 单层或多层; （c）类型 - 通常是香草RNN，长期短期记忆（LSTM）或门控复发单位（GRU）。有兴趣的读者可以在这篇[博客文章中](http://colah.github.io/posts/2015-08-Understanding-LSTMs/)找到有关RNN和LSTM的更多信息。

在本教程中，我们将深层多层RNN视为单向，并使用LSTM作为循环单元。我们在图2中展示了这种模型的一个例子。在这个例子中，我们建立了一个模型，将源句“我是学生”翻译成目标句子“Jesuisétudiant”。在高层次上，NMT模型由两个递归神经网络组成：编码器 RNN简单地消耗输入源字而不进行任何预测; 的 解码器，另一方面，同时预测下一话处理目标句子。

有关更多信息，请参阅本教程所基于的[Luong（2016）](https://github.com/lmthang/thesis)。

  
图2. **神经机器翻译** - 用于将源句“我是学生”翻译成目标句子“Jesuisétudiant”而提出的深度重复体系结构的示例。这里，“<s>”标志着解码过程的开始，而“</ s>”告诉解码器停止。

## 安装教程

要安装本教程，您需要在系统上安装TensorFlow。本教程需要TensorFlow Nightly。要安装TensorFlow，请按照[此处](https://www.tensorflow.org/install/)的[安装说明进行操作](https://www.tensorflow.org/install/)。

安装TensorFlow后，您可以通过运行以下命令下载本教程的源代码：

git clone https://github.com/tensorflow/nmt/

## 培训 - 如何建立我们的第一个NMT系统

让我们首先深入探讨使用具体代码片段构建NMT模型的核心，我们将通过该代码片段更详细地解释图2。我们将数据准备和完整代码推迟到以后。这部分是指文件 [**model.py**](https://github.com/tensorflow/nmt/blob/master/nmt/model.py)。

在底层，编码器和解码器RNN接收以下内容作为输入：首先，源句子，然后是指示从编码到解码模式的转换的边界标记“<s>”和目标句子。对于培训，我们将使用以下张量来提供系统，这些张量是时间主要格式并包含单词索引：

* **encoder\_inputs** [max\_encoder\_time，batch\_size]：源输入字。
* **decoder\_inputs** [max\_decoder\_time，batch\_size]：目标输入字。
* **decoder\_outputs** [max\_decoder\_time，batch\_size]：目标输出字，这些是decoder\_inputs向左移动一个时间步，右边附有一个句末标记。

为了提高效率，我们一次训练多个句子（batch\_size）。测试略有不同，我们稍后会讨论。

### 嵌入

鉴于单词的分类性质，模型必须首先查找源和目标嵌入以检索相应的单词表示。要使此嵌入层起作用，首先为每种语言选择词汇表。通常，选择词汇量V，并且仅将最频繁的V词视为唯一。所有其他单词都转换为“未知”标记，并且都获得相同的嵌入。每种语言一套的嵌入权重通常在训练期间学习。

# Embedding

embedding\_encoder = variable\_scope.get\_variable(

"embedding\_encoder", [src\_vocab\_size, embedding\_size], ...)

# Look up embedding:

# encoder\_inputs: [max\_time, batch\_size]

# encoder\_emb\_inp: [max\_time, batch\_size, embedding\_size]

encoder\_emb\_inp = embedding\_ops.embedding\_lookup(

embedding\_encoder, encoder\_inputs)

同样，我们可以构建*embedding\_decoder*和*decoder\_emb\_inp*。请注意，可以选择使用预训练的单词表示（如word2vec或Glove向量）初始化嵌入权重。通常，在给定大量训练数据的情况下，我们可以从头开始学习这些嵌入。

### 编码器

一旦被检索，然后将字嵌入作为输入馈送到主网络，主网络由两个多层RNN组成 - 源语言的编码器和目标语言的解码器。原则上，这两个RNN可以共享相同的权重; 然而，在实践中，我们经常使用两个不同的RNN参数（这样的模型在拟合大型训练数据集时做得更好）。该 编码器 RNN使用零个向量为出发状态，是建立如下：

# Build RNN cell

encoder\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units)

# Run Dynamic RNN

# encoder\_outputs: [max\_time, batch\_size, num\_units]

# encoder\_state: [batch\_size, num\_units]

encoder\_outputs, encoder\_state = tf.nn.dynamic\_rnn(

encoder\_cell, encoder\_emb\_inp,

sequence\_length=source\_sequence\_length, time\_major=True)

请注意，句子有不同的长度以避免浪费计算，我们通过source\_sequence\_length告诉 dynamic\_rnn确切的源句长度 。由于我们的输入是时间主要的，我们设置 time\_major = True。在这里，我们只构建单层LSTM，encoder\_cell。我们将在后面的部分中描述如何构建多层LSTM，添加dropout和使用注意力。

### 解码器

该*解码器*还需要能够访问源信息，和一个简单的方法来实现这一目标是与编码器，最后隐藏状态初始化它*encoder\_state*。在图2中，我们将源词“student”的隐藏状态传递给解码器端。

# Build RNN cell

decoder\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units)

# Helper

helper = tf.contrib.seq2seq.TrainingHelper(

decoder\_emb\_inp, decoder\_lengths, time\_major=True)

# Decoder

decoder = tf.contrib.seq2seq.BasicDecoder(

decoder\_cell, helper, encoder\_state,

output\_layer=projection\_layer)

# Dynamic decoding

outputs, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder, ...)

logits = outputs.rnn\_output

这里，该代码的核心部分是BasicDecoder对象，解码器，它接收decoder\_cell（类似于encoder\_cell），一个帮助器，以及前一个encoder\_state作为输入。通过分离出解码器和助手，我们可以重用的代码库不同，例如，TrainingHelper可以与取代GreedyEmbeddingHelper做贪心解码。在[helper.py中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/helper.py)查看更多 [信息](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/helper.py)。

最后，我们还没有提到projection\_layer是一个密集矩阵，可以将顶部隐藏状态转换为维度V的logit向量。我们在图2的顶部说明了这个过程。

projection\_layer = layers\_core.Dense（

tgt\_vocab\_size，use\_bias = False）

### 失利

鉴于logits以上，我们现在已经准备好来计算我们的培训损失：

crossent = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(

labels=decoder\_outputs, logits=logits)

train\_loss = (tf.reduce\_sum(crossent \* target\_weights) /

batch\_size)

这里，target\_weights是与decoder\_outputs大小相同的零一矩阵 。它屏蔽目标序列长度之外的填充位置，值为0。

***重要提示***：值得指出的是，我们将损失除以 batch\_size，因此我们的超参数对batch\_size是“不变的”。有些人将损失除以（batch\_size \* num\_time\_steps），这减少了短句中的错误。更巧妙的是，我们的超参数（应用于前一种方式）不能用于后一种方式。例如，如果两种方法都使用学习为1.0的SGD，则后一种方法有效地使用1 / num\_time\_steps的小得多的学习速率。

### 梯度计算和优化

我们现在定义了NMT模型的前向通道。计算反向传播过程只需几行代码：

# Calculate and clip gradients

params = tf.trainable\_variables()

gradients = tf.gradients(train\_loss, params)

clipped\_gradients, \_ = tf.clip\_by\_global\_norm(

gradients, max\_gradient\_norm)

训练RNN的重要步骤之一是梯度削波。在这里，我们按照全球规范进行剪辑。最大值max\_gradient\_norm通常设置为5或1之类的值。最后一步是选择优化程序。Adam优化器是一种常见的选择。我们还选择学习率。learning\_rate的值 通常在0.0001到0.001的范围内; 并且可以随着训练的进行而减少。

# Optimization

optimizer = tf.train.AdamOptimizer(learning\_rate)

update\_step = optimizer.apply\_gradients(

zip(clipped\_gradients, params))

在我们自己的实验中，我们使用标准SGD（tf.train.GradientDescentOptimizer），学习速率计划越来越低，从而产生更好的性能。查看[基准](https://github.com/tensorflow/nmt#benchmarks)。

## 动手 - 让我们训练一个NMT模型

让我们训练我们的第一个NMT模型，从越南语翻译成英语！我们代码的入口点是 [**nmt.py**](https://github.com/tensorflow/nmt/blob/master/nmt/nmt.py)。

我们将使用小规模的并行TED演讲语料库（133K训练样例）进行此练习。我们在这里使用的所有数据都可以在以下[网址](https://nlp.stanford.edu/projects/nmt/)找到：[https](https://nlp.stanford.edu/projects/nmt/)： [//nlp.stanford.edu/projects/nmt/](https://nlp.stanford.edu/projects/nmt/)。我们将使用tst2012作为我们的开发数据集，并使用tst2013作为我们的测试数据集。

运行以下命令下载训练NMT模型的数据：  
nmt/scripts/download\_iwslt15.sh /tmp/nmt\_data

运行以下命令以开始培训：

mkdir / tmp / nmt\_model

python -m nmt.nmt \

--src = vi --tgt = en \

--vocab\_prefix = / tmp / nmt\_data / vocab \

--train\_prefix = / tmp / nmt\_data / train \

--dev\_prefix = / tmp / nmt\_data / tst2012 \

--test\_prefix = / tmp / nmt\_data / tst2013 \

--out\_dir = / tmp / nmt\_model \

--num\_train\_steps = 12000 \

--steps\_per\_stats = 100 \

--num\_layers = 2 \

--num\_units = 128 \

--dropout = 0.2 \

--metrics =bleu

上述命令训练了一个2层LSTM seq2seq模型，该模型具有128个暗淡的隐藏单元和嵌入12个时期。我们使用0.2的丢失值（保持概率0.8）。如果没有错误，我们应该看到类似于下面的日志，我们训练时的复杂度值会降低。

# First evaluation, global step 0

eval dev: perplexity 17193.66

eval test: perplexity 17193.27

# Start epoch 0, step 0, lr 1, Tue Apr 25 23:17:41 2017

sample train data:

src\_reverse: </s> </s> Điều đó , dĩ nhiên , là câu chuyện trích ra từ học thuyết của Karl Marx .

ref: That , of course , was the <unk> distilled from the theories of Karl Marx . </s> </s> </s>

epoch 0 step 100 lr 1 step-time 0.89s wps 5.78K ppl 1568.62 bleu 0.00

epoch 0 step 200 lr 1 step-time 0.94s wps 5.91K ppl 524.11 bleu 0.00

epoch 0 step 300 lr 1 step-time 0.96s wps 5.80K ppl 340.05 bleu 0.00

epoch 0 step 400 lr 1 step-time 1.02s wps 6.06K ppl 277.61 bleu 0.00

epoch 0 step 500 lr 1 step-time 0.95s wps 5.89K ppl 205.85 bleu 0.00

有关详细信息，请参阅[**train.py**](https://github.com/tensorflow/nmt/blob/master/nmt/train.py) .

我们可以启动Tensorboard以在培训期间查看模型的摘要：

tensorboard --port 22222 --logdir / tmp / nmt\_model /

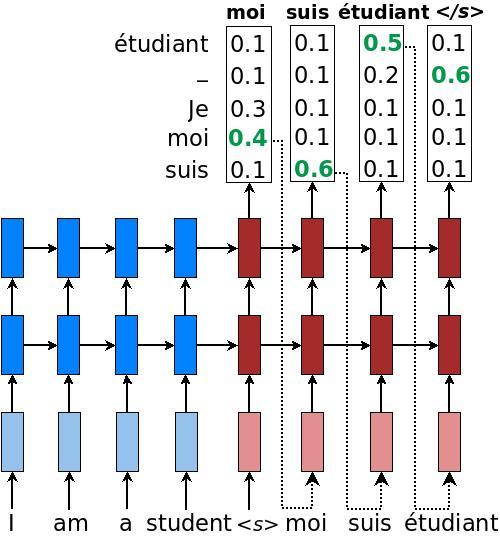
从英语和越南语开始反向训练可以通过以下方式进行：  
--src=en --tgt=vi

## 推理 - 如何生成翻译

在训练你的NMT模型时（一旦你训练了模型），你可以获得以前看不见的源句子的翻译。此过程称为推理。训练和推理（测试）之间有明显的区别：在推理时，我们只能访问源句，即encoder\_inputs。有许多方法可以执行解码。解码方法包括贪婪，采样和波束搜索解码。在这里，我们将讨论贪婪的解码策略。

这个想法很简单，我们在图3中说明了这一点：

1. 我们仍然以与训练期间相同的方式对源句子进行编码以获得encoder\_state，并且该encoder\_state用于初始化解码器。
2. 一旦解码器接收到起始符号“<s>”（在我们的代码中称为tgt\_sos\_id），就开始解码（转换）过程;
3. 对于解码器端的每个时间步，我们将RNN的输出视为一组logits。我们选择最可能的单词，即与最大logit值相关联的id作为发出的单词（这是“贪婪”行为）。例如，在图3中，单词“moi”在第一解码步骤中具有最高的转换概率。然后我们将这个单词作为输入提供给下一个时间步。
4. 该过程一直持续到句子结束标记“</ s>”作为输出符号产生（在我们的代码中称为tgt\_eos\_id）。

   
图3. **贪婪解码** - 一个训练有素的NMT模型如何使用贪婪搜索为源句“Jesuisétudiant”生成翻译的示例。

第3步是推理与培训不同的原因。推理不使用正确的目标词作为输入，而是使用模型预测的词。这是实现贪婪解码的代码。它与训练解码器非常相似。

# Helper

helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(

embedding\_decoder,

tf.fill([batch\_size], tgt\_sos\_id), tgt\_eos\_id)

# Decoder

decoder = tf.contrib.seq2seq.BasicDecoder(

decoder\_cell, helper, encoder\_state,

output\_layer=projection\_layer)

# Dynamic decoding

outputs, \_ = tf.contrib.seq2seq.dynamic\_decode(

decoder, maximum\_iterations=maximum\_iterations)

translations = outputs.sample\_id

在这里，我们使用GreedyEmbeddingHelper而不是TrainingHelper。由于我们事先不知道目标序列长度，因此我们使用maximum\_iterations来限制转换长度。一种启发式方法是解码源语句长度的两倍。

maximum\_iterations = tf.round（tf.reduce\_max（source\_sequence\_length）\* 2）

在训练了模型后，我们现在可以创建推理文件并翻译一些句子：

cat > /tmp/my\_infer\_file.vi

# (copy and paste some sentences from /tmp/nmt\_data/tst2013.vi)

python -m nmt.nmt \

--out\_dir=/tmp/nmt\_model \

--inference\_input\_file=/tmp/my\_infer\_file.vi \

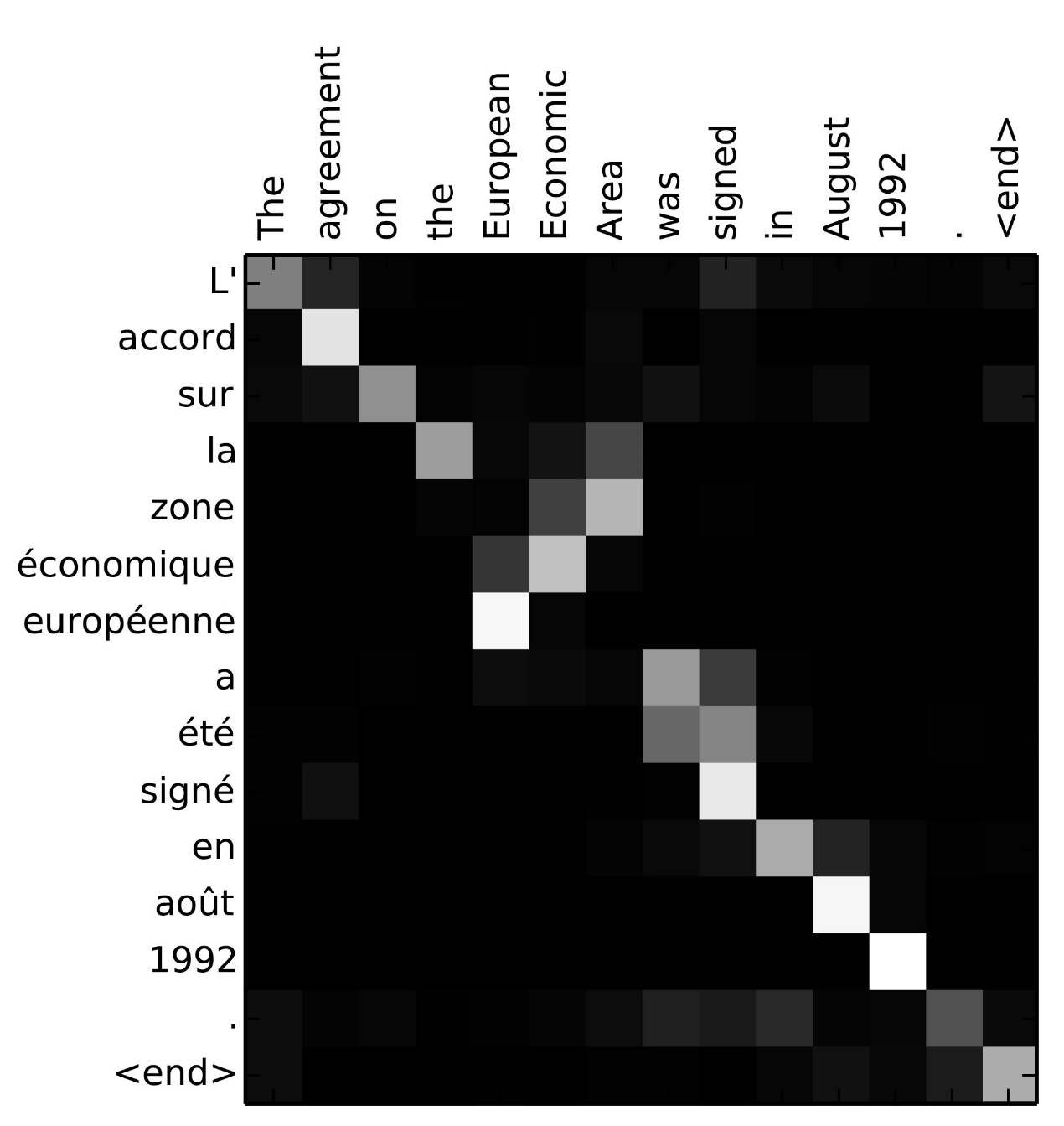
--inference\_output\_file=/tmp/nmt\_model/output\_infer

cat /tmp/nmt\_model/output\_infer # To view the inference as output

注意，只要存在训练检查点，也可以在模型仍在训练时运行上述命令。有关更多详细信息，请参阅[**inference.py**](https://github.com/tensorflow/nmt/blob/master/nmt/inference.py)。

## 中间

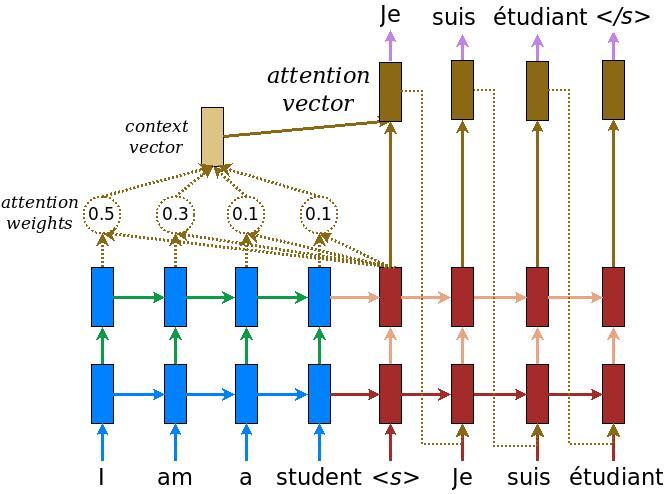
通过最基本的seq2seq模型，让我们变得更先进！为了建立最先进的神经机器翻译系统，我们需要更多的“秘密调料”：注意机制，由[Bahdanau等人](https://arxiv.org/abs/1409.0473)首次引入[，2015年](https://arxiv.org/abs/1409.0473)，后来由[Luong等人，2015年](https://arxiv.org/abs/1508.04025)和其他。关注机制的关键思想是通过在翻译时“关注”相关的源内容，在目标和源之间建立直接的快捷连接。注意机制的一个很好的副产品是源语句和目标句子之间易于可视化的对齐矩阵（如图4所示）。

 图4. **注意可视化** - 源句和目标句之间对齐的示例。图像取自（Bahdanau等，2015）。

请记住，在vanilla seq2seq模型中，我们在开始解码过程时将最后一个源状态从编码器传递到解码器。这适用于短句和中长句; 然而，对于长句，单个固定大小的隐藏状态成为信息瓶颈。注意机制不是丢弃在源RNN中计算的所有隐藏状态，而是提供允许解码器窥视它们的方法（将它们视为源信息的动态存储器）。通过这样做，注意机制改善了较长句子的翻译。如今，注意机制是事实上的标准，并已成功应用于许多其他任务（包括图像标题生成，语音识别和文本摘要）。

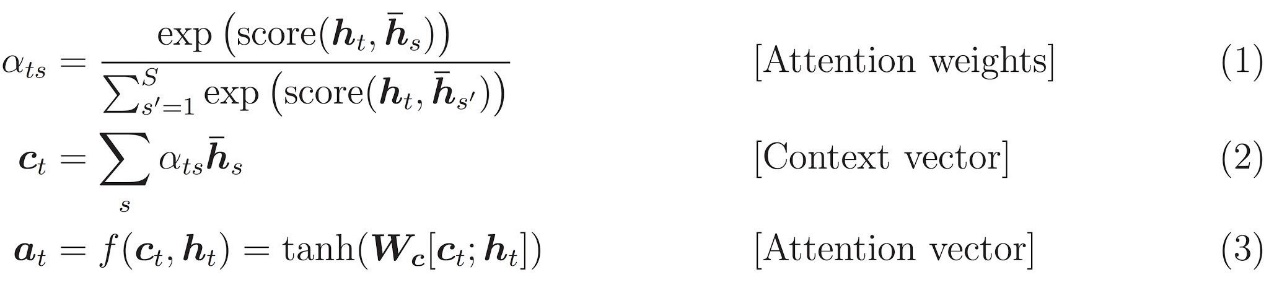
## 关注机制的背景

我们现在描述（Luong et al。，2015）中提出的注意机制的一个实例，该实例已被用于几个最先进的系统，包括开源工具包，如[OpenNMT](http://opennmt.net/about/)和TF seq2seq API，教程。我们还将提供与注意机制的其他变体的连接。

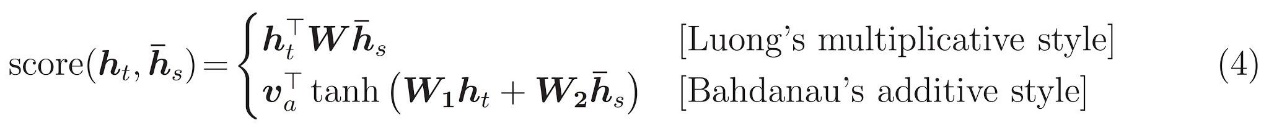
 图5. **注意机制** - 如（Luong等，2015）所述的基于注意力的NMT系统的示例。我们将详细介绍注意力计算的第一步。为清楚起见，我们没有显示图（2）中的嵌入和投影层。

如图5所示，注意力计算在每个解码器时间步骤发生。它包括以下几个阶段：

1. 将当前目标隐藏状态与所有源状态进行比较以得出 注意权重（可以如图4中那样可视化）。
2. 基于注意权重，我们计算上下文向量作为源状态的加权平均值。
3. 将上下文向量与当前目标隐藏状态组合以产生最终关注向量
4. 注意向量作为输入馈送到下一个时间步（输入馈送）。前三个步骤可以通过以下等式汇总：



这里，该函数score用于将目标隐藏状态$$ h\_t $$与每个源隐藏状态$$ \ overline {h} \_s $$进行比较，并将结果标准化为生成的注意权重（源位置上的分布） ）。评分功能有多种选择; 流行的评分函数包括公式中给出的乘法和加法形式。（4）。一旦计算，注意向量$$ a\_t $$用于导出softmax logit和loss。这类似于vanilla seq2seq模型顶层的目标隐藏状态。该功能 f也可以采取其他形式。



注意机制的各种实现可以在[attention\_wrapper.py中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py)找到 。

***注意机制有什么关系？***

如上述等式所示，有许多不同的注意变体。这些变体取决于评分函数和注意函数的形式，以及是否在（Bahdanau）最初建议的评分函数中使用先前状态$$ h\_ {t-1} $$而不是$$ h\_t $$等人，2015）。根据经验，我们发现只有某些选择才有意义。首先，需要存在基本的关注形式，即目标和源之间的直接联系。其次，重要的是将注意力向量提供给下一个时间步，以便向网络通报过去的注意力决定，如（Luong等，2015）所示。最后，评分功能的选择通常会导致不同的表现。在[基准测试结果](https://github.com/tensorflow/nmt#benchmarks) 部分中查看更多信息

## 注意包装器API

在我们的[AttentionWrapper](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/seq2seq/python/ops/attention_wrapper.py)实现中 ，我们从他们的内存网络工作中 借用了一些术语[（Weston et al。，2015）](https://arxiv.org/abs/1410.3916)。本教程中介绍的注意机制不是具有可读和可写内存，而是一个只读内存。具体来说，一组源隐藏状态（或它们的转换版本，例如Luong的得分风格中的$$ W \ overline {h} \_s $$或者Bahdanau的得分风格中的$ W\_2 \ overline {h} \_s $$）被引用作为“记忆”。在每个时间步，我们使用当前目标隐藏状态作为“查询”决定要读取的内存部分。通常，需要将查询与对应于各个存储器槽的键进行比较。在上面的注意机制介绍中，我们碰巧使用源隐藏状态集（或其转换版本，例如Bahdanau评分风格中的$$ W\_1h\_t $$）作为“键”。人们可以从这种记忆网络术语中获得启发，以获得其他形式的关注！

感谢注意包装，延长我们的香草seq2seq代码注意力是微不足道的。这部分引用文件[**attention\_model.py**](https://github.com/tensorflow/nmt/blob/master/nmt/attention_model.py)

首先，我们需要定义一种注意机制，例如（Luong et al。，2015）：

# attention\_states: [batch\_size, max\_time, num\_units]

attention\_states = tf.transpose(encoder\_outputs, [1, 0, 2])

# Create an attention mechanism

attention\_mechanism = tf.contrib.seq2seq.LuongAttention(

num\_units, attention\_states,

memory\_sequence\_length=source\_sequence\_length)

在前面的[编码器](https://github.com/tensorflow/nmt#encoder)部分中，encoder\_outputs是顶层所有源隐藏状态的集合，其形状为[max\_time，batch\_size，num\_units]（因为我们使用dynamic\_rnn，time\_major设置为 True以提高效率）。对于注意机制，我们需要确保传入的“内存”是批量主要的，因此我们需要转移 attention\_states。我们将source\_sequence\_length传递给注意机制，以确保注意权重被正确归一化（仅在非填充位置上）。

定义了注意机制后，我们使用AttentionWrapper来包装解码单元：

decoder\_cell = tf.contrib.seq2seq.AttentionWrapper(

decoder\_cell, attention\_mechanism,

attention\_layer\_size=num\_units)

其余的代码几乎与Section [Decoder中](https://github.com/tensorflow/nmt#decoder)的相同！

## 实践 - 构建基于注意力的NMT模型

要启用的关注，我们需要使用的一个luong，scaled\_luong，bahdanau 或normed\_bahdanau作为价值attention标志的训练中。该标志指定我们将使用哪种注意机制。此外，我们需要为注意力模型创建一个新目录，因此我们不会重复使用先前训练的基本NMT模型。

运行以下命令以开始培训：

mkdir / tmp / nmt\_attention\_model

python -m nmt.nmt \

--attention = scaled\_luong \

--src = vi --tgt = en \

--vocab\_prefix = / tmp / nmt\_data / vocab \

--train\_prefix = / tmp / nmt\_data / train \

--dev\_prefix = / tmp / nmt\_data / tst2012 \

--test\_prefix = / tmp / nmt\_data / tst2013 \

--out\_dir = / tmp / nmt\_attention\_model \

--num\_train\_steps = 12000 \

--steps\_per\_stats = 100 \

--num\_layers = 2 \

--num\_units = 128 \

--dropout = 0.2 \

--metrics =bleu

训练之后，我们可以使用相同的推理命令和新的out\_dir进行推理：

python -m nmt.nmt \

--out\_dir = / tmp / nmt\_attention\_model \

--inference\_input\_file = / tmp / my\_infer\_file.vi \

--inference\_output\_file = / TMP / nmt\_attention\_model / output\_infer

## 提示与技巧

## 构建训练，评估和推理图

在TensorFlow中构建机器学习模型时，通常最好构建三个单独的图：

* 培训图表，其中：
  + 批量，桶和可能的子样本从一组文件/外部输入中输入数据。
  + 包括前向和后向操作。
  + 构造优化器，并添加训练操作。
* Eval图，其中：
  + 批处理和存储桶从一组文件/外部输入中输入数据。
  + 包括培训前瞻性操作，以及未用于培训的其他评估操作。
* 推理图，其中：
  + 可能无法批量输入数据。
  + 不进行子采样或桶输入数据。
  + 从占位符读取输入数据（数据可以通过feed\_dict或C ++ TensorFlow服务二进制文件直接提供给图形）。
  + 包括模型转发操作的子集，以及可能用于在session.run调用之间存储状态的其他特殊输入/输出。

构建单独的图表有几个好处：

* 推理图通常与其他两个图非常不同，因此单独构建它是有意义的。
* eval图变得更简单，因为它不再具有所有额外的backprop操作。
* 可以为每个图单独实施数据馈送。
* 变量重用更简单。例如，在eval图中，不需要使用reuse = True重新打开变量作用域，因为Training模型已经创建了这些变量。因此，可以重用相同的代码，而无需在任何地方使用reuse = arguments。
* 在分布式培训中，让不同的工作人员进行培训，评估和推理是司空见惯的。无论如何，这些都需要构建自己的图形。因此，以这种方式构建系统可以帮助您进行分布式培训。

复杂性的主要来源变为如何在单个机器设置中跨三个图形共享变量。这可以通过为每个图使用单独的会话来解决。训练会话定期保存检查点，并从检查点恢复eval会话和推断会话恢复参数。以下示例显示了两种方法之间的主要差异。

**之前：单个图中的三个模型并共享一个会话**

with tf.variable\_scope('root'):

train\_inputs = tf.placeholder()

train\_op, loss = BuildTrainModel(train\_inputs)

initializer = tf.global\_variables\_initializer()

with tf.variable\_scope('root', reuse=True):

eval\_inputs = tf.placeholder()

eval\_loss = BuildEvalModel(eval\_inputs)

with tf.variable\_scope('root', reuse=True):

infer\_inputs = tf.placeholder()

inference\_output = BuildInferenceModel(infer\_inputs)

sess = tf.Session()

sess.run(initializer)

for i in itertools.count():

train\_input\_data = ...

sess.run([loss, train\_op], feed\_dict={train\_inputs: train\_input\_data})

if i % EVAL\_STEPS == 0:

while data\_to\_eval:

eval\_input\_data = ...

sess.run([eval\_loss], feed\_dict={eval\_inputs: eval\_input\_data})

if i % INFER\_STEPS == 0:

sess.run(inference\_output, feed\_dict={infer\_inputs: infer\_input\_data})

**之后：三个图中的三个模型，三个会话共享相同的变量**

train\_graph = tf.Graph()

eval\_graph = tf.Graph()

infer\_graph = tf.Graph()

with train\_graph.as\_default():

train\_iterator = ...

train\_model = BuildTrainModel(train\_iterator)

initializer = tf.global\_variables\_initializer()

with eval\_graph.as\_default():

eval\_iterator = ...

eval\_model = BuildEvalModel(eval\_iterator)

with infer\_graph.as\_default():

infer\_iterator, infer\_inputs = ...

infer\_model = BuildInferenceModel(infer\_iterator)

checkpoints\_path = "/tmp/model/checkpoints"

train\_sess = tf.Session(graph=train\_graph)

eval\_sess = tf.Session(graph=eval\_graph)

infer\_sess = tf.Session(graph=infer\_graph)

train\_sess.run(initializer)

train\_sess.run(train\_iterator.initializer)

for i in itertools.count():

train\_model.train(train\_sess)

if i % EVAL\_STEPS == 0:

checkpoint\_path = train\_model.saver.save(train\_sess, checkpoints\_path, global\_step=i)

eval\_model.saver.restore(eval\_sess, checkpoint\_path)

eval\_sess.run(eval\_iterator.initializer)

while data\_to\_eval:

eval\_model.eval(eval\_sess)

if i % INFER\_STEPS == 0:

checkpoint\_path = train\_model.saver.save(train\_sess, checkpoints\_path, global\_step=i)

infer\_model.saver.restore(infer\_sess, checkpoint\_path)

infer\_sess.run(infer\_iterator.initializer, feed\_dict={infer\_inputs: infer\_input\_data})

while data\_to\_infer:

infer\_model.infer(infer\_sess)

注意后一种方法如何“准备好”转换为分布式版本。

新方法的另一个不同之处在于，我们使用有状态迭代器对象，而不是使用feed\_dicts 在每个session.run调用（从而执行我们自己的数据的批处理，分段和操作）中提供数据。这些迭代器使单输入和分布式设置中的输入管道更加容易。我们将在下一节介绍新的输入数据管道（如TensorFlow 1.2中所介绍的）。

## 数据输入管道

在TensorFlow 1.2之前，用户有两种方法可以将数据提供给TensorFlow培训和评估管道：

1. 在每次培训session.run调用时直接通过feed\_dict提供数据。
2. 使用tf.train中的排队机制（例如tf.train.batch）和 tf.contrib.train。
3. 使用来自更高级别框架的帮助程序，如tf.contrib.learn或 tf.contrib.slim（有效地使用＃2）。

对于不熟悉TensorFlow或需要进行外来输入修改（即，他们自己的小批量排队）的用户来说，第一种方法更容易，只能在Python中完成。第二种和第三种方法更标准，但灵活性稍差; 他们还需要启动多个python线程（队列运行程序）。此外，如果使用不正确，队列可能会导致死锁或不透明的错误消息。尽管如此，队列比使用feed\_dict更有效，并且是单机和分布式训练的标准。

从TensorFlow 1.2开始，有一个新系统可用于将数据读入TensorFlow模型：数据集迭代器，如**tf.data** 模块中所示。数据迭代器灵活，易于推理和操作，并通过利用TensorFlow C ++运行时提供效率和多线程。

甲**数据集**可以从一批数据张量，一个文件名，或包含多个文件名一个张量来创建。一些例子：

# Training dataset consists of multiple files.

train\_dataset = tf.data.TextLineDataset(train\_files)

# Evaluation dataset uses a single file, but we may

# point to a different file for each evaluation round.

eval\_file = tf.placeholder(tf.string, shape=())

eval\_dataset = tf.data.TextLineDataset(eval\_file)

# For inference, feed input data to the dataset directly via feed\_dict.

infer\_batch = tf.placeholder(tf.string, shape=(num\_infer\_examples,))

infer\_dataset = tf.data.Dataset.from\_tensor\_slices(infer\_batch)

可以通过输入处理类似地处理所有数据集。这包括读取和清理数据，分组（在训练和评估的情况下），过滤和批处理。

例如，要将每个句子转换为单词字符串的向量，我们使用数据集映射转换：

dataset = dataset.map(lambda string: tf.string\_split([string]).values)

然后我们可以将每个句子向量切换为包含向量及其动态长度的元组：

dataset = dataset.map(lambda words: (words, tf.size(words))

最后，我们可以对每个句子进行词汇查找。给定查找表对象表，此映射将第一个元组元素从字符串向量转换为整数向量。

dataset = dataset.map(lambda words, size: (table.lookup(words), size))

加入两个数据集也很容易。如果两个文件包含彼此的逐行转换，并且每个文件都被读入其自己的数据集，则可以通过以下方式创建包含压缩行的元组的新数据集：

source\_target\_dataset = tf.data.Dataset.zip((source\_dataset, target\_dataset))

对可变长度句子进行批处理很简单。以下转换从source\_target\_dataset批处理batch\_size元素，并分别将源向量和目标向量填充到每个批次中最长源和目标向量的长度。

batched\_dataset = source\_target\_dataset.padded\_batch(

batch\_size,

padded\_shapes=((tf.TensorShape([None]), # source vectors of unknown size

tf.TensorShape([])), # size(source)

(tf.TensorShape([None]), # target vectors of unknown size

tf.TensorShape([]))), # size(target)

padding\_values=((src\_eos\_id, # source vectors padded on the right with src\_eos\_id

0), # size(source) -- unused

(tgt\_eos\_id, # target vectors padded on the right with tgt\_eos\_id

0))) # size(target) -- unused

从此数据集发出的值将是嵌套元组，其张量的最左侧维度为batch\_size。结构将是：

* 迭代器[0] [0]具有批处理和填充的源句子矩阵。
* iterator [0] [1]具有批处理的源大小向量。
* 迭代器[1] [0]具有批处理和填充的目标句子矩阵。
* 迭代器[1] [1]具有批量目标大小向量。

最后，也可以将类似大小的源句子拼凑在一起。有关更多详细信息和完整实现，请参阅文件 [utils / iterator\_utils.py](https://github.com/tensorflow/nmt/blob/master/nmt/utils/iterator_utils.py)。

从数据集读取数据需要三行代码：创建迭代器，获取其值并初始化它。

batched\_iterator = batched\_dataset.make\_initializable\_iterator()

((source, source\_lengths), (target, target\_lengths)) = batched\_iterator.get\_next()

# At initialization time.

session.run(batched\_iterator.initializer, feed\_dict={...})

初始化迭代器后，访问源或目标张量的每个session.run调用都将从基础数据集请求下一个小批量。

## 更好的NMT模型的其他细节

### 双向RNN

编码器侧的双向性通常提供更好的性能（随着使用更多层，速度会有一些降低）。在这里，我们给出了如何使用单个双向层构建编码器的简化示例：

# Construct forward and backward cells

forward\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units)

backward\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units)

bi\_outputs, encoder\_state = tf.nn.bidirectional\_dynamic\_rnn(

forward\_cell, backward\_cell, encoder\_emb\_inp,

sequence\_length=source\_sequence\_length, time\_major=True)

encoder\_outputs = tf.concat(bi\_outputs, -1)

变量encoder\_outputs和encoder\_state可以与Section Encoder中相同的方式使用。请注意，对于多个双向层，我们需要稍微操作一下encoder\_state，有关详细信息，请参阅[model.py](https://github.com/tensorflow/nmt/blob/master/nmt/model.py)，方法 \_build\_bidirectional\_rnn（）。

### 光束搜索

虽然贪婪解码可以为我们提供非常合理的翻译质量，但是波束搜索解码器可以进一步提高性能。光束搜索的想法是通过在我们翻译时保留一小组顶级候选者来更好地探索所有可能翻译的搜索空间。光束的大小称为 *光束宽度* ; 最小的光束宽度，例如尺寸10，通常就足够了。有关更多信息，请参阅[Neubig的](https://arxiv.org/abs/1703.01619)第7.2.3节[（2017）](https://arxiv.org/abs/1703.01619)。以下是梁搜索的示例：

# Replicate encoder infos beam\_width times

decoder\_initial\_state = tf.contrib.seq2seq.tile\_batch(

encoder\_state, multiplier=hparams.beam\_width)

# Define a beam-search decoder

decoder = tf.contrib.seq2seq.BeamSearchDecoder(

cell=decoder\_cell,

embedding=embedding\_decoder,

start\_tokens=start\_tokens,

end\_token=end\_token,

initial\_state=decoder\_initial\_state,

beam\_width=beam\_width,

output\_layer=projection\_layer,

length\_penalty\_weight=0.0)

# Dynamic decoding

outputs, \_ = tf.contrib.seq2seq.dynamic\_decode(decoder, ...)

请注意，使用相同的*dynamic\_decode（）* API调用，类似于Section [Decoder](https://github.com/tensorflow/nmt#decoder)。解码后，我们可以按如下方式访问翻译：

translations = outputs.predicted\_ids

# Make sure translations shape is [batch\_size, beam\_width, time]

if self.time\_major:

translations = tf.transpose(translations, perm=[1, 2, 0])

有关更多详细信息，请参阅[model.py](https://github.com/tensorflow/nmt/blob/master/nmt/model.py)，方法\_build\_decoder（）。

### 超参数

有几个超参数可以带来额外的性能。在这里，我们根据自己的经验列出一些[免责声明：其他人可能不同意我们写的东西！]。

***优化器***：虽然Adam可以为“不熟悉的”架构带来合理的结果，但如果您可以使用SGD进行培训，那么SGD的调度通常会带来更好的性能。

***注意***：Bahdanau风格的注意力往往需要编码器方面的双向性才能正常工作; 而Luong式的注意力往往适用于不同的环境。对于本教程代码，我们建议使用Luong和Bahdanau风格注意的两个改进版本： scaled\_luong和 normed bahdanau。

### 多GPU培训

训练NMT模型可能需要几天时间。在不同的GPU上放置不同的RNN层可以提高训练速度。这是在多个GPU上创建RNN层的示例。

cells = []

for i in range(num\_layers):

cells.append(tf.contrib.rnn.DeviceWrapper(

tf.contrib.rnn.LSTMCell(num\_units),

"/gpu:%d" % (num\_layers % num\_gpus)))

cell = tf.contrib.rnn.MultiRNNCell(cells)

此外，我们需要启用colocate\_gradients\_with\_ops选项 tf.gradients以并行化渐变计算。

您可能会注意到，随着GPU数量的增加，基于注意力的NMT模型的速度提升非常小。标准注意力架构的一个主要缺点是使用顶层（最终）层的输出来查询每个时间步的注意力。这意味着每个解码步骤必须等待其上一步完全完成; 因此，我们不能通过简单地将RNN层放置在多个GPU上来并行化解码过程。

所述[GNMT关注架构](https://arxiv.org/pdf/1609.08144.pdf) 通过使用底（第一）层的输出来查询注意并行化解码器的计算。因此，每个解码步骤可以在其前一步骤的第一层和注意力计算完成时立即开始。我们实施的架构 [GNMTAttentionMultiCell](https://github.com/tensorflow/nmt/blob/master/nmt/gnmt_model.py)，的一个子类tf.contrib.rnn.MultiRNNCell。以下是如何使用GNMTAttentionMultiCell创建解码器单元的示例。

cells = []

for i in range(num\_layers):

cells.append(tf.contrib.rnn.DeviceWrapper(

tf.contrib.rnn.LSTMCell(num\_units),

"/gpu:%d" % (num\_layers % num\_gpus)))

attention\_cell = cells.pop(0)

attention\_cell = tf.contrib.seq2seq.AttentionWrapper(

attention\_cell,

attention\_mechanism,

attention\_layer\_size=None, # don't add an additional dense layer.

output\_attention=False,)

cell = GNMTAttentionMultiCell(attention\_cell, cells)

## 参考

* Dzmitry Bahdanau，Kyunghyun Cho和Yoshua Bengio。2015. [通过联合学习对齐和翻译的神经机器翻译](https://arxiv.org/pdf/1409.0473.pdf)。ICLR。
* Minh-Thang Luong，Hieu Pham和Christopher D Manning。2015. [基于注意力的神经机器翻译的有效方法](https://arxiv.org/pdf/1508.04025.pdf)。EMNLP。
* Ilya Sutskever，Oriol Vinyals和Quoc V. Le。2014年[的序列测序与神经网络学习](https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf)。NIPS。