TensorFlow教程-keras構建RNN

迴圈神經網路是一種在時間維度上進行反覆運算的神經網路，在建模序列資料方面有著優越的性能。 TensorFlow2中包含了主流的rnn網路實現，以Keras RNN API的方式提供調用。其特性如下：

- 易用性： 內置tf.keras.layers.RNN，tf.keras.layers.LSTM，tf.keras.layers.GRU， 可以快速構建RNN模組。
- 易於定制：可以使用自訂操作迴圈來構建RNN模組，並通過tf.keras.layers.RNN調用。

In [1]:
```
from __future__ import absolute_import, division, print_function, unicode_literal

import collections
import matplotlib.pyplot as plt
import numpy as np

import tensorflow as tf
print(tf.__version__)
from tensorflow.keras import layers
```

```
/home/doit/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWar
ning: Conversion of the second argument of issubdtype from `float` to `np.float
ing` is deprecated. In future, it will be treated as `np.float64 == np.dtype(fl
oat).type`.
  from ._conv import register_converters as _register_converters

2.0.0
```

1 構建一個簡單模型

keras中內置了三個RNN層

- tf.keras.layers.SimpleRNN，普通RNN網路。
- tf.keras.layers.GRU，門控迴圈神經網路。
- tf.keras.layers.LSTM，長短期記憶神經網路。 下面是一個迴圈神經網路的例子，它使用LSTM層來處理輸入的詞嵌入，LSTM反覆運算的次數與詞個個數一致

In [2]:
```
model = tf.keras.Sequential()
# input_dim是詞典大小， output_dim是詞嵌入維度
model.add(layers.Embedding(input_dim=1000, output_dim=64))
# 添加lstm層，其會輸出最後一個時間步的輸出
model.add(layers.LSTM(128))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          64000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               98816
_________________________________________________________________
dense (Dense)                (None, 10)                1290
=================================================================
Total params: 164,106
Trainable params: 164,106
Non-trainable params: 0
_________________________________________________________________
```

2 輸出和狀態

預設情況下RNN層輸出最後一個時間步的輸出，輸出的形狀為(batch_size, units), 其中unit是傳給層的構造參數。 如果要返回rnn每個時間步的序列，需要置return_sequence=True, 此時輸出的形狀為(batch_size, time_steps, units)。

In [3]:
```
model = tf.keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
# 返回整個rnn序列的輸出
model.add(layers.GRU(128, return_sequences=True))
model.add(layers.SimpleRNN(128))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 64)          64000
_________________________________________________________________
gru (GRU)                    (None, None, 128)         74496
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, 128)               32896
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 172,682
Trainable params: 172,682
Non-trainable params: 0
_________________________________________________________________
```

RNN層可以返回最後一個時間步的狀態。

返回的狀態可以用於恢復RNN或初始化另一個RNN。此設置通常在seq2seq模型中用到，其將編碼器的最終狀態作為解碼器的初始狀態。

要使RNN層返回最終的狀態需要在創建圖層時置return_state=True。請注意，LSTM有兩個狀態向量，而GRU只有一個。

要配置圖層的初始狀態， 需要網路構建中傳入initial_state。其中狀態的維度必須與圖層的unit大小一樣。

In [4]:
```
encoder_vocab = 1000
decoder_vocab = 2000


# 編碼層
encode_input = layers.Input(shape=(None, ))
encode_emb = layers.Embedding(input_dim=encoder_vocab, output_dim=64)(encode_inpu
# 同時返回狀態
encode_out, state_h, state_c = layers.LSTM(64, return_state=True, name='encoder')
encode_state = [state_h, state_c]

# 解碼層
decode_input =layers.Input(shape=(None, ))
decode_emb = layers.Embedding(input_dim=encoder_vocab, output_dim=64)(decode_inpu
# 編碼器的最終狀態， 作為解碼器的初始狀態
decode_out = layers.LSTM(64, name='decoder')(decode_emb, initial_state=encode_sta
output = layers.Dense(10, activation='softmax')(decode_out)

model = tf.keras.Model([encode_input, decode_input], output)
model.summary()
```

```
Model: "model"
__________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
input_1 (InputLayer)            [(None, None)]       0
__________________________________________________________________________________
input_2 (InputLayer)            [(None, None)]       0
__________________________________________________________________________________
embedding_2 (Embedding)         (None, None, 64)     64000       input_1[0][0]
__________________________________________________________________________________
embedding_3 (Embedding)         (None, None, 64)     64000       input_2[0][0]
__________________________________________________________________________________
encoder (LSTM)                  [(None, 64), (None,  33024       embedding_2[0]
[0]
__________________________________________________________________________________
decoder (LSTM)                  (None, 64)           33024       embedding_3[0]
[0]
                                                                 encoder[0][1]
                                                                 encoder[0][2]
__________________________________________________________________________________
dense_2 (Dense)                 (None, 10)           650         decoder[0][0]
==================================================================================
Total params: 194,698
Trainable params: 194,698
Non-trainable params: 0
```

3 RNN層和RNN Cell

除了內置的RNN層以外， RNN API還提供單元級API。與可以處理整批次的輸入序列不同， RNN Cell僅能處理單個時間步的資料。 如果想處理整批次的輸入資料，需要將RNN Cell包含在tf.keras.layers.RNN中， 如：RNN(LSTMCell(10)

同效果上說， RNN(LSTMCell(10))等價於LSTM(10)。但內置的GRU和LSTM層可以使用CuDNN，以提高計算性能。

下面是三個內置的RNN單元， 以及其對應的RNN層。

- tf.keras.layers.SimpleRNNCell對應於SimpleRNN圖層。
- tf.keras.layers.GRUCell對應於GRU圖層。
- tf.keras.layers.LSTMCell對應於LSTM圖層。

注： tf.keras.layers.RNN類可以使為研究實現自訂RNN體系結構變得非常容易。

In [5]:
```
model = tf.keras.Sequential()
# input_dim是詞典大小， output_dim是詞嵌入維度
model.add(layers.Embedding(input_dim=1000, output_dim=64))
# 添加lstm層，其會輸出最後一個時間步的輸出
model.add(layers.RNN(layers.LSTMCell(64)))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, None, 64)          64000
_________________________________________________________________
rnn (RNN)                    (None, 64)                33024
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 97,674
Trainable params: 97,674
Non-trainable params: 0
_________________________________________________________________
```

4 跨批次狀態

在處理超長序列（有可能無限長）時，可能需要使用到跨批次狀態的模式。

通常，每次看到新的批次時， 都會重置RNN層的內部狀態（state，非權重。即該層看到的每個樣本都獨立於過去）。

但，當序列過長，需要分不同批次輸入時，則無需重置RNN的狀態(上一批的結束狀態，為下一批的初始狀態)。這樣，即使一次只看到一個子序列，網路層也可以保留整個序列的資訊。

我們可以置state_ful=True來設置跨批次狀態。

如果有序列s = [t0, t1, ... t1546, t1547]， 可以將其分為以下批次數據：

s1 = [t0, t1, ... t100]

s2 = [t101, ... t201]

...

16 = [t1501, ... t1547]

具體調用方法如下：

In [6]:
```
sub_sequence = []
lstm_layer = layers.LSTM(64, stateful=True)
for s in sub_sequence:
    output = lstm_layer(s)
```

要清除狀態時，可以使用layer.reset_states()。

注意：在此設置中，i假定給定批次中的樣品i是上一個批次中樣品的延續。這意味著所有批次應包含相同數量的樣本（批次大小）。例如，如果一個批次包含[sequence_A_from_t0_to_t100, sequence_B_from_t0_to_t100]，則下一個批次應包含[sequence_A_from_t101_to_t200, sequence_B_from_t101_to_t200]。

跨批次狀態例子

In [7]:
```
para1 = np.random.random((20, 10, 50)).astype(np.float32)
para2 = np.random.random((20, 10, 50)).astype(np.float32)
para3 = np.random.random((20, 10, 50)).astype(np.float32)
lstm_layer = layers.LSTM(64, stateful=True)
output = lstm_layer(para1)
output = lstm_layer(para2)
output = lstm_layer(para3)
lstm_layer.reset_states()
```

5 雙向LSTM

對於時間序列以外的序列，比如文本，RNN不僅可以正向處理序列，也可以反向處理序列。例如要預測句子的某個單純，上下文對單詞都有用。

Keras提供了tf.keras.layers.Bidirectional的API，構建雙向RNN。

In [8]:
```
model = tf.keras.Sequential()
model.add(layers.Bidirectional(layers.LSTM(64, return_sequences=True),
                               input_shape=(5, 10)))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional (Bidirectional (None, 5, 128)            38400
_________________________________________________________________
bidirectional_1 (Bidirection (None, 64)                41216
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
=================================================================
Total params: 80,266
Trainable params: 80,266
Non-trainable params: 0
_________________________________________________________________
```

在內部， Bidirectional將複製傳入的RNN，並將逆序輸入新複製的RNN，並輸出前向輸出和後向輸出的疊加。如果想要其他合併行為（如串聯），可以修改merge_mode等參數。

6 TensorFlow2.0中的性能優化和CuDNN內核

在TensorFlow2.0中，內置的LSTM和GRU層已更新，已在有GPU時默認使用CuDNN內核。通過此更改，先前的keras.layers.CuDNNLSTM/CuDNNGRU層已經棄用，可以簡易的構建模型而不必擔心其運行的硬體。

由於CuDNN內核是根據某些假設構建的，因此如果改變內置LSTM或GRU的默認設置，則該層無法使用CuDNN內核，例如:

- 將activation從tanh改為其他啟動函數。
- 將recurrent_activation由sigmoid改為其他啟動函數。
- 使recurrent_dropout> 0。
- 設置unroll為True，將強制LSTM / GRU將內部分解tf.while_loop為展開的for迴圈。
- 設置use_bias為False。
- 當輸入資料未嚴格右填充時使用掩蔽（如果遮罩對應於嚴格右填充資料，則仍可以使用CuDNN。這是最常見的情況）。

6.1 在可用時使用CuDNN內核

我們構建一個簡單的LSTM網路，來實現MNIST數位識別。

In [9]:
```
batch_size = 64
input_dim = 28
units = 64
output_size = 10

def build_model(allow_cudnn_kernel=True):
    if allow_cudnn_kernel:
        # LSTM預設cudnn加速
        lstm_layer = tf.keras.layers.LSTM(units, input_shape=(None, input_dim))
    else:
        # LSTMCell內核沒有使用
        lstm_layer = tf.keras.layers.RNN(tf.keras.layers.LSTMCell(units),
                                         input_shape=(None, input_dim))
    model = tf.keras.models.Sequential([
        lstm_layer,
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(output_size, activation='softmax')
    ])
    return model
```

載入MNIST資料集

In [10]:
```
mnits = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnits.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
sample, sample_label = x_train[0], y_train[0]
```

創建模型實例並進行編譯

In [11]:
```
model = build_model(allow_cudnn_kernel=True)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

In [16]:
```
# 使用cudnn的訓練
model.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          batch_size=batch_size,
          epochs=2)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/2
60000/60000 [==============================] - 10s 169us/sample - loss: 0.1457
- accuracy: 0.9560 - val_loss: 0.1290 - val_accuracy: 0.9566
Epoch 2/2
60000/60000 [==============================] - 10s 170us/sample - loss: 0.1284
- accuracy: 0.9614 - val_loss: 0.1394 - val_accuracy: 0.9543
```

Out[16]:
```
<tensorflow.python.keras.callbacks.History at 0x7f94d0c8e7b8>
```

在沒有CuDNN內核的情況下構建新模型

In [15]:
```
slow_model = build_model(allow_cudnn_kernel=False)
#slow_model.set_weights(model.get_weights())
slow_model.compile(loss='sparse_categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
slow_model.fit(x_train, y_train,
               validation_data=(x_test, y_test),
               batch_size=batch_size,
               epochs=2)
```

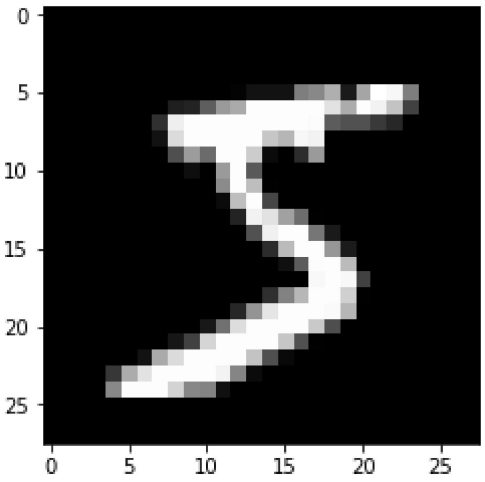
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/2
60000/60000 [==============================] - 11s 185us/sample - loss: 0.9310
- accuracy: 0.7030 - val_loss: 0.5106 - val_accuracy: 0.8419
Epoch 2/2
60000/60000 [==============================] - 10s 172us/sample - loss: 0.4209
- accuracy: 0.8715 - val_loss: 0.4863 - val_accuracy: 0.8416
```

Out[15]:
```
<tensorflow.python.keras.callbacks.History at 0x7f94f31478d0>
```

使用cudnn的模型也可以在cpu環境中進行推理

In [18]:
```
with tf.device('CPU:0'):
    cpu_model = build_model(allow_cudnn_kernel=True)
    cpu_model.set_weights(model.get_weights())
    result = tf.argmax(cpu_model.predict_on_batch(tf.expand_dims(sample, 0)), axi
    print('預測結果: %s, 正確標籤: %s' % (result.numpy(), sample_label))
    plt.imshow(sample, cmap=plt.get_cmap('gray'))
```

预测结果: [5], 正确标签: 5



7.具有清單/字典輸入或嵌套輸入的RNN

嵌套結構可以在單個時間步內包含更多資訊。例如，一個視頻幀可以同時具有音訊和視頻輸入。比如以下的資料格式：

[batch, timestep, {"video": [height, width, channel], "audio": [frequency]}]

另一個例子，如筆跡資料。可以有當前的位置座標x，y和壓力資訊。格式如下：

[batch, timestep, {"location": [x, y], "pressure": [force]}]

7.1 定義一個支持嵌套輸入/輸出的自訂儲存格

In [26]:
```
NestedInput = collections.namedtuple('NestedInput', ['feature1', 'feature2'])
NestedState = collections.namedtuple('NestedState', ['state1', 'state2'])

class NestedCell(tf.keras.layers.Layer):
    # 初始化，獲取相關參數
    def __init__(self,unit1, unit2, unit3, **kwargs):
        self.unit1 = unit1
        self.unit2 = unit2
        self.unit3 = unit3
        self.state_size = NestedState(state1=unit1,
                                      state2=tf.TensorShape([unit2, unit3]))

        self.output_size = (unit1, tf.TensorShape([unit2, unit3]))
        super(NestedCell, self).__init__(**kwargs)
    # 構建權重、網路
    def build(self, input_shapes):
        # input_shape包含2個特徵項 [(batch, i1), (batch, i2, i3)]
        input1 = input_shapes.feature1[1]
        input2, input3 = input_shapes.feature2[1:]

        self.kernel_1 = self.add_weight(
            shape=(input1, self.unit1), initializer='uniform', name='kernel_1'
        )
        self.kernel_2_3 = self.add_weight(
            shape=(input2, input3, self.unit2, self.unit3),
            initializer='uniform',
            name='kernel_2_3'
        )
    # 前向連接網路
    def call(self, inputs, states):
        # inputs: [(batch, input_1), (batch, input_2, input_3)]
        # state: [(batch, unit_1), (batch, unit_2, unit_3)]
        input1, input2 = tf.nest.flatten(inputs)
        s1, s2 = states

        output_1 = tf.matmul(input1, self.kernel_1)
        output_2_3 = tf.einsum('bij,ijkl->bkl', input2, self.kernel_2_3)

        state_1 = s1 + output_1
        state_2_3 = s2 + output_2_3

        output = [output_1, output_2_3]
        new_states = NestedState(state1=state_1, state2=state_2_3)
        return output, new_states
```

7.2 使用嵌套的輸入輸出構建RNN

In [28]:
```
unit_1 = 10
unit_2 = 20
unit_3 = 30

input_1 = 32
input_2 = 64
input_3 = 32
batch_size = 64
num_batch = 100
timestep = 50
cell = NestedCell(unit_1, unit_2, unit_3)
rnn = tf.keras.layers.RNN(cell)
input1 = tf.keras.Input((None, input_1))
input2 = tf.keras.Input((None, input_2, input_3))
outputs = rnn(NestedInput(feature1=input1, feature2=input2))
model = tf.keras.models.Model([input1, input2], outputs)
model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
```

構造數據訓練

In [29]:
```
input_1_data = np.random.random((batch_size * num_batch, timestep, input_1))
input_2_data = np.random.random((batch_size * num_batch, timestep, input_2, input
target_1_data = np.random.random((batch_size * num_batch, unit_1))
target_2_data = np.random.random((batch_size * num_batch, unit_2, unit_3))
input_data = [input_1_data, input_2_data]
target_data = [target_1_data, target_2_data]

model.fit(input_data, target_data, batch_size=batch_size)
```

```
Train on 6400 samples
6400/6400 [==============================] - 191s 30ms/sample - loss: 0.3808 -
rnn_7_loss: 0.1197 - rnn_7_1_loss: 0.2611 - rnn_7_accuracy: 0.1003 - rnn_7_1_ac
curacy: 0.0333
```

Out[29]:
```
<tensorflow.python.keras.callbacks.History at 0x7f94c9ee4ba8>
```

In [ ]:
```

```