Opinion Mining Of Spanglish And Hinglish Text Data Using BERT And Transformer Model

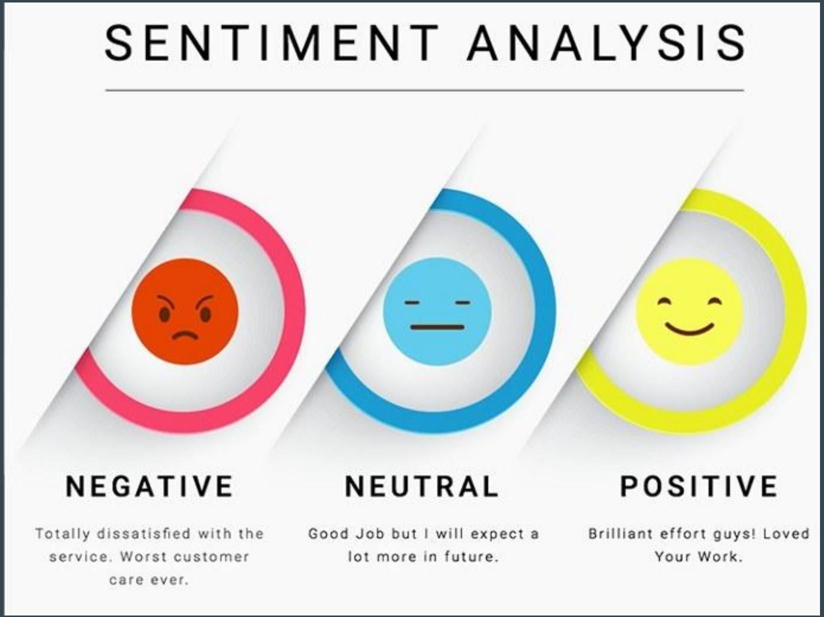
• • •

Aviral Bajpai SRIP-2020 Summer Research Intern

OPINION MINING

What is Opinion Mining ?

Opinion Mining is the interpretation and classification of emotions (positive, negative and neutral) within text data using text **analysis** techniques.



Dataset

Dataset contains 14000 tweets for testing which are multilingual tweets and test dataset is of length 3000 .And they are to be classified into 3 categories .Namely (“positive”, “neutral” , “negative)

```
print(len(df['tweets']))
print(df['tweets'][0])
print(df['tweets'][1])
print()

print(len(test_tweets))
print(test_tweets[0])
print(test_tweets[1])
print()

print(len(testlabels))
print(testlabels['Sentiment'][0])
```

```
14000
nen á vist bolest vztek smutek zmatek osam ě lost beznad ě j a nakonec jen klid asi takhle vypad á m ů j life .
haan yaar neha pensive_face pensive_face kab karega woh post loudly_crying_face usne na sach mein photoshoot karna chahiye phir woh post karega .

3000
@ 454dkhan @ heisunberg _ agr kse ko itni importantce chaeay ni tou ðŸ˜…
logon ko alloo pyaz tomator me toh allah pak ka naam nazar aa jata hai pr aankhon k samne allah pak ke bande nazar â€¦ https // t . co / hbg7zs0viy

3000
0
```

Natural Language Processing Algorithms For Opinion Mining

1. Bag Of Words Model
2. Seq2Seq Model
3. Transformer Model(BERT with attention Mechanism)

Algo 1 - Applying Bag Of Words Model to the Training Dataset

Bag of Words Example

Document 1

The quick brown fox jumped over the lazy dog's back.

Document 2

Now is the time for all good men to come to the aid of their party.

Term	Document 1	Document 2
aid	0	1
all	0	1
back	1	0
brown	1	0
come	0	1
dog	1	0
fox	1	0
good	0	1
jump	1	0
lazy	1	0
men	0	1
now	0	1
over	1	0
party	0	1
quick	1	0
their	0	1
time	0	1

Stopword List

for
is
of
the
to

Multi Layer Neural Network For The Tokenized Tweets

```
import keras
from keras.models import Sequential
from keras.layers import Dense
model=Sequential()
model.add(Dense(7000,input_dim=x_strat.shape[1],activation='sigmoid'))
model.add(Dense(3000,activation='sigmoid'))
model.add(Dense(1000,activation='sigmoid'))
model.add(Dense(3,activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
return model
```

Results On Test Labels

Results on Test Labels -

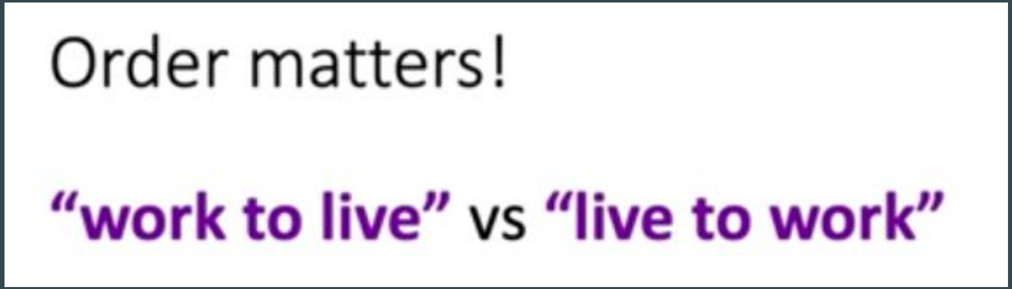
Accuracy - 58%

Reason -

1. Order of words does not matter

in Bag of words model

2. Most Of the matrix is sparse



Algo-2 RNN's (Seq2Seq Models)

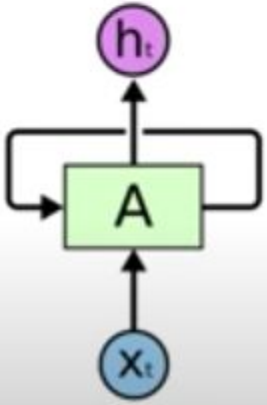
RNN: a new approach

How to calculate $f(x_1, x_2, x_3, \ldots, x_N)$

A for-loop in math.

$$H_{i+1} = A(H_i, x_i)$$

$$f(\vec{x}) = H_N$$



Problems with RNN’s

Vanishing & Exploding Gradients

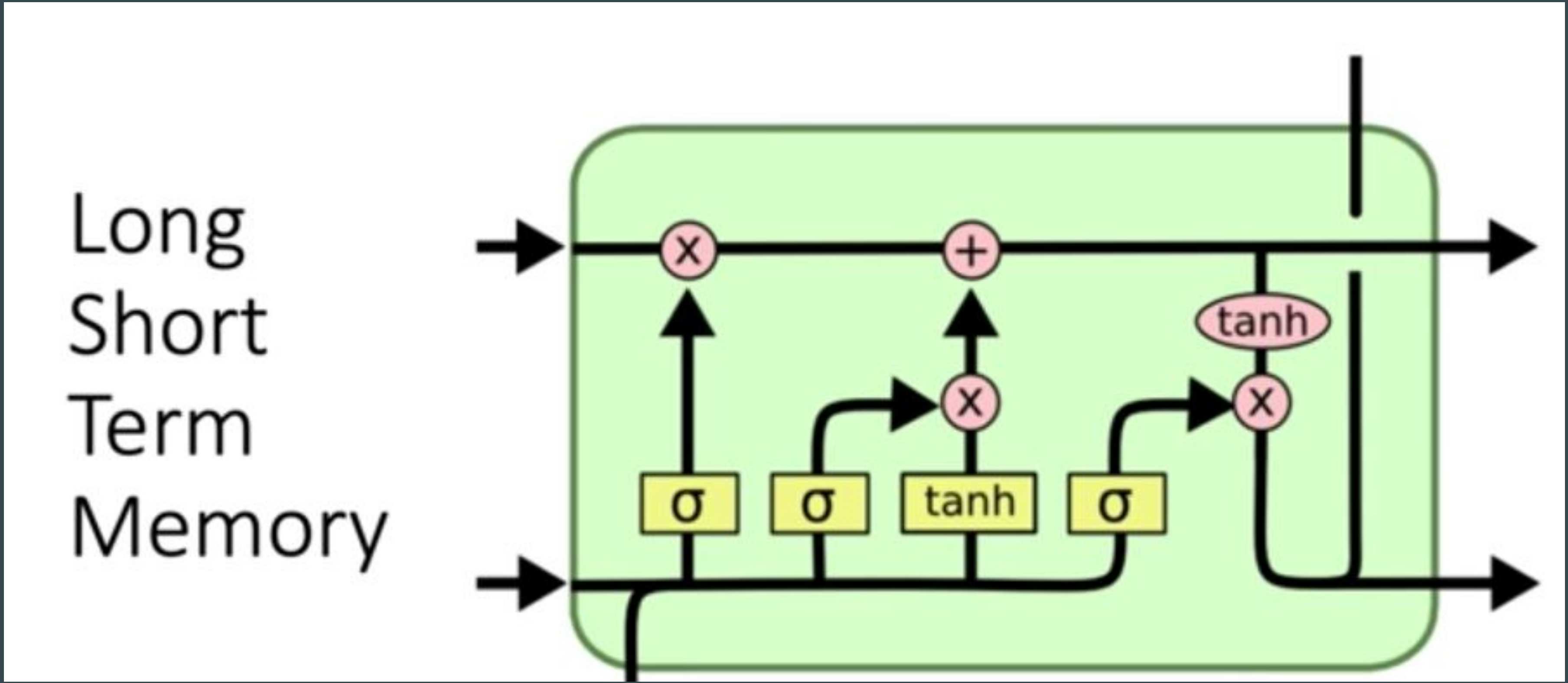
$$H_{i+1} = A(H_i, x_i)$$

$$H_3 = A(A(A(H_0, x_0), x_1), x_2)$$

$$A(H, x) := \mathbf{W}x + \mathbf{Z}H$$

$$H_N = \mathbf{W}^N x_0 + \mathbf{W}^{N-1} x_1 + \ldots$$

Introduction Of LSTM



Results of LSTM on Test Labels

Model Used

```
embed_dim = 128
lstm_out = 196

model = Sequential()
model.add(Embedding(max_fatures, embed_dim,input_length = X.shape[1]))
model.add(SpatialDropout1D(0.4))
model.add(LSTM(lstm_out, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(3,activation='softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer='adam',metrics = ['accuracy'])
print(model.summary())
```

Accuracy Achieved on test labels -

63.4%

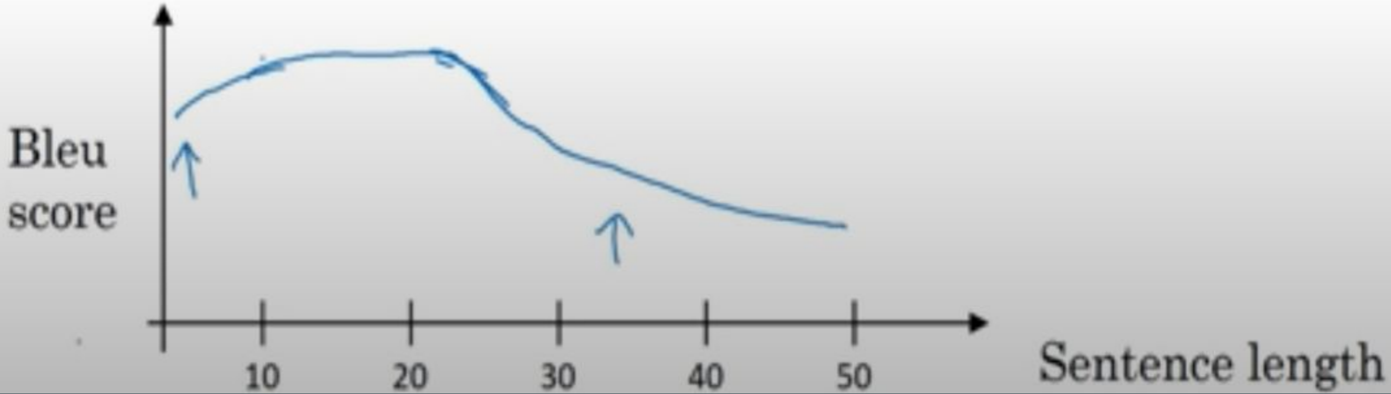
```
sklearn.metrics.accuracy_score(testlabels['Sentiment'],output)*100

63.43333333333333
```

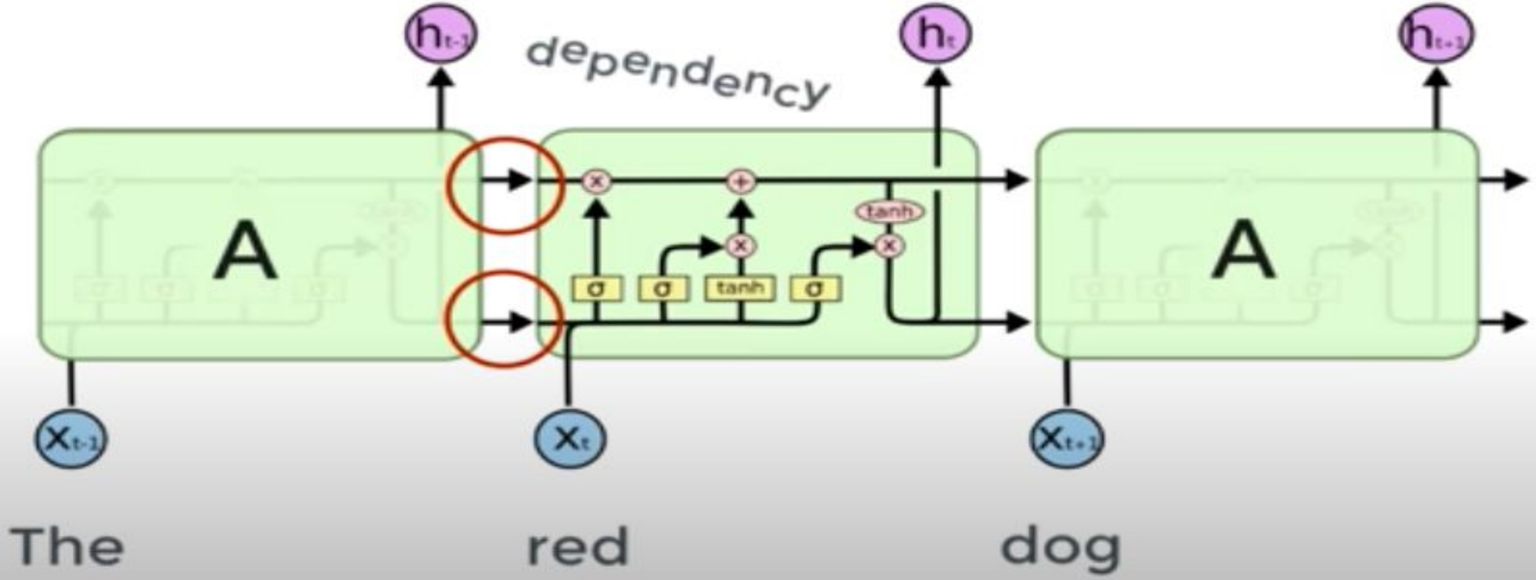
Reasons For Low Accuracy

LSTM's limitations

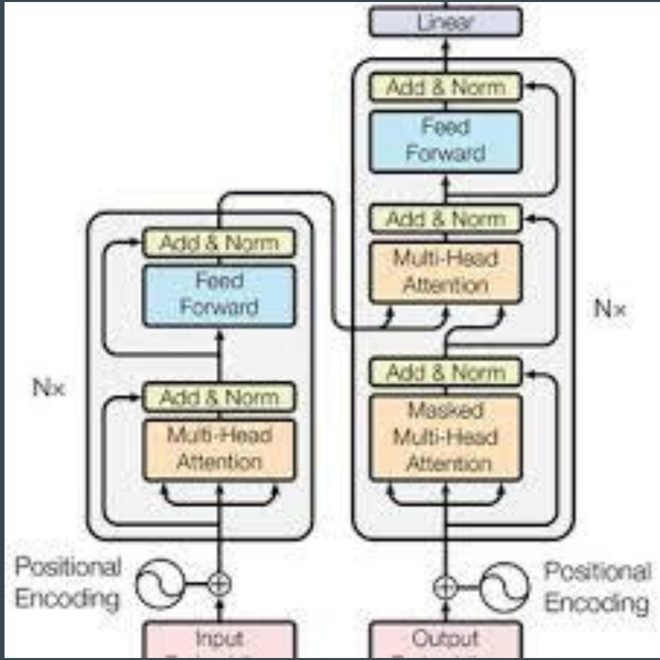
- Difficult to train
- Very long gradient paths
  - LSTM on 100-word doc has gradients like 100-layer network
- Transfer learning never really worked



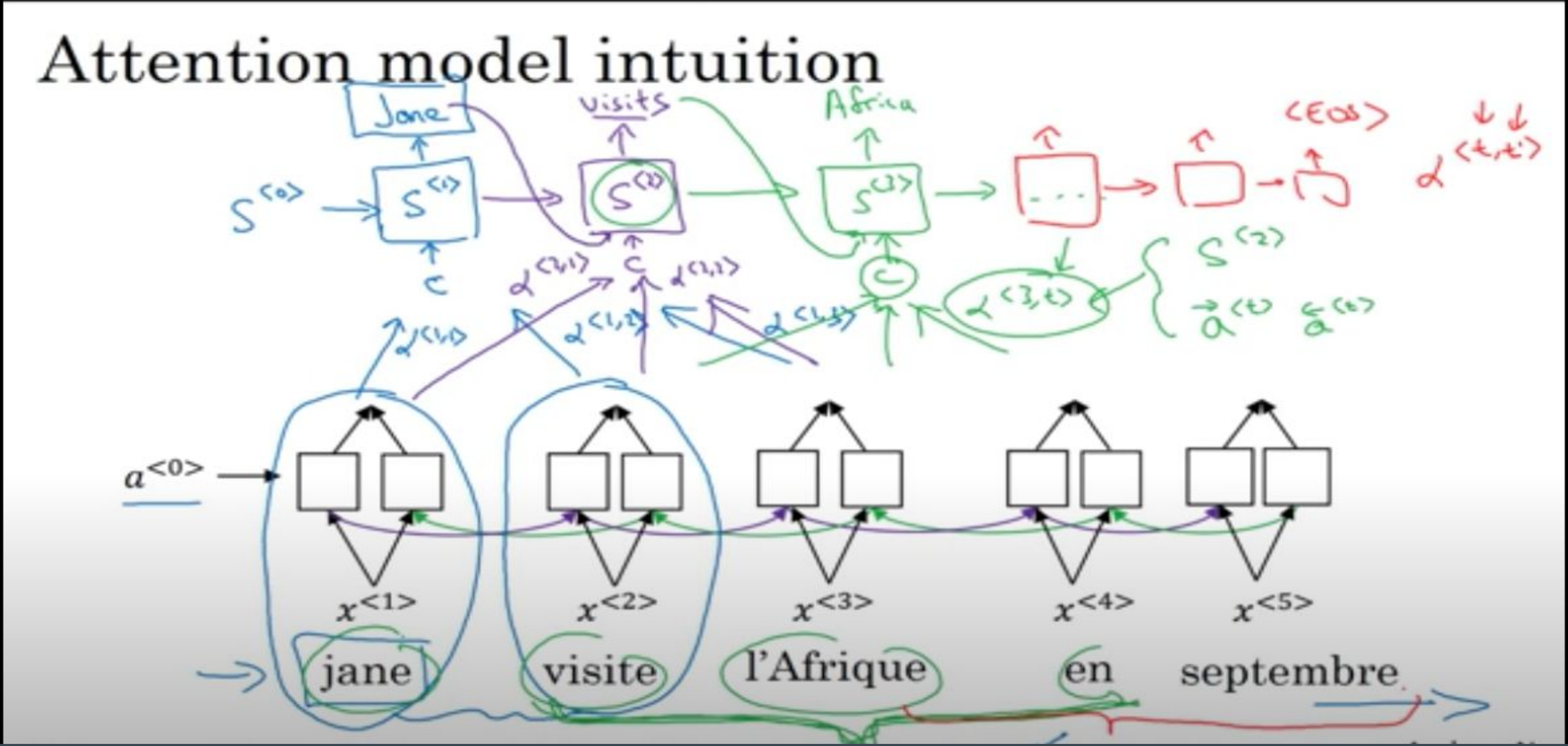
LSTM Networks



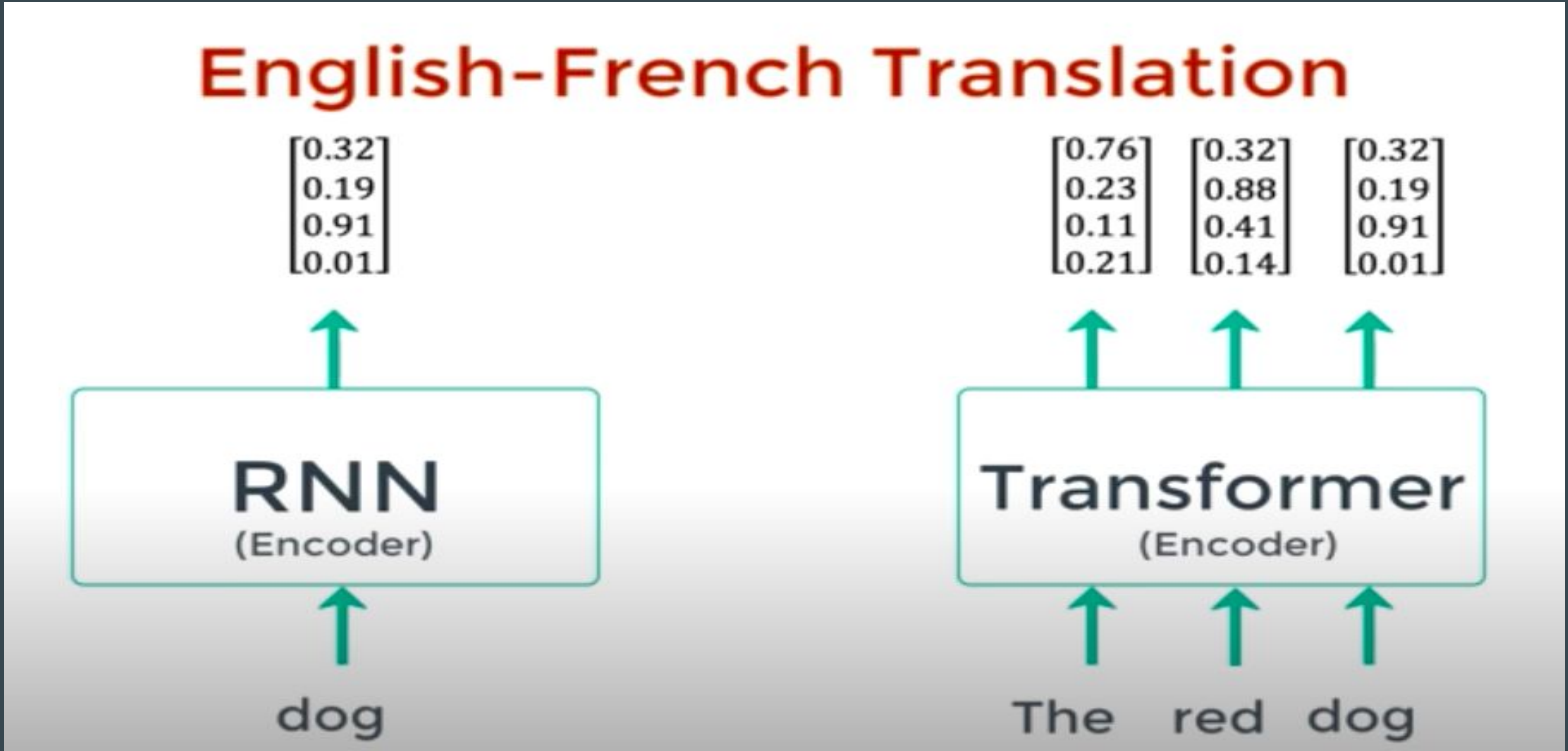
Algo 3 - Attention Mechanism(BERT Model)



Attention Intuition



How is Transformer Different ?



Parts Of Transformer Architecture

<u>Positional Encoder</u> :vector that gives context based on position of word in sentence

AJ's dog is a cutie → Position 2

AJ looks like a dog → Position 5

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

<u>Positional Encoder</u> :vector that gives context based on position of word in sentence

$$\begin{bmatrix}0.37\\0.99\\0.01\\0.08\end{bmatrix} + \boxed{\text{Positional Encoding}} \rightarrow \begin{bmatrix}0.42\\0.84\\0.12\\0.81\end{bmatrix}$$

Embedding of "Dog"

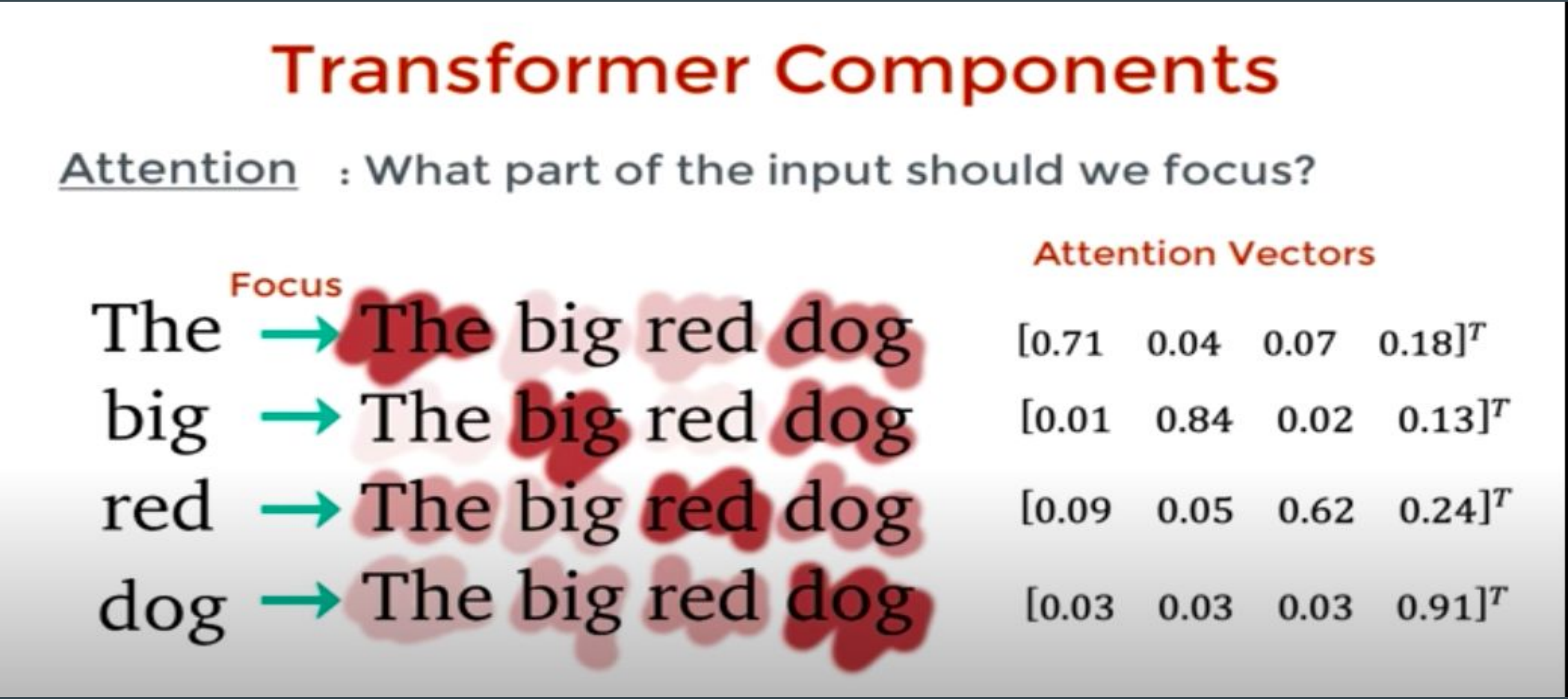
Vector Encoding of position in sentence

Embedding of Dog (with context info)

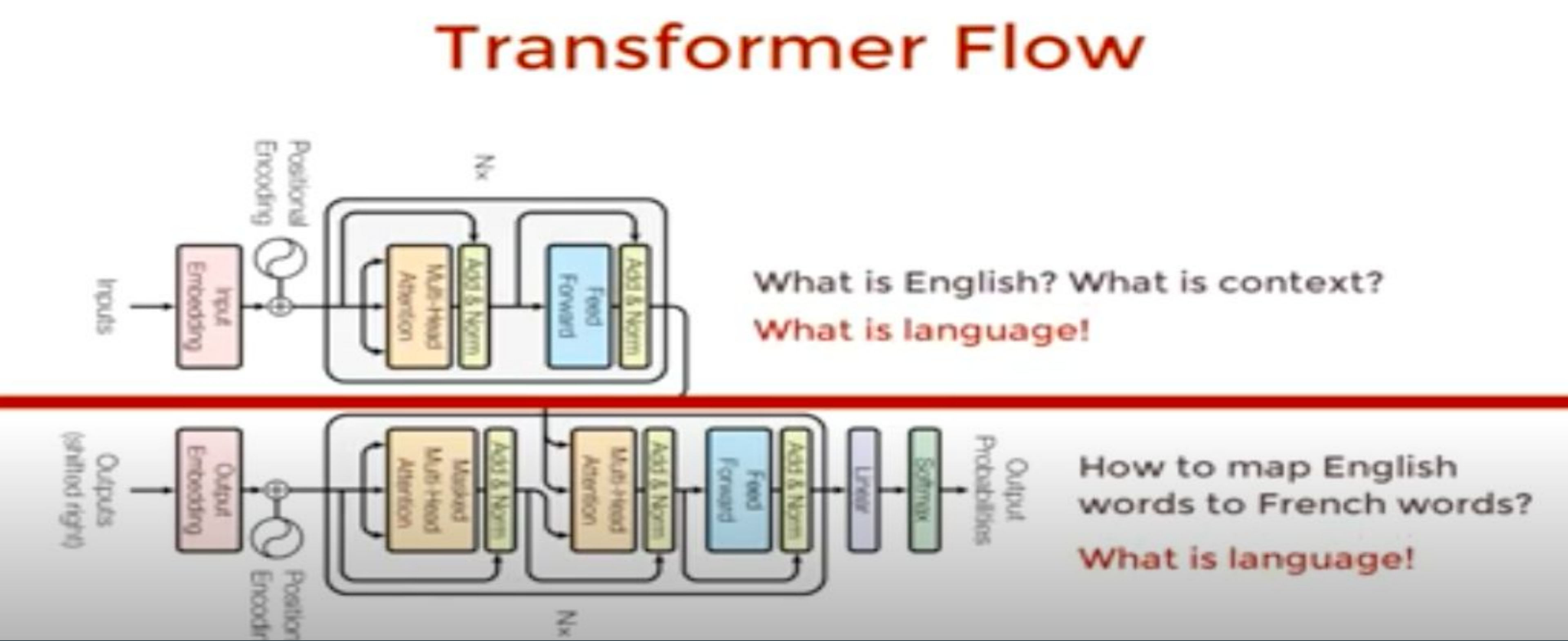
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Attention Vectors

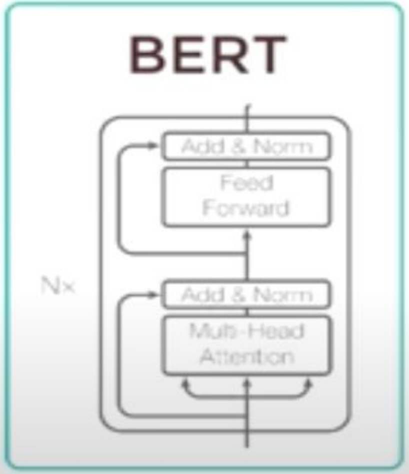


Encoders And Decoders Flow



Implementing Pre Trained Bert Model

Bidirectional Encoder Representation from Transformers



Problems to Solve

- Neural Machine Translation
- Question Answering
- Sentiment Analysis
- Text summarization

Needs Language understanding

How to solve Problems

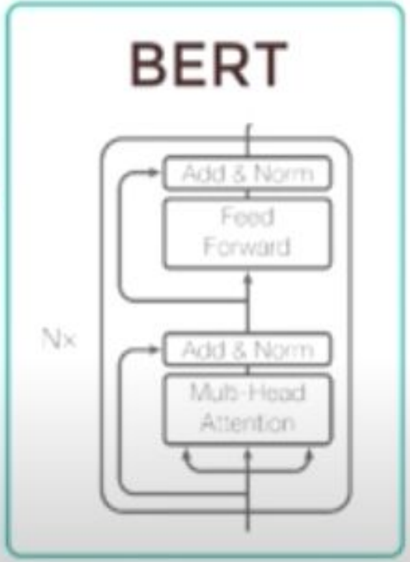
- Pretrain BERT to understand langauge
- Fine tune BERT to learn specific task

Pretraining BERT MODEL

Pretraining (Pass 1) : "What is language? What is context?"

Masked Language Model (MLM)

The [MASK1] brown fox [MASK2] over the lazy dog.



[MASK1] = quick
[MASK2]= jumped

Next Sentence Prediction (NSP)

A: Ajay is a cool dude.
B: He lives in Ohio

Yes. Sentence B follows sentence A

Building Our Own BERT Classifier

BERT Tokenizer

To feed our text to BERT, it must be split into tokens, and then these tokens must be mapped to their index in the tokenizer vocabulary.

The tokenization must be performed by the tokenizer included with BERT--the below cell will download this for us. We'll be using the "uncased" version here.

```
[ ]  from transformers import BertTokenizer

     # Load the BERT tokenizer.
     print('Loading BERT tokenizer...')
     tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

Let's apply the tokenizer to one sentence just to see the output.

```
print(' Original: ', sentences[0])

# Print the sentence split into tokens.
print('Tokenized: ', tokenizer.tokenize(sentences[0]))

# Print the sentence mapped to token ids.
print('Token IDs: ', tokenizer.convert_tokens_to_ids(tokenizer.tokenize(sentences[0])))
```

```
 Original:  nen á vist bolest vztek smutek zmatek osam ě lost beznad ě j a nakonec jen klid asi takhle vypad á m ů j life .
Tokenized:  ['ne', '##n', 'a', 'vis', '##t', 'bo', '##les', '##t', 'v', '##z', '##tek', 'sm', '##ute', '##k', 'z', '##mate', '##k', 'os', '##am', 'e', 'lost
Token IDs:  [11265, 2078, 1037, 25292, 2102, 8945, 4244, 2102, 1058, 2480, 23125, 15488, 10421, 2243, 1062, 8585, 2243, 9808, 3286, 1041, 2439, 2022, 2480,
```

Preprocessing tokenized tweets

STEPS-

1. Add special tokens to the start and end of each sentence([SEP],[CLS]).

2. Pad & truncate all sentences to a single constant length.

3. Explicitly differentiate real tokens from padding tokens with the "attention mask".

Encoding Tweets For Training

```
# For every sentence...
for sent in sentences:
    # `encode_plus` will:
    #   (1) Tokenize the sentence.
    #   (2) Prepend the `[CLS]` token to the start.
    #   (3) Append the `[SEP]` token to the end.
    #   (4) Map tokens to their IDs.
    #   (5) Pad or truncate the sentence to `max_length`
    #   (6) Create attention masks for [PAD] tokens.
    encoded_dict = tokenizer.encode_plus(
                        sent,
                        truncation=True,                # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 64,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',
                             # Return pytorch tensors.
                   )

    # Add the encoded sentence to the list.
    input_ids.append(encoded_dict['input_ids'])

    # And its attention mask (simply differentiates padding from non-padding).
    attention_masks.append(encoded_dict['attention_mask'])
```

Creation Of DataLoader Class

We'll also create an iterator for our dataset using the torch DataLoader class. This helps save on memory during training because, unlike a for loop, with an iterator the entire dataset does not need to be loaded into memory.

```
[ ] from torch.utils.data import DataLoader, RandomSampler, SequentialSampler

    # The DataLoader needs to know our batch size for training, so we specify it
    # here. For fine-tuning BERT on a specific task, the authors recommend a batch
    # size of 16 or 32.
    batch_size = 32

    # Create the DataLoaders for our training and validation sets.
    # We'll take training samples in random order.
    train_dataloader = DataLoader(
                train_dataset,  # The training samples.
                sampler = RandomSampler(train_dataset), # Select batches randomly
                batch_size = batch_size # Trains with this batch size.
            )

    # For validation the order doesn't matter, so we'll just read them sequentially.
    validation_dataloader = DataLoader(
                val_dataset, # The validation samples.
                sampler = SequentialSampler(val_dataset), # Pull out batches sequentially.
                batch_size = batch_size # Evaluate with this batch size.
            )
```

Defining Model

```
from transformers import BertForSequenceClassification, AdamW, BertConfig
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", # Use the 12-layer BERT model, with an uncased vocab.
    num_labels = 3, # The number of output labels--2 for binary classification.
                    # You can increase this for multi-class tasks.
    output_attentions = False, # Whether the model returns attentions weights.
    output_hidden_states = False, # Whether the model returns all hidden-states.

)
model.cuda()
```

Results On Test Labels

70%

```
f1_score(testlabels['Sentiment'], pred_labels_i, average="weighted")*100
```

```
70.00648014848848
```

Reasons For Better Results using BERT

1. **Quicker Development**
    - First, the pre-trained BERT model weights already encode a lot of information about our language. As a result, it takes much less time to train our fine-tuned model - it is as if we have already trained the bottom layers of our network extensively and only need to gently tune them while using their output as features for our classification task. In fact, the authors recommend only 2-4 epochs of training for fine-tuning BERT on a specific NLP task (compared to the hundreds of GPU hours needed to train the original BERT model or a LSTM from scratch!).
2. **Less Data**
    - In addition and perhaps just as important, because of the pre-trained weights this method allows us to fine-tune our task on a much smaller dataset than would be required in a model that is built from scratch. A major drawback of NLP models built from scratch is that we often need a prohibitively large dataset in order to train our network to reasonable accuracy, meaning a lot of time and energy had to be put into dataset creation. By fine-tuning BERT, we are now able to get away with training a model to good performance on a much smaller amount of training data.
3. **Better Results**
    - Finally, this simple fine-tuning procedure (typically adding one fully-connected layer on top of BERT and training for a few epochs) was shown to achieve state of the art results with minimal task-specific adjustments for a wide variety of tasks: classification, language inference, semantic similarity, question answering, etc. Rather than implementing custom and sometimes-obscure architetures shown to work well on a specific task, simply fine-tuning BERT is shown to be a better (or at least equal) alternative.