Bài 18 - Các layers quan trọng trong deep learning

02 Dec 2019 - phamdinhkhanh

Menu

1. Giới thiệu chung

Mỗi một mô hình deep learning đều có các layers đặc trưng giúp giải quyết các tác vụ cụ thể của từng bài toán deep learning. Chẳng hạn như trong xử lý ảnh chúng ta thường sử dụng mạng CNN - convolutional neural network (https://www.kaggle.com/phamdinhkhanh/convolutional-neural-network-p1) để trích xuất đặc trưng trên các local regional của bức ảnh là các đường nét chính như dọc, ngang, chéo,…. Hoặc layer LSTM - long short term memory (https://phamdinhkhanh.github.io/2019/04/22/Ly_thuyet_ve_mang_LSTM.html) được sử dụng trong các mô hình dịch máy và mô hình phân loại cảm xúc văn bản (sentiment analysis). Tuy nhiên ngoài các layer trên, chúng ta sẽ còn làm quen với rất nhiều các layers khác trong các bài toán về deep learning. Việc hiểu được công dụng của từng layer cũng như trường hợp áp dụng để mang lại hiệu quả cho mô hình rất quan trọng. Chính vì thế bài viết này nhằm mục đích hệ thống lại các layers quan trọng của deep learning như một tài liệu cheat sheet (tài liệu sổ tay) để sử dụng khi cần.

Do các layer CNN và LSTM đã được trình bày ở 2 bài viết của blog nên tôi sẽ không nêu lại kiến thức của những layers này. Và tất nhiên đó là những layers rất quan trọng mà bạn đọc cần nắm vững để áp dụng trong các mô hình về xử lý ảnh và ngôn ngữ. Bài viết này chỉ hướng tới những layer khác quan trọng hơn.

2. Các layer cơ bản

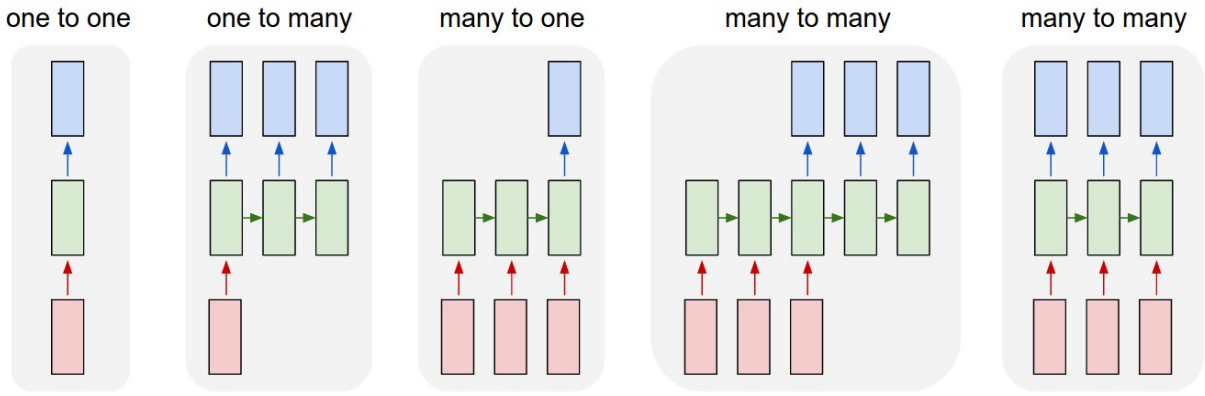
2.1. Time distributed

Dường như cái tên của layer này đã nói lên ý nghĩa của nó là xác định phân phối của dữ liệu theo thời gian.

Để hiểu rõ hơn chúng ta cùng lấy ví dụ về dự báo dạng chuỗi thông qua mạng RNN. Nhưng khoan khoan, để hiểu những gì tôi sắp viết vui lòng hiểu kĩ kiến trúc mạng RNN thông qua bài LSTM - long short term memory (https://phamdinhkhanh.github.io/2019/04/22/Ly_thuyet_ve_mang_LSTM.html).

Bạn đã đọc xong và nắm vững kiến thức về LSTM rồi chứ? Nếu chắc chắn, chúng ta hãy tiếp tục nào. Bên dưới là những dạng dự báo của RNN:

Top



Hình 1: Các dạng dự báo trong RNN

- One to one: Là trường hợp chỉ có 1 input và trả ra kết quả 1 output.
- One to many: Là trường hợp đầu vào chỉ có 1 input. Véc tơ context ở bước liền sau $t+1$ và kết hợp với output dự báo ở bước t sẽ được sử dụng để dự báo được output ở bước $t+1$. Cứ tiếp tục quá trình đến hết chuỗi, từ 1 input ta dự báo được một chuỗi nhiều outputs.
- Many to one: Là trường hợp ta chỉ trả ra kết quả là véc tơ output tại time step cuối cùng trong mạng LSTM (tương đương với cấu hình return_sequence = False trong LSTM layer của keras).
- Many to many: Đây chính là kiến trúc đặc trưng của model dịch máy. Tại mỗi bước thời gian t, input là véc tơ embedding một từ của ngôn ngữ nguồn và trả kết quả đầu ra là một véc tơ phân phối xác suất của từ output ở ngôn ngữ đích.

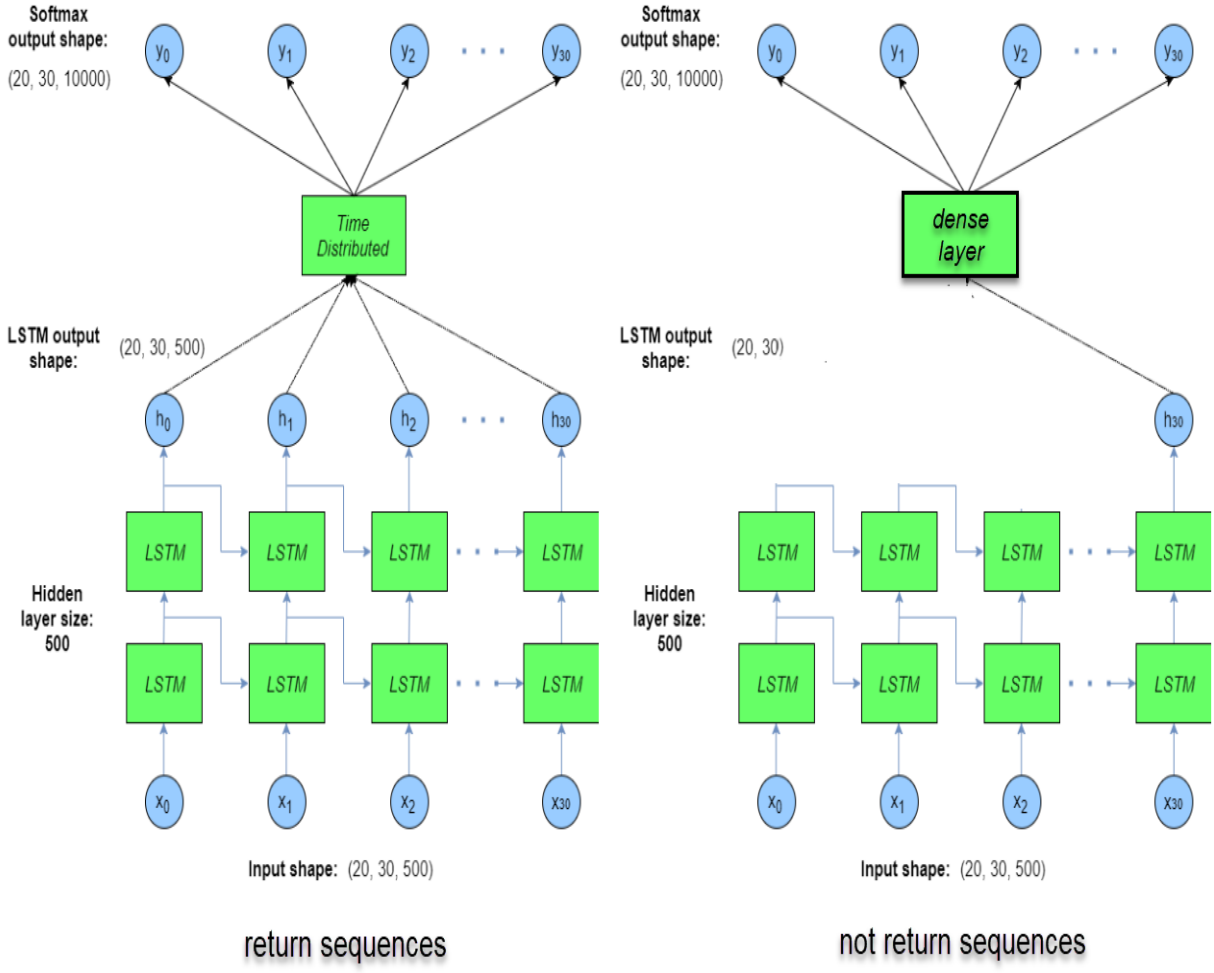
Trong các bài toán về NLP, Khi áp dụng họ các layer RNN thì chúng ta thường có 2 lựa chọn đó là:

Lựa chọn 1: trả ra chỉ kết quả là hidden layer ở véc tơ cuối cùng hoặc

Lựa chọn 2: trả ra chuỗi các hidden véc tơ ở mỗi time step.

Theo sau đó, các layers tiếp theo là những fully connected layers có kiến trúc như một mạng MLP thông thường. Kết quả trả ra là dự báo phân phối xác suất nhãn. Câu hỏi được đặt ra là ta sẽ áp dụng các fully connected layers như thế nào cho từng lựa chọn? Rõ ràng đối với lựa chọn 1 thì do đầu ra của LSTM layer là một véc tơ nên ta dễ dàng truyền qua một Dense Layer (hay còn gọi là fully connected layer) thông thường và xây dựng một chuỗi fully connected layers khá dễ dàng. Tuy nhiên đối với lựa chọn 2 làm thế nào ta có thể kết hợp một Dense Layer với một chuỗi các hidden véc tơ như output của trường hợp `many to many` và `one to many` mà các bạn thấy. Khi đó chúng ta cần một layer đặc biệt hơn, không chỉ có tác dụng như dense layer trong mạng MLP mà còn có tác dụng kết nối tới từng hidden véc tơ ở mỗi bước thời gian, đó chính là Time Distributed Layers. Để dễ hình dung hơn bạn đọc có thể xem hình so sánh bên dưới giữa Time Distributed Layer và Dense Layer.

Top



Hình 2: Time Distributed Layer ở bên trái kết nối chung các hidden véc tơ ở mỗi bước thời gian từ h_0 đến h_{30} tới cùng một dense layer. Trong khi ở hình bên phải, đầu vào của dense layer chính là véc tơ cuối cùng h_{30}.

Như vậy về bản chất Time Distributed Layer không khác gì một Dense Layer thông thường. Chính vì vậy trong một issue When and How to use TimeDistributedDense (https://github.com/keras-team/keras/issues/1029) fchollet tác giả của keras (và rất nhiều các đầu sách về deep learning trên cả python và R) đã giải thích một cách ngắn gọn như khá khó hiểu đối với beginers:

`TimeDistributedDense applies a same Dense (fully-connected) operation to every timestep of a 3D tensor.`

Ngoài ra Time Distributed Layer còn được sử dụng rất nhiều trong các mô hình xử lý video. Giả định rằng đầu vào của bạn là những batch video gồm các 5 chiều: `(batch_size, time, width, height, channels)`. Như vậy để áp dụng một mạng tích chập 2 chiều lên toàn bộ các khung hình theo thời gian chúng ta cần sử dụng Time Distributed để thu được output shape mới ở đầu ra gồm 4 chiều: `(batch_size, new_width, new_height, output_channels)`.

Như vậy bạn đọc đã hình dung ra nguyên lý hoạt động của Time Distributed Layer rồi chứ? Để hiểu thêm cách thức sử dụng trên keras, bạn đọc có thể tham khảo tài liệu Time Distributed Layer - keras document (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed).

Bên dưới chúng ta sẽ cùng đi xây dựng 2 lớp model phân loại mail rác sử dụng 2 phương pháp khác nhau là `LSTM + Dense Layer` và `LSTM + Time Distributed Layer` và đối chiếu kết quả thu được.

Top

Dữ liệu được sử dụng là sms spam collection dataset (https://www.kaggle.com/uciml/sms-spam-collection-dataset) gồm 5574 emails tiếng anh đã được gán nhãn sẵn là các email spam (email rác)/ham (email hợp lệ). Để xây dựng model chúng ta sẽ đi qua các bước như sau:

Bước 1: Xử lý dữ liệu.

Tại bước này chúng ta sẽ cần đọc và khảo sát dữ liệu để kiểm tra tính cân bằng, loại bỏ các từ stop words, dấu câu, kí tự đặc biệt và tạo từ điển để mã hóa các từ sang index.

Tất cả các công việc này được thực hiện khá dễ dàng nhờ những module có sẵn của `gensim` và `keras`.

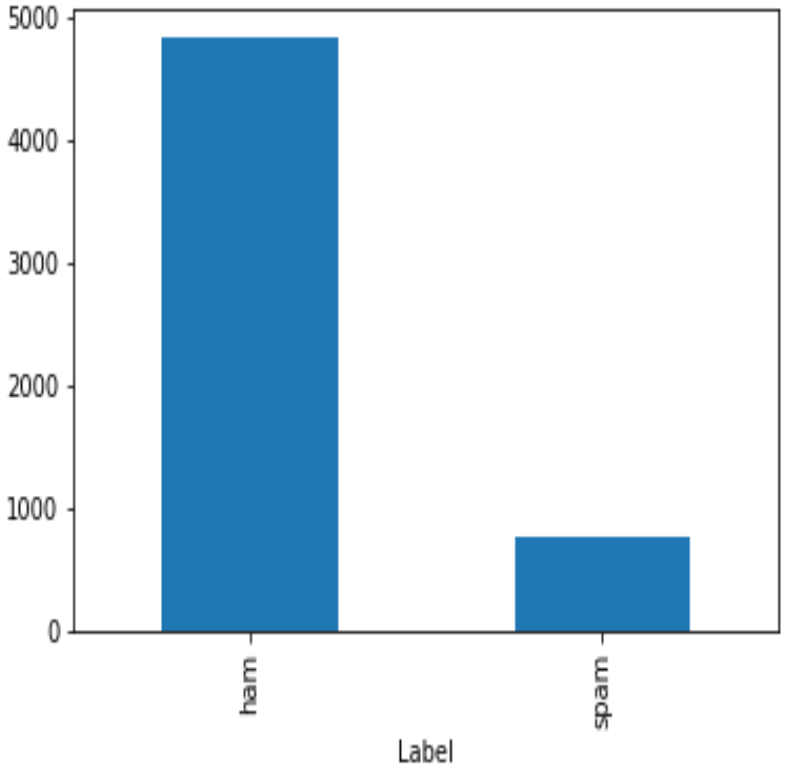
- Đọc và khảo sát dữ liệu:

```
import pandas as pd
dataset = pd.read_csv('spam.csv', header=0, sep=',',encoding='latin-1'
dataset = dataset.iloc[:, :2]
dataset.columns = ['Label', 'Email']
dataset.head()
```

	Label	Email
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Đồ thị phân phối các classes:

```
dataset.groupby('Label').Email.count().plot.bar()
```



- Làm sạch dữ liệu bằng cách chuẩn hóa các từ viết hoa thành viết thường, loại bỏ dấu câu, loại bỏ chữ số, tách số dính liền với từ và loại bỏ stop words,… thông qua package gensim.

Top

```
import gensim
from gensim.parsing.preprocessing import strip_non_alphanum, strip_mu
import re
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
stop_words = stopwords.words('english')

# Chuyển chữ hoa sang chữ thường
def lower_case(docs):
  return [doc.lower() for doc in docs]

# Loại bỏ các dấu câu và kí tự đặc biệt
def remove_punc(docs):
  return [strip_non_alphanum(doc).strip() for doc in docs]

# Tách các số và chữ liền nhau
def separate_num(docs):
  return [split_alphanum(doc) for doc in docs]

# Loại bỏ những từ gồm 1 chữ cái đứng đơn lẻ
def remove_one_letter_word(docs):
  return [strip_short(doc) for doc in docs]

# Loại bỏ các con số trong văn bản vì chúng không có nhiều ý nghĩa tr
def remove_number(docs):
  return [strip_numeric(doc) for doc in docs]

# Thay thế nhiều khoảng spaces bằng 1 khoảng space
def replace_multiple_whitespaces(docs):
  return [strip_multiple_whitespaces(doc) for doc in docs]

# Loại bỏ các stop words
def remove_stopwords(docs):
    return [[word for word in doc.split() if word not in stop_words]

docs = lower_case(dataset['Email'])
docs = remove_punc(docs)
docs = separate_num(docs)
docs = remove_one_letter_word(docs)
docs = remove_number(docs)
docs = replace_multiple_whitespaces(docs)
docs = remove_stopwords(docs)
dataset['Content_Clean'] = docs
```

- Khởi tạo tokenizer để mã hóa các email.

Để chuyển các câu văn thành ma trận số, chúng ta cần tạo ra một từ điển mapping mỗi từ với index tương ứng của nó. module tokenizer dễ dàng giúp ta thực hiện việc này.

Top

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing import sequence

def _tokenize_matrix(docs, max_words=1000, max_len=150):
  #max_words: Số lượng từ lớn nhất xuất hiện trong tokenizer được lấy
  #max_len: Số lượng các từ lớn nhất trong một câu văn.
  #docs: Tợp hợp các đoạn email.
  tok = Tokenizer(num_words=max_words)
  tok.fit_on_texts(docs)
  X_tok =  tok.texts_to_sequences(docs)
  X = sequence.pad_sequences(X_tok, maxlen=max_len)
  return X_tok, X, tok

X_tok, X, tok = _tokenize_matrix(docs=dataset['Content_Clean'], max_w
```

Mã hóa nhãn spam/ham về biến one-hot.

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(dataset['Label'])
```

Bước 2: Xây dựng model LSTM

Sau khi đã preprocessing dữ liệu, chúng ta thu được đầu vào là các ma trận padding X mà các dòng của nó là những câu văn có độ dài bằng nhau. Từ ma trận padding, ta cần đi qua một layer embedding để tạo véc tơ nhúng cho mỗi từ trong câu, và sau đó đi vào mạng LSTM. Chúng ta sẽ áp dụng cả 2 kiến trúc mô hình là Dense Layer và Time Distributed Layer.

- Khởi tạo model Dense Layer

Top

```
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Ac
from tensorflow.keras.models import Model, Sequential
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.optimizers import RMSprop, Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping

def RNN_Dense(maxword=1000, embedding_size=100, max_len=150, n_unit_l
    inputs = Input(name='inputs', shape=[max_len])
    layer = Embedding(maxword, embedding_size, input_length=max_len)(
    # Embedding (input_dim: size of vocabolary,
    # output_dim: dimension of dense embedding,
    # input_length: length of input sequence)
    layer = LSTM(n_unit_lstm)(layer)
    layer = Dense(n_unit_dense, name='FC1')(layer)
    layer = Activation('relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(1, name='out_layer')(layer)
    layer = Activation('sigmoid')(layer)
    model = Model(inputs=inputs,outputs=layer)
    return model

lstm_dense=RNN_Dense()
lstm_dense.summary()
lstm_dense.compile(loss='binary_crossentropy', optimizer=Adam(), metr
```

```
Model: "model_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inputs (InputLayer)          [(None, 150)]             0
_________________________________________________________________
embedding_20 (Embedding)     (None, 150, 100)          100000
_________________________________________________________________
lstm_20 (LSTM)               (None, 64)                42240
_________________________________________________________________
FC1 (Dense)                  (None, 256)               16640
_________________________________________________________________
activation_29 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_14 (Dropout)         (None, 256)               0
_________________________________________________________________
out_layer (Dense)            (None, 1)                 257
_________________________________________________________________
activation_30 (Activation)   (None, 1)                 0
=================================================================
Total params: 159,137
Trainable params: 159,137
Non-trainable params: 0
_________________________________________________________________
```

- Huấn luyện model dense

Top

```
lstm_dense.fit(X, y, batch_size=128,
                epochs=10,
                validation_split=0.2,
                callbacks=[EarlyStopping(monitor='val_loss',min_delta=0
```

```
Train on 4457 samples, validate on 1115 samples
Epoch 1/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 2/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
...
Epoch 9/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
Epoch 10/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
```

- Khởi tạo model TimeDistributed Layer

```
def RNN_TimeDis(maxword=1000, embedding_size=100, max_len=150, n_unit_
    inputs = Input(name='inputs', shape=[max_len])
    layer = Embedding(maxword, embedding_size, input_length=max_len)(
    # Embedding (input_dim: size of vocabolary,
    # output_dim: dimension of dense embedding,
    # input_length: length of input sequence)
    layer = LSTM(n_unit_lstm, return_sequences=True)(layer)
    layer = TimeDistributed(Dense(n_unit_dense))(layer)
    layer = Flatten()(layer)
    layer = Activation('relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(1, name='out_layer')(layer)
    layer = Activation('sigmoid')(layer)
    model = Model(inputs=inputs,outputs=layer)
    return model

lstm_timedis=RNN_TimeDis()
lstm_timedis.summary()
lstm_timedis.compile(loss='binary_crossentropy', optimizer=Adam(), me
```

Top

```
Model: "model_15"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inputs (InputLayer)          [(None, 150)]             0
_________________________________________________________________
embedding_22 (Embedding)     (None, 150, 100)          100000
_________________________________________________________________
lstm_22 (LSTM)               (None, 150, 64)           42240
_________________________________________________________________
time_distributed_3 (TimeDist (None, 150, 64)           4160
_________________________________________________________________
flatten_1 (Flatten)          (None, 9600)              0
_________________________________________________________________
activation_31 (Activation)   (None, 9600)              0
_________________________________________________________________
dropout_15 (Dropout)         (None, 9600)              0
_________________________________________________________________
out_layer (Dense)            (None, 1)                 9601
_________________________________________________________________
activation_32 (Activation)   (None, 1)                 0
=================================================================
Total params: 156,001
Trainable params: 156,001
Non-trainable params: 0
_________________________________________________________________
```

- Huấn luyện model Time Distributed Layer

```
lstm_timedis.fit(X, y, batch_size=128,
                 epochs=10,
                 validation_split=0.2,
                 callbacks=[EarlyStopping(monitor='val_loss',min_delta=0
```

```
Train on 4457 samples, validate on 1115 samples
Epoch 1/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 2/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 3/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 4/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 5/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
Epoch 6/10
4457/4457 [==============================] - 13s 3ms/sample - loss: 0
```

Kết quả cho thấy model áp dụng Time Distributed và áp dụng Dense đều khá tốt. Mức độ chính xác thu được trên tập validation đều trên 95%.

Top

Ngoài ra thì Time Distributed còn được áp dụng trong phân loại nội dung video rất hiệu quả. Có thời gian tôi sẽ hướng dẫn các bạn.

2.2. Batch Normalization

Nếu bạn đã làm quen với các kiến trúc mô hình trong CNN như Alexnet, VGG16, VGG19 thì batch normalization là một layer được sử dụng khá nhiều. Layer này thường áp dụng ngay sau Convolutional layer và thường ở những vị trí đầu tiên của mô hình để đạt hiệu quả cao nhất. Mục đích chính của batch normalization đó là chuẩn hóa dữ liệu ở các layer theo batch về phân phối chuẩn để quá trình gradient descent hội tụ nhanh hơn.

Gỉa sử chúng ta có một mini-batch với giá trị của từng quan sát trong batch như sau: $\mathcal{B} = x_1, x_2, \ldots, x_m$ Khi đó Batch Normalization sẽ được xác định thông qua phép chuẩn hóa trung bình và phương sai của các phần tử trong batch theo công thức bên dưới:

$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$

$\hat{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$

giá trị $\hat{x_i}$ chính là kết quả sau chuẩn hóa.

Để hiểu hơn về cách áp dụng và hiệu quả của batch normalization, chúng ta sẽ thử nghiệm xây dựng mô hình LeNet có sử dụng batch normalization và không sử dụng batch normalization và so sánh kết quả mô hình sau huấn luyện. Bộ dữ liệu mà chúng ta sử dụng là mnist gồm các bức ảnh kích thước 28 x 28 của 10 chữ số từ 0 đến 9. Thật may mắn là dữ liệu này có thể load trực tiếp từ keras, rất tiện lợi.

Bước 1: Load dữ liệu từ keras.

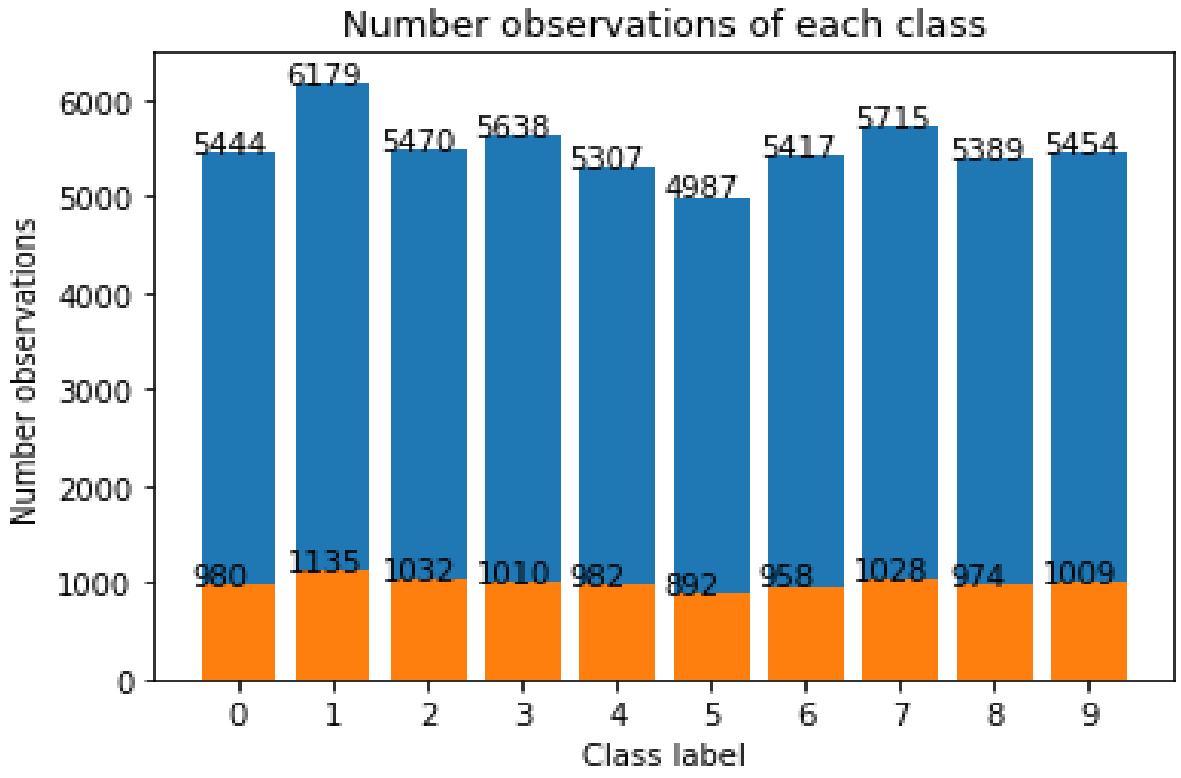
```
# Load dữ liệu.
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
from tensorflow import keras
print(tf.__version__)

# mnist = keras.datasets.mnist
# (X_train, y_train), (X_test, y_test) = mnist.load_data()

mnist = input_data.read_data_sets("MNIST_data/", reshape=False)
X_train, y_train = mnist.train.images, mnist.train.labels
X_val, y_val = mnist.validation.images, mnist.validation.labels
X_test, y_test = mnist.test.images, mnist.test.labels
print('X_train shape: {}'.format(X_train.shape))
print('y_train shape: {}'.format(y_train.shape))
print('X_test shape: {}'.format(X_test.shape))
print('y_test shape: {}'.format(y_test.shape))
print('X_val shape: {}'.format(X_val.shape))
print('y_val shape: {}'.format(y_val.shape))
```

```
X_train shape: (55000, 28, 28, 1)
y_train shape: (55000,)
X_test shape: (10000, 28, 28, 1)
y_test shape: (10000,)
X_val shape: (5000, 28, 28, 1)
y_val shape: (5000,)
```

Top

Bước tiếp theo tuy đơn giản nhưng vô cùng quan trọng. Rất nhiều beginer thường bỏ qua vì chưa có kinh nghiệm. Đó là kiểm tra phân phối số quan sát trên các nhóm.

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

# Thống kê số lượng ở mỗi classes
def _plot_bar(x, title = 'Number observations of each class'):
  x, y = np.unique(x, return_counts=True)
  x_lab = [str(lab) for lab in x]
  plt.bar(x_lab, y)
  plt.xlabel('Class label')
  plt.ylabel('Number observations')
  plt.title(title)

  for i in range(len(x)): # your number of bars
    plt.text(x = x[i]-0.5, #takes your x values as horizontal positio
    y = y[i]+1, #takes your y values as vertical positioning argument
    s = y[i], # the labels you want to add to the data
    size = 10)

_plot_bar(y_train)
_plot_bar(y_test)
```

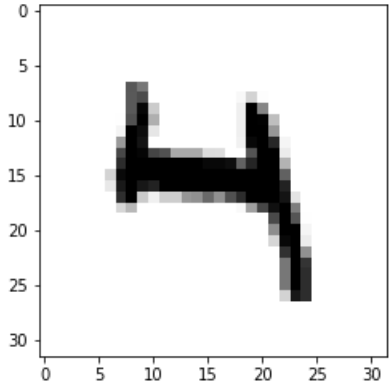


Do input của mạng LeNet có kích thước `32 x 32` nên ta cần padding thêm các chiều để có kích thước là `32 x 32` . Thực hiện dễ dàng như sau:

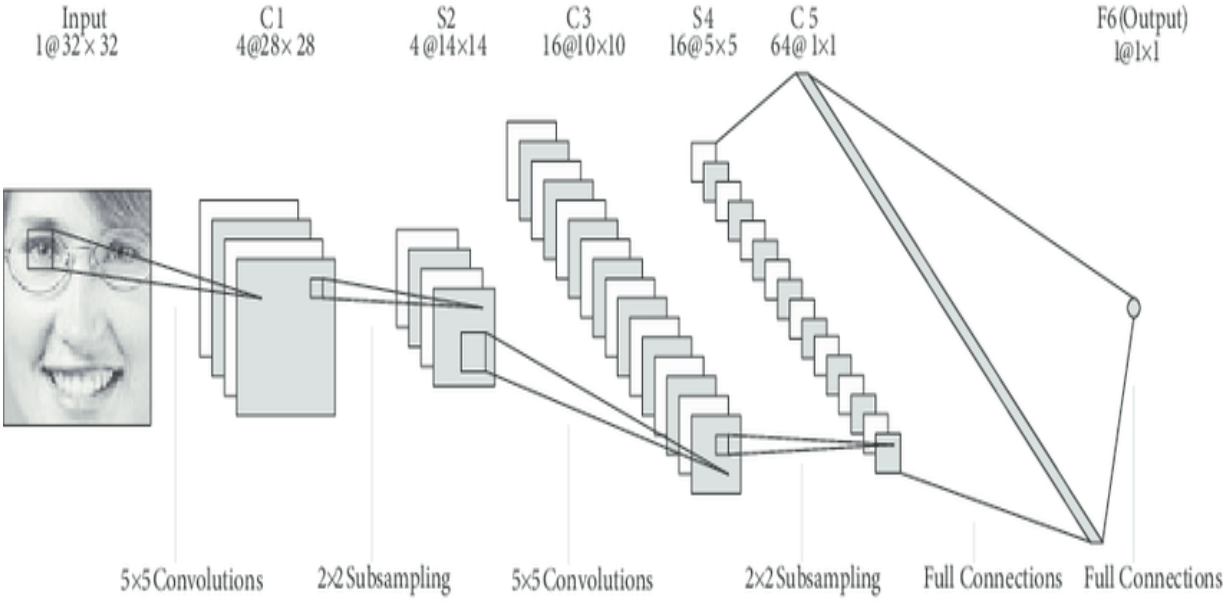
Top

```
X_train = np.pad(X_train, ((0, 0), (2, 2), (2, 2), (0, 0)), 'constant
X_val = np.pad(X_val, ((0, 0), (2, 2), (2, 2), (0, 0)), 'constant')
X_test = np.pad(X_test, ((0, 0), (2, 2), (2, 2), (0, 0)), 'constant')

print('X_train shape: {}'.format(X_train.shape))
print('X_test shape: {}'.format(X_test.shape))
print('X_val shape: {}'.format(X_val.shape))

# Để chắc chắn đã padding đúng thì hiển thị ra 1 hình ảnh số để kiểm
plt.imshow(np.squeeze(X_train[np.random.randint(55000)], axis=2), cma
```

```
X_train shape: (55000, 32, 32, 1)
X_test shape: (10000, 32, 32, 1)
X_val shape: (5000, 32, 32, 1)
```



LeNet là mạng CNN đơn giản được tạo ra vào 1998 bởi Yan LeCun khi ông áp dụng thêm các layers tích chập (Convolutional) kết hợp với MaxPooling. Trong đó layer tích chập giúp nhận diện các đường nét dọc, ngang, chéo,… của vật thể còn MaxPooling nhằm giảm chiều dữ liệu mà không thay đổi các đặc trưng của ảnh giúp nhận diện vật thể. Cụ thể về kiến trúc từng layers các bạn xem hình vẽ bên dưới.



Top

Hình 3: Kiến trúc mạng LeNet. Trong đó input layer nhận đầu vào là những ảnh kích thước 32 x 32. Tiếp theo là 2 lượt (Convolutional 2D + Maxpooling) giúp giảm chiều dữ liệu từ 32 x 32 xuống còn 5 x 5. Flatten kết quả thành véc tơ và chuyển sang một mạng fully connected thông thường ta sẽ thu được giá trị dự báo xác suất ở output.

Chúng ta khởi tạo model LeNet không có BatchNormalization.

Top

```
from tensorflow.keras.layers import Flatten, Dense, Input, Activation
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.models import Sequential, Model

def _Lenet_No_BatchNorm():
  inp = Input(shape=(32, 32, 1))
  conv1 = Conv2D(
      filters=4,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(inp)

  maxpool1 = MaxPooling2D(
      pool_size=2,
      strides=2
  )(conv1)

  conv2 = Conv2D(
      filters=16,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(maxpool1)

  maxpool2 = MaxPooling2D(
      pool_size=2,
      strides=2
  )(conv2)

  conv3 = Conv2D(
      filters=64,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(maxpool2)

  flatten = Flatten()(conv3)
  dense1 = Dense(units=64, activation='relu')(flatten)
  dense2 = Dense(units=32, activation='relu')(dense1)
  dense3 = Dense(10, activation='sigmoid')(dense2)
  output = Activation('softmax')(dense3)
  model = Model(inputs = inp, outputs=output)
  model.summary()
  return model

lenet_no_batchnorm = _Lenet_No_BatchNorm()
```

Top

```
Model: "model_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 32, 32, 1)]       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 4)         104
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 4)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 10, 16)        1616
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 1, 64)          25664
_________________________________________________________________
flatten_3 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_7 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_8 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_9 (Dense)              (None, 10)                330
_________________________________________________________________
activation_34 (Activation)   (None, 10)                0
=================================================================
Total params: 33,954
Trainable params: 33,954
Non-trainable params: 0
_________________________________________________________________
```

Huấn luyện model LeNet khi không có Batch Normalization

```
from tensorflow.keras.callbacks import EarlyStopping
# Compile model
optimizer = Adam(learning_rate=0.005, beta_1=0.9, beta_2=0.999, amsgra
lenet_no_batchnorm.compile(loss='sparse_categorical_crossentropy', op

# Khởi tạo model checkpoint
earlyStopping = EarlyStopping(monitor='val_loss', min_delta=0.0001, pa
# Huấn luyện model
lenet_no_batchnorm.fit(X_train, y_train,
                batch_size=256,
                # validation_data=[X_val, y_val],
                validation_split=0.2,
                epochs=10,
                shuffle=True,
                callbacks=[earlyStopping])
```

Top

```
Train on 44000 samples, validate on 11000 samples
Epoch 1/10
44000/44000 [==============================] - 21s 479us/sample - los
Epoch 2/10
44000/44000 [==============================] - 20s 462us/sample - los
...
Epoch 6/10
44000/44000 [==============================] - 20s 463us/sample - los
Epoch 7/10
44000/44000 [==============================] - 20s 465us/sample - los
```

Khởi tạo model LeNet khi có BatchNormalization

Top

```
from tensorflow.keras.layers import Flatten, Dense, Input, Activation
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.models import Sequential, Model

def _Lenet_BatchNorm():
  inp = Input(shape=(32, 32, 1))
  conv1 = Conv2D(
      filters=4,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(inp)

  batch_norm1 = BatchNormalization()(conv1)

  maxpool1 = MaxPooling2D(
      pool_size=2,
      strides=2
  )(batch_norm1)

  conv2 = Conv2D(
      filters=16,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(maxpool1)

  batch_norm2 = BatchNormalization()(conv2)

  maxpool2 = MaxPooling2D(
      pool_size=2,
      strides=2
  )(batch_norm2)

  conv3 = Conv2D(
      filters=64,
      kernel_size=5,
      padding='valid',
      strides=1,
      activation='relu')(maxpool2)

  flatten = Flatten()(conv3)
  dense1 = Dense(units=64, activation='relu')(flatten)
  dense2 = Dense(units=32, activation='relu')(dense1)
  dense3 = Dense(10, activation='sigmoid')(dense2)
  output = Activation('softmax')(dense3)
  model = Model(inputs = inp, outputs=output)
  model.summary()
  return model

lenet_batchnorm = _Lenet_BatchNorm()
```

Top

```
Model: "model_22"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, 32, 32, 1)]       0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 28, 28, 4)         104
_________________________________________________________________
batch_normalization_6 (Batch (None, 28, 28, 4)         16
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 14, 14, 4)         0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 10, 10, 16)        1616
_________________________________________________________________
batch_normalization_7 (Batch (None, 10, 10, 16)        64
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 1, 1, 64)          25664
_________________________________________________________________
flatten_8 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_22 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_23 (Dense)             (None, 32)                2080
_________________________________________________________________
dense_24 (Dense)             (None, 10)                330
_________________________________________________________________
activation_39 (Activation)   (None, 10)                0
=================================================================
Total params: 34,034
Trainable params: 33,994
Non-trainable params: 40
_________________________________________________________________
```

Huấn luyện model với Batch Normalization

```
from tensorflow.keras.callbacks import EarlyStopping
# Compile model
optimizer = Adam(learning_rate=0.005, beta_1=0.9, beta_2=0.999, amsgra
lenet_batchnorm.compile(loss='sparse_categorical_crossentropy', optim

# Khởi tạo model checkpoint
earlyStopping = EarlyStopping(monitor='val_loss', min_delta=0.0001, pa
# Huấn luyện model
lenet_batchnorm.fit(X_train, y_train,
                batch_size=256,
                # validation_data=[X_val, y_val],
                validation_split=0.2,
                epochs=10,
                shuffle=True,
                callbacks=[earlyStopping])
```

Top

```
Train on 44000 samples, validate on 11000 samples
Epoch 1/10
44000/44000 [==============================] - 30s 688us/sample - los
Epoch 2/10
44000/44000 [==============================] - 29s 659us/sample - los
...
Epoch 6/10
44000/44000 [==============================] - 29s 654us/sample - los
Epoch 7/10
44000/44000 [==============================] - 29s 655us/sample - los
```

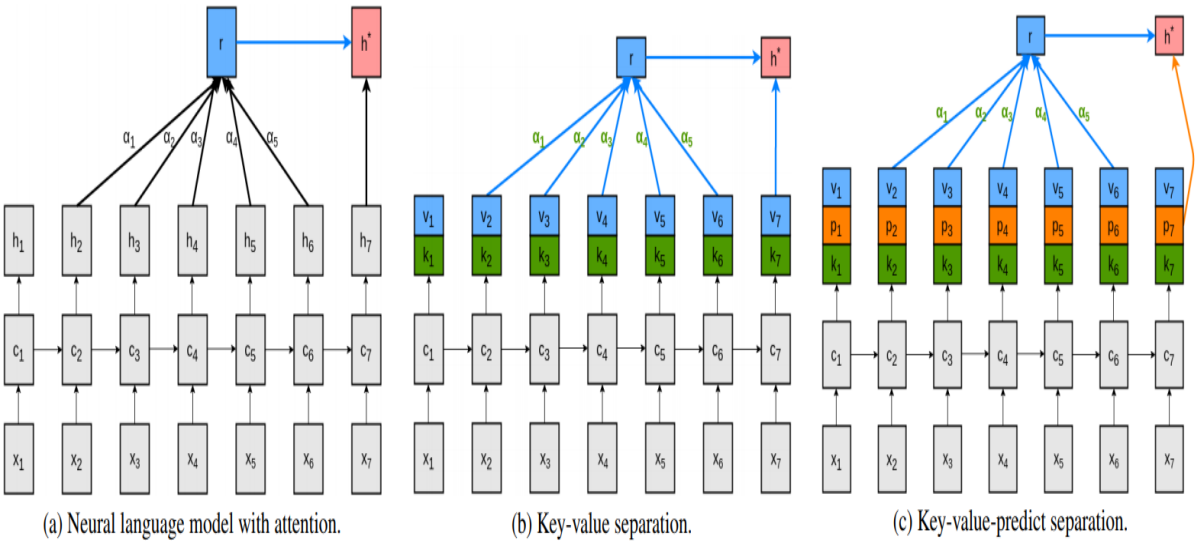
Ta thấy rằng model khi áp dụng Batch Normalization có tốc độ hội tụ nhanh hơn và độ chính xác cũng đồng thời cao hơn chút so với không áp dụng Batch Normalization. Điều đó chứng tỏ cơ chế chuẩn hóa dữ liệu sau mỗi layer đã giúp cho các tham số hội tụ nhanh hơn. Một thay đổi nhỏ nhưng hiệu quả mang lại thật bất ngờ.

2.3. Attention Layer

Trong những năm gần đây, độ chính xác của các mô hình dịch máy được cải thiện một cách đáng kể, thành tựu đó đạt được chính là nhờ một layer đặc biệt có tác dụng phân bố lại trọng số attention weight của các một từ input lên từ output sao cho càng ở vị trí gần thì trọng số càng cao.

Cụ thể về thuật toán attention tôi sẽ không trình bày chi tiết ở đây bởi tôi đã giới thiệu ở một bài trước đó Bài 4: Attention is all you need (https://phamdinhkhanh.github.io/2019/06/18/AttentionLayer.html).

Như đã biết về cơ chế của thuật toán RNN, chúng ta yêu cầu output véc tơ tại mỗi time step liên tục lưu trữ thông tin để dự báo từ tiếp theo, tính toán attention và encode nội dung tương ứng với các step trong tương lai. Việc sử dụng output quá tải như vậy có thể ảnh hưởng đến tốc độ và hiệu năng của mô hình. Một kiến trúc key-value(-predict) attention đã được giới thiệu bởi Daniluk, 2017 (https://arxiv.org/abs/1702.04521.pdf) để giải quyết vấn đề trên. Kiến trúc này cho phép tách riêng ác phần tính attention và dự báo output. Cụ thể kiến trúc này như bên dưới:



(a) Neural language model with attention. (b) Key-value separation. (c) Key-value-predict separation.

Top

> **Hình 4:** Sơ đồ cơ chế key-value attention. Chúng ta chia đôi hidden output véc tơ thành các cặp key-value véc tơ để giảm tải gánh nặng cho quá trình tính toán output.

Các cặp key-value được chia ra từ output véc tơ thành 2 phần như hình (b). Công dụng của các cặp key này như sau:

- phần keys chiếm một nứa hidden output được sử dụng để tính toán attention.
- phần value chiếm một nửa hidden output được sử dụng để giải mã phân phối của từ tiếp theo và biểu diễn của véc tơ context.

Các input đầu vào của mô hình là x và ta gọi đó là query.Trong một số trường hợp, ta lựa chọn giá trị của query và value là trùng nhau. Từ giá trị keys và values chúng ta tính được ra giá trị dự báo cho các bước tiếp theo và giá trị này chính là predict. Như vậy đầu ra cuối cùng của mô hình trả ra 3 véc tơ tại mỗi bước thời gian gồm: các véc tơ v được dùng để giải mã phân phối của từ tiếp theo, các véc tơ k được sử dụng như là key cho tính toán các attention véc tơ, véc tơ p như là value cuối cùng sau khi áp dụng cơ chế attention như chúng ta nhìn thấy trong hình 4(c).

Hiện nay kiến trúc key-value attention layer cho kết quả và tốc độ tính toán tốt hơn so với các kiến trúc attention trước đó.

Trong tensorflow version 2.0.0, attention layer được khởi tạo chính là dựa trên kiến trúc key-value attention layer. Các bạn có thể tham khảo tại: Attention Layer - Tensorflow (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Attention). Ngoài ra layer này còn cho phép sử dụng thêm các layer mask như query_mask và value_mask để loại bỏ attention tại một số vị trí nhất định trong query hoặc value.

Do mức độ tập trung phụ thuộc vào vị trí của từ trong câu nên attention layer được sử dụng chủ yếu ở những mô hình Seq2Seq. Ngoài ra trong các mô hình phân loại văn bản sử dụng LSTM thì attention layer cũng hoạt động khá hiệu quả.

Sau đây ta sẽ áp dụng Attention Layer đằng sau LSTM model trong tác vụ phân loại mail spam và so sánh hiệu quả so với áp dụng các layer như TimeDistrubted Layer và Dense Layer.

Top

```
from tensorflow.keras.layers import Layer, InputSpec
from tensorflow.keras import initializers

class AttLayer(Layer):
    def __init__(self, **kwargs):
        self.init = initializers.RandomNormal()
        #self.input_spec = [InputSpec(ndim=3)]
        super(AttLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape)==3
        #self.W = self.init((input_shape[-1],1))
        self.W = self.init((input_shape[-1],))
        #self.input_spec = [InputSpec(shape=input_shape)]
        self.trainable_weights = [self.W]
        super(AttLayer, self).build(input_shape)  # be sure you call

    def call(self, x, mask=None):
        eij = K.tanh(K.dot(x, self.W))

        ai = K.exp(eij)
        weights = ai/K.sum(ai, axis=1).dimshuffle(0,'x')

        weighted_input = x*weights.dimshuffle(0,1,'x')
        return weighted_input.sum(axis=1)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0], input_shape[-1])
```

Top

```
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Ac
from tensorflow.keras.models import Model, Sequential
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.optimizers import RMSprop, Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping

def RNN_AttLayer(maxword=1000, embedding_size=100, max_len=150, n_uni
    inputs = Input(name='inputs', shape=[max_len])
    layer = Embedding(maxword, embedding_size, input_length=max_len)(
    # Embedding (input_dim: size of vocabolary,
    # output_dim: dimension of dense embedding,
    # input_length: length of input sequence)
    layer = LSTM(n_unit_lstm, return_sequences=False)(layer)
    layer = Attention()([layer, layer])
    layer = Dense(n_unit_dense, name='FC1')(layer)
    layer = Activation('relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(1, name='out_layer')(layer)
    layer = Activation('sigmoid')(layer)
    model = Model(inputs=inputs,outputs=layer)
    return model

lstm_attlayer=RNN_AttLayer()
lstm_attlayer.summary()
lstm_attlayer.compile(loss='binary_crossentropy', optimizer=Adam(), m
```

Top

```
Model: "model_24"
_________________________________________________________________
Layer (type)                 Output Shape              Param #     Conn
=================================================================
inputs (InputLayer)          [(None, 150)]             0
_________________________________________________________________
embedding_28 (Embedding)     (None, 150, 100)          100000      inpu
_________________________________________________________________
lstm_28 (LSTM)               (None, 64)                42240       embe
_________________________________________________________________
attention_2 (Attention)      (None, 64)                0           lstm_
                                                                   lstm_
_________________________________________________________________
FC1 (Dense)                  (None, 64)                4160        atte
_________________________________________________________________
activation_42 (Activation)   (None, 64)                0           FC1[
_________________________________________________________________
dropout_17 (Dropout)         (None, 64)                0           acti
_________________________________________________________________
out_layer (Dense)            (None, 1)                 65          drop
_________________________________________________________________
activation_43 (Activation)   (None, 1)                 0           out_
=================================================================
Total params: 146,465
Trainable params: 146,465
Non-trainable params: 0
_________________________________________________________________
```

```
lstm_dense.fit(X, y, batch_size=128,
               epochs=10,
               validation_split=0.2,
               callbacks=[EarlyStopping(monitor='val_loss',min_delta=0
```

```
Train on 4457 samples, validate on 1115 samples
Epoch 1/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
Epoch 2/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
Epoch 3/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
Epoch 4/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
Epoch 5/10
4457/4457 [==============================] - 12s 3ms/sample - loss: 0
```

Như vậy ta thấy rằng khi áp dụng attention layer vào mô hình phân loại email thì độ chính xác của mô hình cao hơn. Ngay từ những epoch đầu tiên mô hình đã đạt được độ chính xác dường như hoàn hảo.

Điều này cho thấy attention layer rất mạnh trong việc nắm bắt các liên kết dài của chuỗi các từ trong câu và khắc phục được nhược điểm về sự phụ thuộc dài hạn kém của các mô hình Recurrent Neural Network.

top

Như vậy bài này tôi đã giới thiệu đến các bạn một số layers nổi bật của Deep Learning bao gồm Time Distributed, Batch Normalization và Attention Layer bên cạnh các layers chuyên biệt cho các mô hình xử lý ảnh (CNN) và ngôn ngữ tự nhiên (LSTM, GRU, RNN, BiDirectional RNN) mà các bạn chắc chắn phải nắm vững.

Qua các ví dụ minh họa chúng ta cũng so sánh được một cách tương đối hiệu quả của việc áp dụng các layers này và trường hợp nào, lớp mô hình hoặc bài toán nào thì nên áp dụng.

Ngoài ra còn rất nhiều các layers quan trọng khác của deep learning mà tôi sẽ viết ở những bài sau nữa nếu có dịp. Hi vọng rằng các bạn có thể áp dụng tốt những layers trên vào các bài toán của mình để nâng cao hiệu quả cho mô hình.

Và cuối cùng không thể thiếu trong các bài viết của Khanh Blog là Tài liệu tham khảo như một sự tôn trọng và tri ân gửi tới các tác giả mà tôi đã học hỏi.

3. Tài liệu tham khảo

1. Attention in NLP - Kate Loginova (https://medium.com/@joealato/attention-in-nlp-734c6fa9d983)
2. How to Use the TimeDistributed Layer in Keras - machinelearningmastery (https://machinelearningmastery.com/timedistributed-layer-for-long-short-term-memory-networks-in-python/)
3. How to work with Time Distributed data in a neural network - Patrice Ferlet (https://medium.com/smileinnovation/how-to-work-with-time-distributed-data-in-a-neural-network-b8b39aa4ce00)
4. Batch Normalization Tensorflow Keras Example - Cory Maklin (https://towardsdatascience.com/backpropagation-and-batch-normalization-in-feedforward-neural-networks-explained-901fd6e5393e)
5. Time Distributed Layer - tensorflow 2.0 (https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed)
6. Batch Normalization - tensorflow 2.0 (https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization)
7. Attention Layer - tensorflow 2.0 (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Attention)

Top