Social Bot Detection Based on Deep Learning

Presentation by: Waleed Khan (UCID: wak7, email: wak7@njit.edu)
Research by: Heng Ping and Sujuan Qin

Outline:

1. This Presentation will focus on Deep Learning used for Social Bot Detection
2. The Algorithm in the paper "**A Social Bots Detection Model Based on Deep Learning Algorithm**" combines a series of deep learning algorithms to create **DeBD**. These algorithms include:
    1. Word embedding using a Chosen Bag of Words (CBOW) Deep Neural Net (DNN)
    2. Feature extraction and classification using a Convolutional Neural Net (CNN)
    3. Feature extraction using a Long Short Term Memory (LSTM) Recurring Neural Network (RNN)
3. I have delved deepest into researching CBOW DNNs for Word Embedding, since this algorithm is what is used to discover the patterns and values of word usage. This is how we generate the data for human and bot language models. We then discover the difference and predict accordingly.
4. I have also provided examples and research into CNNs, and LSTM RNNs in order to learn how we train on these Word Embeddings in order to detect bots with the DeBD algorithm.

What are Social Bots?

- "Bot" is short for software robot
- "Chat" Bots were a previous trend, in which users could have text-based conversations with a robot online
- "Social" bots are artificial intelligences that interact with users (real and non-real) through social ecosystems such as Twitter
    - Many Social Bots can go ***undetected***:
        - These can be introduced into a social ecosystem without real users' knowledge
- The abundance of Social Bots is currently unknown due to the difficulty in differentiating between a bot and a real person

Why is it Important to Detect Social Bots?

- Social bots “automatically generate content, interact with people, attempt to imitate and change the behavior of others”
- Some are basic in design, and send simple tweets
- However, more complex bots can evolve to evade detections
- It is important to be able to detect the behavior of a malicious social bot
    - **Malicious bots can promote disinformation as well as biased political and social agendas** that may be harmful to society

How do We Begin to Detect Social Bots?

- "A Social Bots Detection Model Based on Deep Learning Algorithm", by Heng Ping and Sujuan Qin, proposes a model that demonstrates **more than 99% detecting accuracy**
- This uses the "**DeBD**" algorithm, which does the following:
  - Use the Convolutional Neural Network (**CNN)** algorithm:
    - Extract the joint features of the tweet content and the relationship between them →**"Joint Content Feature Extraction Layer"**
  - Use the Long Short-Term Memory (**LSTM)** algorithm:
    - Extract potential temporal features of the tweet metadata → **" Temporal Feature Extraction Layer"**
  - Temporal feature predictions are then fused with the joint content feature predictions to more accurately detect a social bot → "Fusing Layer"

The DeBd Algorithm Structure:

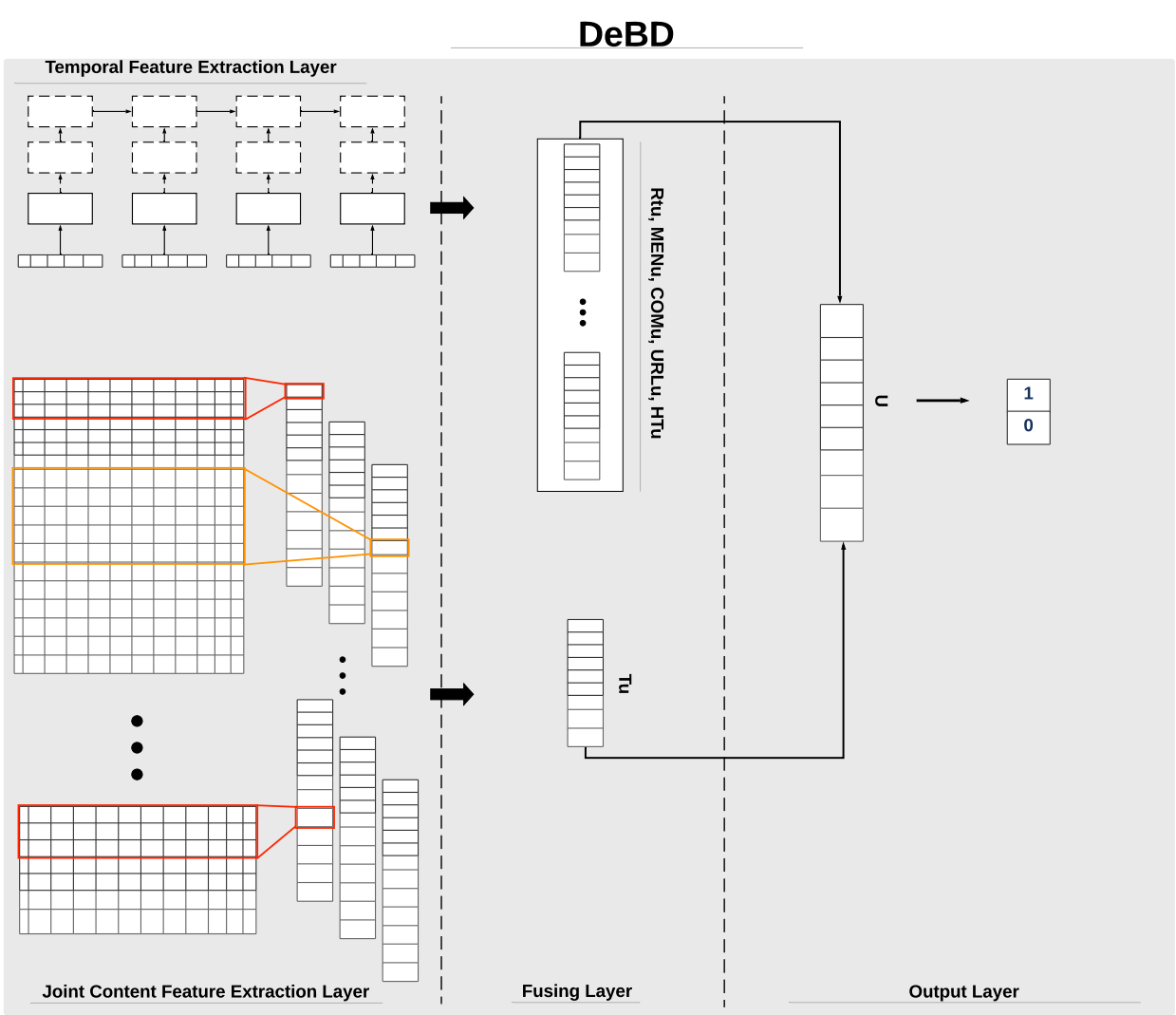


Diagram by Waleed Khan (me)

Joint Content Feature Extraction

- A tweet by user **u** is T_{ui}, where $i \in 1,2.....n$, where **n** is the total number of tweets posted by **u**
- All tweets of user u can be represented by a tweet sequence $T_u = T_{u1}, T_{u2} \dots T_{un}$
- To extract the features of tweets, they are converted into a **word embedding**
- Assume the number of words in T_{ui} is ***m***, then it is converted into matrix $T_{ui} \in R^{(m*w)}$ where ***w*** represents the dimension of the word embedding
- The Google unsupervised learning tool **Word2Vec (CBOW)** is used to produce this **word embedding**
    - This allows for a mathematical understanding of the words in a tweet, since words are converted into a vector space
    - **Word2Vec** is trained on millions of words from Google News Data in order to give accurate word embeddings

Word Embedding

- We want to give a value to words such that:
  - “King” - “Man” + “Woman” = “Queen”
  - Word associations can be determined mathematically this way, where we can predict the likelihood of a word appearing by calculating the word association for adjacent words
- All words from tweet ***i*** for user ***u*** are put into Matrix ***R*** with dimensions ***(m*** x ***w),*** where

  $T_{ui} \in R^{(m*w)}$
- Dimension ***m*** represents all unique words in our tweet, and dimension ***w*** is the number of features/neurons for each word m_i (***w*** is the vector representation of the word **m**)

Word Embedding – Vectors

- If user **u** had the tweet $\mathbf{T_u}$ = "I am hungry, sad, and smart", the embedded form would be:

	1	2	3	4	5
I					
Am					
Hungry					
Sad					
And					
Smart					

This table has only 5 dimensions for each word, Google's Word2Vec actually has **300 dimensions**

- The value at each dimension is a "feature" or neuron → dimensions are calculated for every word in a user's matrix of tweets $\mathbf{T_u}$

Word Embedding

- Dimension values are calculated using **embedding models** (such as Google's Word2Vec [CBOW] in this case) which have trained on data from millions of appearances of millions words
- Embedders use Deep Neural Nets (DNNs) to create vectors that give words mathematical equivalences (in the form of arrays of numbers)
- The **result** of embedding:
  - Convert a tweet into a set of **vectors** that represent each word, with the ability to perform calculations between vectors to demonstrate relation between words
- Embedding is a very complex process, and even highly-sophisticated algorithms tend to use publicly-available embedding models (Word2Vec, GloVe, fastText)

Word Embedding – Logic for CBOW

- The CBOW model trains by giving each word a base value, which is determined by a calculation of weights and adjustments. These are calibrated with the training set.
- In order to achieve values of words that are related (ex: “King – Man + Woman = Queen”), the model predicts words based on contextual math (words around the predicted word are summed and modified through weighted calculation).
- This is different from the “skip-gram” model, which predicts context words
- With correct and incorrect predictions, weights and formulas are adjusted for predictor functions.

Word Embedding – Logic for CBOW Example:

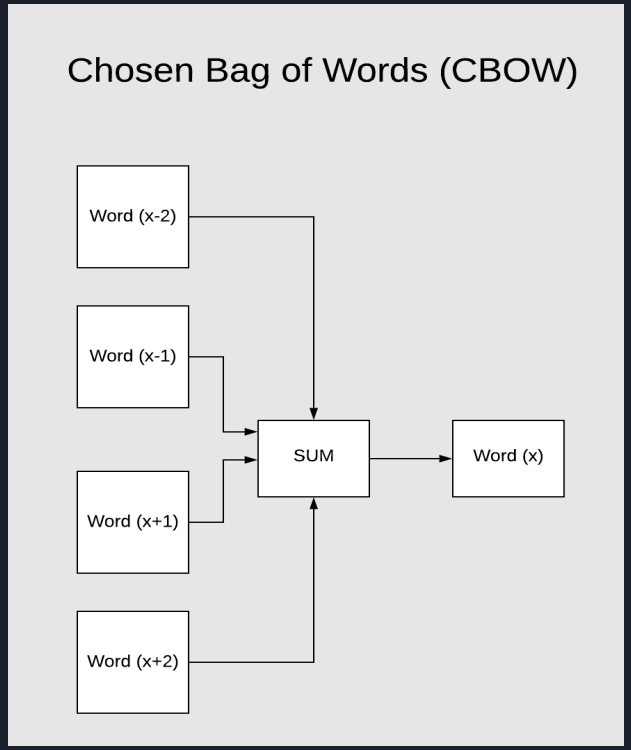


Diagram Created by Waleed Khan

Word Embedding – Demonstration: Training a Word Embedding Model

- My code for creating your own Word Embedding program in python using Google's Word2Vec:

See next slide →

Users > Waleed > Desktop > tester.py > ...

```python
from gensim.models import word2vec
from gensim.models.keyedvectors import KeyedVectors

# Code for loading word2vec vector data
# model = KeyedVectors.load_word2vec_format('/Users/Waleed/Desktop/GoogleNews-vectors-negative300.bin', binary=True)

# Convert .bin to a text file
# model.save_word2vec_format('GoogleNews-vectors-negative300.txt', binary=False)

# Code for reading vector data from a text file --> Terrible idea, too much data in Google's Word2Vec, will take many GB of space
# with open("/Users/Waleed/Desktop/GoogleNews-vectors-negative300.txt") as myfile:
#     head = [next(myfile) for x in xrange(10)]
# print head

# My Self-Created Data to Train on
tweetData = [
    'He was scared of the dark',
    'he was scared of the dark',
    'She was attracted to the light',
    'she was attracted to the light',
    'They ran together in the field',
    'Who knows why it is so dark at night'
]

# Tokenize data
for i, tweet in enumerate(tweetData):
    tokenized= []
    for word in tweet.split(' '):
        word = word.split('.')[0]
        # word = word.lower()
        tokenized.append(word)
    tweetData[i] = tokenized

# Create model with 300 values, workers is the number of cores, window = max distance between word predictions, sg = 1 for skipgram but we are implementing CBOW so we pick 0
model = word2vec.Word2Vec(tweetData, workers = 1, size = 300, min_count = 1, window = 3, sg = 0)

similarWord1 = model.wv.most_similar('He')[0]
similarWord2 = model.wv.most_similar('She')[0]
similarWord3 = model.wv.most_similar('she')[0]
similarWord4 = model.wv.most_similar('he')[0]
he = model.wv.get_vector('he')

print("The word vector for \"he\" is:{}".format(he))
print("The word most similar to \"He\" is:{}".format(similarWord1))
print("The word most similar to \"She\" is:{}".format(similarWord2))
print("The word most similar to \"she\" is:{}".format(similarWord3))
print("The word most similar to \"he\" is:{}".format(similarWord4))
```

Word Embedding – Demonstration: Training a Word Embedding Model

- This program reads “tweet Data” , tokenizes words from the sentences, and trains on the words in the word2Vec trainer

```
14
15  # My Self-Created Data to Train on
16  tweetData = [
17      'He was scared of the dark',
18      'he was scared of the dark',
19      'She was attracted to the light',
20      'she was attracted to the light',
21      'They ran together in the field',
22      'Who knows why it is so dark at night'
23  ]
24
25  # Tokenize data
26  for i, tweet in enumerate(tweetData):
27      tokenized= []
28      for word in tweet.split(' '):
29          word = word.split('.')[0]
30          # word = word.lower()
31          tokenized.append(word)
32      tweetData[i] = tokenized
33
34  # Create model with 300 values, workers is the number of cores, window = max distance between word predictions, sg = 1 for skipgram but we are implementing CBOW so we pick 0
35  model = word2vec.Word2Vec(tweetData, workers = 1, size = 300, min_count = 1, window = 3, sg = 0)
36
```

Word Embedding – Demonstration: Training a Word Embedding Model

- The word embedding model was tasked with finding the most similar words for ”He”, ”She”, “he” , and “she”, and the vector for ‘he’, after training

```
similarWord1 = model.wv.most_similar('He')[0]
similarWord2 = model.wv.most_similar('She')[0]
similarWord3 = model.wv.most_similar('she')[0]
similarWord4 = model.wv.most_similar('he')[0]
he = model.wv.get_vector('he')

print("The word vector for \"he\" is:{}".format(he))
print("The word most similar to \"He\" is:{}".format(similarWord1))
print("The word most similar to \"She\" is:{}".format(similarWord2))
print("The word most similar to \"she\" is:{}".format(similarWord3))
print("The word most similar to \"he\" is:{}".format(similarWord4))
```

(Results on next page) →

Word Embedding – Demo Word2Vec Results

- Results:

The word vector for "he" is: [-1.25429593e-03 2.32844046e-04 1.20008795e-03 ….]
The word most similar to "He" is:('attracted', 0.0860222727060318)
The word most similar to "She" is:('light', 0.122720308601856 23)
The word most similar to "she" is:('why', 0.129128217697143 55)
The word most similar to "he" is:('it', 0.0755527913570404)

- The word vector is a 300-index word embedding, and too long to put here. Contact me for full word vector if interested in the data.

Word Embedding – Demonstration: Using Word2Vec Model on Google News Dataset

- We will now train our Word2Vec model off of a real and large dataset
- We use the Google News Vector dataset (~100 billion words, turned into 300-index vectors using a Deep Neural Net) to embed words
    - Code on next slide→

Users > Waleed > Desktop > tester.py > ...

```python
from gensim.models import word2vec
from gensim.models.keyedvectors import KeyedVectors

# Code for loading word2vec vector data
model = KeyedVectors.load_word2vec_format('/Users/Waleed/Desktop/GoogleNews-vectors-negative300.bin', binary=True)

# Convert .bin to a text file
# model.save_word2vec_format('GoogleNews-vectors-negative300.txt', binary=False)

# Code for reading vector data from a text file --> Terrible idea, too much data in Google's Word2Vec, will take many GB of space
# with open("/Users/Waleed/Desktop/GoogleNews-vectors-negative300.txt") as myfile:
#     head = [next(myfile) for x in xrange(10)]
# print head

# My Self-Created Data to Train on
# tweetData = [
#     'He was scared of the dark',
#     'he was scared of the dark',
#     'She was attracted to the light',
#     'she was attracted to the light',
#     'They ran together in the field',
#     'Who knows why it is so dark at night'
# ]

# Tokenize data
# for i, tweet in enumerate(tweetData):
#   tokenized= []
#   for word in tweet.split(' '):
#       word = word.split('.')[0]
#       # word = word.lower()
#       tokenized.append(word)
#   tweetData[i] = tokenized

# Create model with 300 values, workers is the number of cores, window = max distance between word predictions, sg = 1 for skipgram but we are implementing CBOW so we pick 0
# model = word2vec.Word2Vec(tweetData, workers = 1, size = 300, min_count = 1, window = 3, sg = 0)

similarWord1 = model.wv.most_similar('He')[0]
similarWord2 = model.wv.most_similar('She')[0]
similarWord3 = model.wv.most_similar('she')[0]
similarWord4 = model.wv.most_similar('he')[0]
he = model.wv.get_vector('he')

print("The word vector for \"he\" is:{}".format(he))
print("The word most similar to \"He\" is:{}".format(similarWord1))
print("The word most similar to \"She\" is:{}".format(similarWord2))
print("The word most similar to \"she\" is:{}".format(similarWord3))
print("The word most similar to \"he\" is:{}".format(similarWord4))
```

Word Embedding - Real Demonstration

- This program reads vectors from the Google News dataset
- The word embedding model was then once again tasked with finding the most similar words for ”He”, ”She”, “he” , and “she”, and the vector for ‘he’, after training

Users > Waleed > Desktop > tester.py > ...

```
1   from gensim.models import word2vec
2   from gensim.models.keyedvectors import KeyedVectors
3
4   # Code for loading word2vec vector data
5   model = KeyedVectors.load_word2vec_format('/Users/Waleed/Desktop/GoogleNews-vectors-negative300.bin', binary=True)
```

```
43  print("The word vector for \"he\" is:{}".format(he))
44  print("The word most similar to \"He\" is:{}".format(similarWord1))
45  print("The word most similar to \"She\" is:{}".format(similarWord2))
46  print("The word most similar to \"she\" is:{}".format(similarWord3))
47  print("The word most similar to \"he\" is:{}".format(similarWord4))
```

Results on next page →

Word Embedding – Real Word2Vec Results

- Results:

The word vector for "he" is:[1.92382812e-01 1.27929688e-01 -1.91650391e-02…]
The word most similar to "She" is:(u'He', 0.8116158246994019)
The word most similar to "she" is:(u'her', 0.7834683656692505)
The word most similar to "he" is:(u'He', 0.6712614297866821)

- The word vector is a 300-index word embedding, and too long to put here. Contact me for full word vector if interested in the data.

Word Embedding – Demonstration Observations:

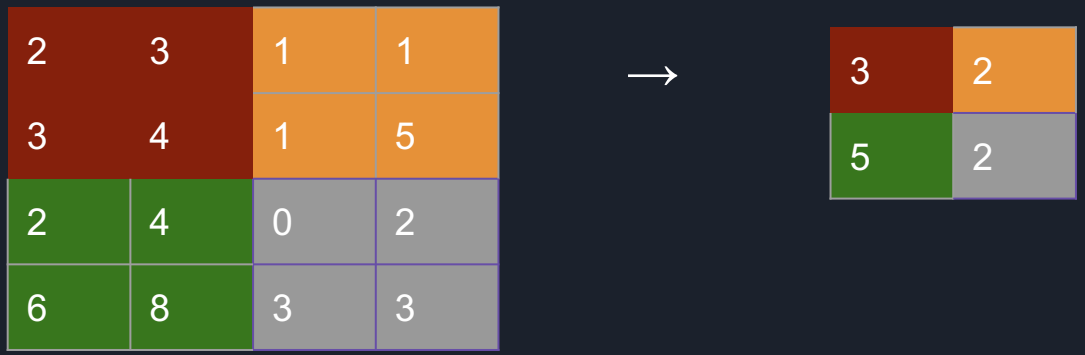
- The model that trained off my self-generated data came up with different most similar words for ”He”, ”She”, “he” , and “she” compared to the model trained off of the Google News dataset.
- This is because of the difference in word values constructed from each dataset. The Deep Neural Network trains by adjusting the words’ values in relation to words around it.
    - Because of this, the results of distance and relationship queries will differ depending on the training data
    - By training a model off of a robot’s set of tweets and comparing it to a model trained off of a google news dataset of millions of human-written sentences, we find the hidden differences of word-relationships in sentence construction through deep learning
        - This is precisely how our CNN will train it’s classifier – by using feature extraction on word embeddings as training data

Convolutional Neural Network - Convolving Layer

- So far, we have embedded each tweet $\mathbf{T_{ui}}$ of user **u** into its own vector, and have matrix $\mathbf{R_i^{(m*w)}}$, which holds **m** words for **w-length embeddings** (w=300 for Google News data)
- We then concatenate these embedded matrices for all tweets $\mathbf{T_u = T_{u1}, T_{u2} \dots T_{un}}$ where: $\mathbf{R_u^{(m*w)} = R_{u1}^{(m*w)}, R_{u2}^{(m*w)}, \dots, R_{un}^{(m*w)}}$
- This matrix is inserted into the **CNN**:
  - The Convolutional layer of the **CNN** has **"filters"** which will iterate over the set of tweet data and map sets of values to other values → This is called "**Convolving**"
    - Each filter has **width** ***k*** there are ***s*** filters such that filters $\mathbf{F = f_1, f_2, \dots, f_s}$
    - Filters have size $\mathbf{f_i}$ such that $\mathbf{f_i \in R^{(k*w)}}$
    - The result after applying mathematical convolution is $\mathbf{C_u \in R^{(n*m-k+1,1)}}$
    - $\mathbf{C_u}$ is an alternate representation of the data, called a "**feature map**" from the filter performing operations on the data
    - The feature map is created when using training data, and compared against when testing
- The resulting feature maps are then **pooled** in the pooled layer

Convolutional Neural Network - Pooling Layer And Flattening

- Now that we have generated multiple feature maps from filtering the data for each user, we need to do "**Pooling**" on the data set in order to merge the feature maps for key features
- This can be done in multiple ways, such as by "Average Pooling":

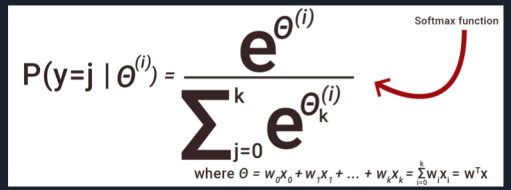


- We continue to pool, until we are "**flattening**" - converting our data into a 1-dimensional array for the next layer of processing

Chart created by Waleed Khan

Convolutional Neural Network - Classifying

- Now that we have generated a single vector of data containing dimension data that relates to a user's tweet:
  - We want to put this data through the "**Softmax Layer**", in which we convert the data into a probability between 0 to 1 (for non-bot and bot detection)
    - The softmax function safely distributes probability between 0 and 1 for all items in a set



    - In the "training" phase of the convolutional neural network, we will feed these vectors as sample data along with the answer to whether this data is from a human or social bot ("bot" or "not bot")
      - CNN will adjust weights and values as needed in order to increase algorithm accuracy based on the given classification of "bot" or "not bot"

Formula provided by Hamza Mahmood

Convolutional Neural Network – Example Implementation

- Using Keras and Python, we can create a CNN
  - Simple Cat and Dog classifier that uses image data to train a classifier
- Instructions on Keras usage provided by Abishek Chatterjee
- Next slide →

Training Code:

```
path="train"
new_path_train="/train/train_new_resized_96"
label=[]
data1=[]
counter=0
for file in os.listdir("train"):
    image_data=cv2.imread(os.path.join(path,file), cv2.IMREAD_GRAYSCALE)
    image_data=cv2.resize(image_data,(96,96))
    if file.startswith("cat"):
        label.append(0)
    elif file.startswith("dog"):
        label.append(1)
    try:
        data1.append(image_data/255)
    except:
        label=label[:len(label)-1]
    counter+=1
    if counter%1000==0:
        print (counter," image data retreived")
from keras import Sequential
from keras.layers import Dense,MaxPooling2D,Conv2D,Flatten,Dropout
model=Sequential()
model.add(Conv2D(kernel_size=(3,3),filters=3,input_shape=(96,96,1),activation="relu"))
model.add(Conv2D(kernel_size=(3,3),filters=10,activation="relu",padding="same"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(kernel_size=(3,3),filters=3,activation="relu"))
model.add(Conv2D(kernel_size=(5,5),filters=5,activation="relu"))
model.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))
model.add(Conv2D(kernel_size=(2,2),strides=(2,2),filters=10))
model.add(Flatten())
model.add(Dropout(0.3))
model.add(Dense(100,activation="sigmoid"))
model.add(Dense(1,activation="sigmoid"))
model.summary()
model.compile(optimizer="adadelta",loss="binary_crossentropy",metrics=["accuracy"])
import numpy as np
data1=np.array(data1)
print(data1)
print (data1.shape)
data1=data1.reshape((data1.shape)[0],(data1.shape)[1],(data1.shape)[2],1)
data1=data1/255
labels=np.array(label)
print (data1.shape)
print (labels.shape)
model.fit(data1,labels,validation_split=0.25,epochs=2,batch_size=10)
model.save_weights("model.h5")
```

Testing Code:

```
test_data=[]
id=[]
counter=0
for file in os.listdir("test1"):
    image_data=cv2.imread(os.path.join("test1",file), cv2.IMREAD_GRAYSCALE)
    try:
        image_data=cv2.resize(image_data,(96,96))
        test_data.append(image_data/255)
        id.append((file.split("."))[0])
    except:
        print ("excepted");
    counter+=1
    if counter%1000==0:
        print (counter," image data retreived")
test_data1=np.array(test_data)
print (test_data1.shape)
test_data1=test_data1.reshape((test_data1.shape)[0],(test_data1.shape)[1],(test_data1.shape)[2],1)
dataframe_output=pd.DataFrame({"id":id})
predicted_labels=model.predict(test_data1)
predicted_labels=np.round(predicted_labels,decimals=2)
labels=[1 if value>0.5 else 0 for value in predicted_labels]
dataframe_output["label"]=labels
dataframe_output.to_csv("results.csv",index=False)
```

Convolutional Neural Network – Example Implementation

- We can see the convolution process in code, during which we convolute, pool, and flatten data repeatedly
- This generates features maps of the data that our program can analyze
- The sigmoid function allows us to predict on a scale from 0 to 1 whether the subject is a cat or dog (cat = 0, dog = 1)

```
from keras import Sequential
from keras.layers import Dense,MaxPooling2D,Conv2D,Flatten,Dropout
model=Sequential()
model.add(Conv2D(kernel_size=(3,3),filters=3,input_shape=(96,96,1),activation="relu"))
model.add(Conv2D(kernel_size=(3,3),filters=10,activation="relu",padding="same"))
model.add(MaxPooling2D(pool_size=(2,2),strides=(2,2)))
model.add(Conv2D(kernel_size=(3,3),filters=3,activation="relu"))
model.add(Conv2D(kernel_size=(5,5),filters=5,activation="relu"))
model.add(MaxPooling2D(pool_size=(3,3),strides=(2,2)))
model.add(Conv2D(kernel_size=(2,2),strides=(2,2),filters=10))
model.add(Flatten())
model.add(Dropout(0.3))
model.add(Dense(100,activation="sigmoid"))
model.add(Dense(1,activation="sigmoid"))
model.summary()
model.compile(optimizer="adadelta",loss="binary_crossentropy",metrics=["accuracy"])
import numpy as np
data1=np.array(data1)
print(data1)
print (data1.shape)
data1=data1.reshape((data1.shape)[0],(data1.shape)[1],(data1.shape)[2],1)
data1=data1/255
labels=np.array(label)
print (data1.shape)
print (labels.shape)
model.fit(data1,labels,validation_split=0.25,epochs=2,batch_size=10)
model.save_weights("model.h5")
```

Convolutional Neural Network

- In summary, using the embedded values corresponding to the tweet data, we can use math operations to generate feature maps that **detect** the sentence structure of the speech data through operations on embedded values of words
- This gives our CNN the ability to train its classifier on the sentence structure of billions of words, learning both human-like and robot-like word relation.
    - However, the DeBD algorithm requires more features before feeding the feature data to the classifier of the CNN
- An additional layer of an **LSTM** will be added to analyze the temporal (time-based) content of these tweets.
    - The LSTM data will be fed to the CNN as additional feature map data, which will then train the CNN classifier

Temporal Feature Extraction Layer using LSTM

- In order to prepare and analyze temporal (time-based) content we:
  - Divide the twitter behavior (**T**) of user **u** into every minute for every day **d** such that:
    - $\mathbf{T_u = [t_1, t_2, \ldots, t_{1440}]}$
  - Count the total number of retweets every minute $\mathbf{n_i}$ (for N tweets on day **D** such that:
    - $\mathbf{rt = n_1 + n_2 + \ldots + n_N}$
  - The total statistic of retweeting on day **d** can be expressed such that:
    - $\mathbf{RT_u = rt_1, rt_2, \ldots, rt_D}$
    - $\mathbf{RT_u \in R^{D*1440}}$
  - This is done similarly for comments, mentions, URLs, and hashtags relating to the user, defined as:
    - Comments → $COM_u = com_1, com_2, \ldots, com_D$
    - Mentions → $MEN_w = men_1, men_2, \ldots, men_D$
    - URLs → $URL_u = url_1, url_2, \ldots, url_D$
    - Hashtags → $HT_u = ht_1, ht_2, .., ht_D$

Temporal Feature Extraction Layer (LSTM)

- After extracting these temporal features, we feed them into our LSTM as input
- LSTM has a "**cell**" that stores, memory with 3 regulating gates:
  - Input Gate (Gives input)
  - Forget Gate (Manages memory stored)
  - Output Gate (Gives output)

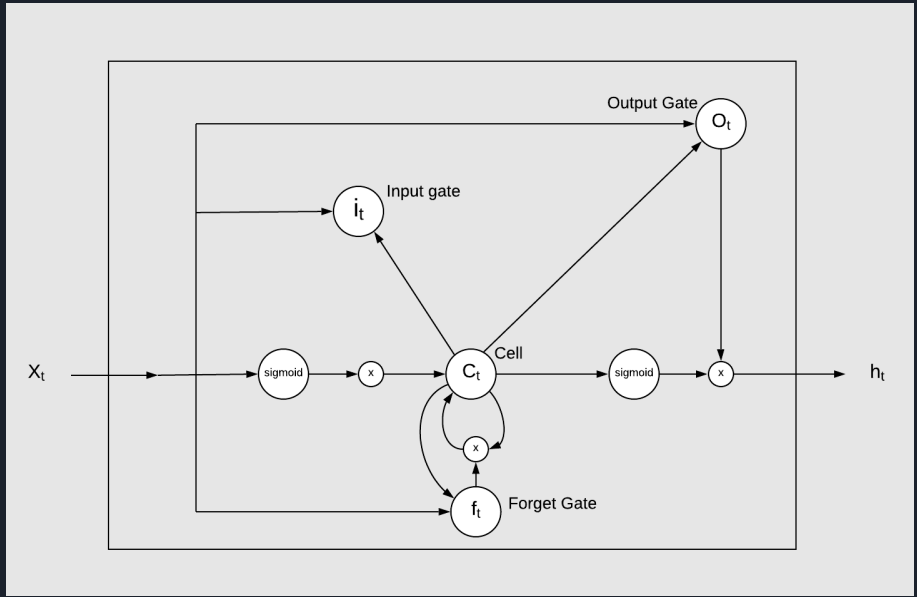


Diagram Created by Waleed Khan

Temporal Feature Extraction Layer (LSTM)

- After every gate, an ‘x’ indicates element multiplication between two inputs in order to received an adjusted result
- The Cell stores a given portion of memory for recall, and uses this memory to adjust outputs based on weights
- Weights are used and adjusted during each gate operation, based on previous inputs into the LSTM
- The resulting output of the LSTM is then inserted into the CNN

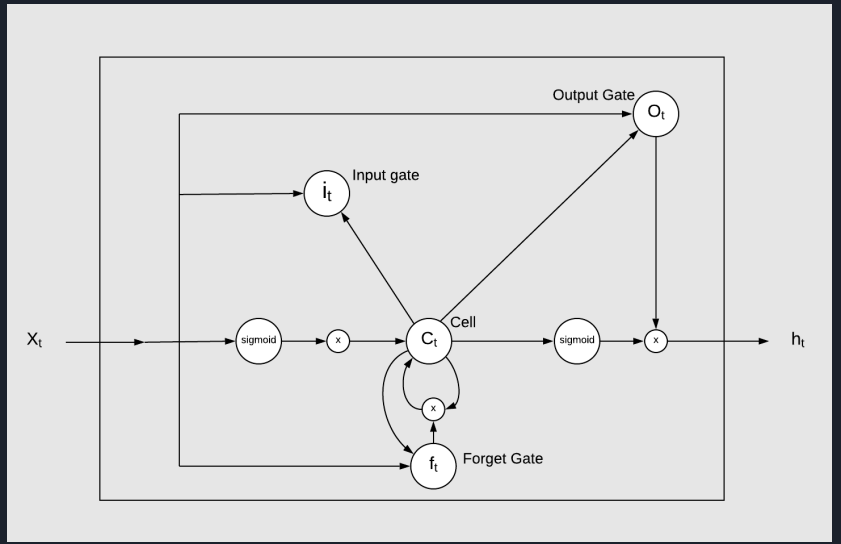


Diagram Created by Waleed Khan

Temporal Feature Extraction LSTM → CNN

- With the output of the LSTM, we now have adjusted values for the following:
  - Comments → $COM_u = com_1, com_2, \ldots, com_D$
  - Mentions → $MEN_w = men_1, men_2, \ldots, men_D$
  - URLs → $URL_u = url_1, url_2, \ldots, url_D$
  - Hashtags → $HT_u = ht_1, ht_2, .., ht_D$
- We use this as input into the CNN, where it will extract feature maps for this data, pool the data, flatten the data, and then output its prediction after placing the result through a softmax layer
  - In the “training” phase of the convolutional neural network, we will feed the flattened vectors as sample data along with the answer to whether this data is from a human or social bot (“bot” or “not bot”)
    - CNN will adjust weights and values as needed in order to increase algorithm accuracy based on the given classification of “bot” or “not bot”

Fusing Layer - DeBD Algorithm

- Now that we have extraction of both Joint Content and Temporal Features, we are able to run a respective ***Word Embedding → CNN*** and ***LSTM → CNN*** processes in which we can determine whether a user is a bot or not. We will now join these.
- In order to complete the **DeBD Algorithm**, we fuse the results of the CNN feature map extractions from both processes into one equation:
  - $U = W_1 * RT_u + W_2 * COM_u + W_3 * MEN_u + W_4 * URL + W_5 * HT_u \oplus C_u$
    - $\oplus$ denotes vector connection
    - W_i is a weight, adjusted by a neural network during classification training
      - This can be the LSTM in which it will take each value and adjust weights based on previous results
    - The resulting **U** is input into the CNN
      - The CNN will do feature extraction, pooling, and flattening, on this value before passing it through the softmax layer and classifier which gives our **final prediction of a 0 or a 1**
        - In the training phase these values are used to teach the classifier so it can adjust its independent weights

Sources

(Main Article):

- Ping, Heng, and Sujuan Qin. “A Social Bots Detection Model Based on Deep Learning Algorithm.” *2018 IEEE 18th International Conference on Communication Technology (ICCT)*, 2018, doi:10.1109/icct.2018.8600029.

- Ferrara, Emilio, et al. “The Rise of Social Bots.” *ArXiv.org*, Cornell University, 6 Mar. 2017, https://arxiv.org/abs/1407.5225, https://cacm.acm.org/magazines/2016/7/204021-the-rise-of-social-bots/fulltext.

- Jeong, Jiwon. “The Most Intuitive and Easiest Guide for CNN.” *Medium*, Towards Data Science, 17 July 2019, towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480.

- Mahmood, Hamza. “Softmax Function, Simplified.” *Medium*, Towards Data Science, 26 Nov. 2018, towardsdatascience.com/softmax-function-simplified-714068bf8156.

Sources

- Jeong, Jiwon. “Long Short-Term Memory.” *Wikipedia*, Wikimedia Foundation, 1 Dec. 2019, en.wikipedia.org/wiki/Long_short-term_memory#/media/File:Peephole_Long_Short-Term_Memory.svg.
- “Dogs vs. Cats.” *Kaggle*, 2013, www.kaggle.com/c/dogs-vs-cats/data.
- Chatterjee, Abishek. “cat_dog_Try.” *Kaggle*, Kaggle, 30 Sept. 2018, www.kaggle.com/abhishekrock/cat-dog-try.
- Google. “Google Code Archive - word2vec.” *Google*, Google, 2013, code.google.com/archive/p/word2vec/.
- Brownlee, Jason. “What Are Word Embeddings for Text?” *Machine Learning Mastery*, 7 Aug. 2019, machinelearningmastery.com/what-are-word-embeddings/.