

Data 586 Project Report

Applications of Deep learning and neural networks in natural language processing tasks of text classification, chat bots and QA systems.

Swapil Paliwal and Vinay Nori

Main Objectives :

1. Getting familiar with CNN and RNN architectures while solving some basic NLP problems.
2. Getting introduced to applications of attention models like BERT in areas of text classification, QA systems and chat bots.

In this regard we decided to do a few small projects listed below.

1. Hate speech and toxic comments classification using

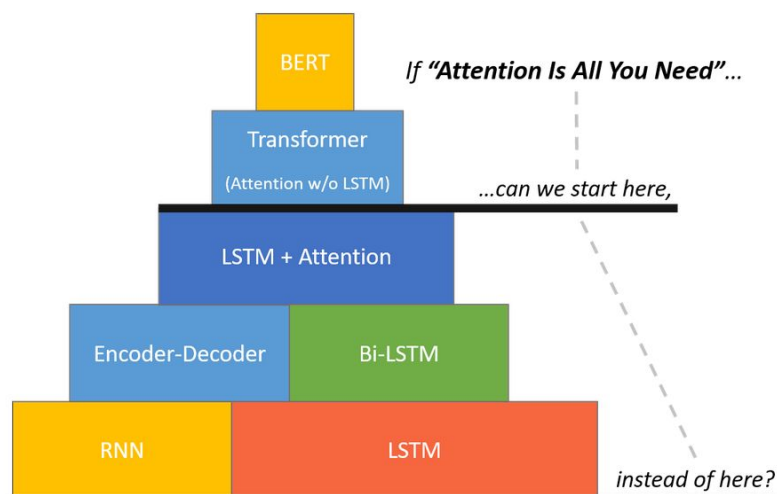
- 2D CNN on Fasttext based word embeddings.
- KimCNN on BERT based fine tuned embeddings using bert encoder and CNN as decoder.
- BERT based encoder decoder architecture.

The main goal was to learn about development of a basic CNN based classifier that can be trained using word embeddings like ones from FastText and fine tuned BERT model. We decided to also leverage the BERT based encoder-decoder architecture to solve this task without using CNN's as a comparison.

2. Building chatbots to mimic movie corpus dialogues and creation of Question Answering system to answer COVID-19 related questions.

- Chatbot was built using RNN, LSTM and GRU based models.
- QA system was built by refining a pretrained - ROBERT Model on COVID-QA corpus using SQUAD-2 style formatting.

We were basically trying to learn the basics of implementing RNN, CNN etc. but were also inspired by attention models recent glories in natural language processing tasks and the diagram below summarizes our objective of using BERT based models.



Project 1 : Hate speech and toxic comments classification

Introduction to Dataset :

The dataset is obtained from a kaggle competition for toxic comment classification.

Link- <https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge>.

We are provided with a large number of Wikipedia comments which have been labeled by human raters for toxic behavior. The types of toxicity are:

- toxic
- severe_toxic
- obscene
- threat
- insult
- identity_hate

The objective is to create a model which predicts a probability of each type of toxicity for each comment. We have 4 files named train.csv, test.csv, test_labels.csv and sample_submission.csv. train.csv is the main file used for training and evaluating the models.

Sample records in train.csv file are as below and this is a classic problem of multiple label classification.

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\n\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

There are a total of 159571 rows in the training set.

Total columns are 8, 6 of which are binary columns containing labels for toxicity etc. and two others are comment_text and id for the comment.

There are 16225 rows where at least one of the six labels have a value of > 0 and 143346 rows where all the six labels have values = 0.

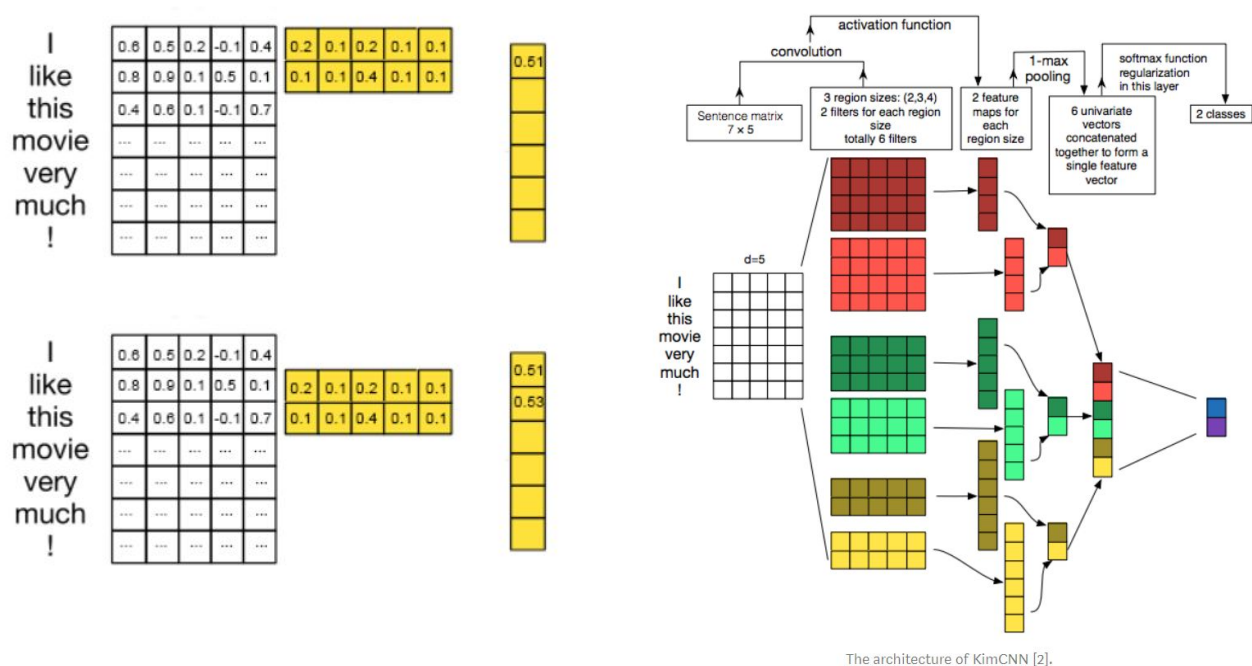
Thus this is a highly unbalanced dataset.

[0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	0,	0,	0,	0,	0,	0,	0,	0,	
	733,	78,	1,	140,	131,	182,	30,	712,	4438,
	10284,	1252,	86,	368,	51,	2230,	14039,	49,	6744,
	15,	60,	2624,	151,	7,	2832,	33,	115,	1246,
	16129,	2517,	5,	50,	59,	256,	1,	370,	31,
	1,	46,	29,	144,	72,	3931,	89,	4208,	6368,
	2687,	1183]							

Here the number of 0's in the beginning are the padded numbers in addition to tokens from the first 100,000 words in FastText embedding to ensure the vector size is 200 units.

Once the sentences have been tokenized we create an embedding matrix of dimension = minimum of (number of words in corpus or 100,000) * number of embeddings (300). This is what we use for 2D convolutions.

Note the 2-D convolutions in NLP tasks like these are actually fixed in the 2nd dimension. I.e. we never change the size of the number of columns. But we may use a filter of size 2,3,4 or more based on whether we want to use bigrams, trigrams etc. as the sequence related data in the sentence.



In the left figure above we can see how convolution and pooling is done using a 2D convolution where the 2nd dimension is equal to the length of the number of embeddings. The figure on the right is a KimCNN architecture which is slightly more complicated than basic 2D CNN as it also allows for 2 filters for each convolution layer. In our CNN based neural network architecture to classify sentences we used following parameters.

1. FastText based embedding are inputs which is a $n \times 300$ dimension matrix.
2. 4 convolution layers to model bigrams, trigrams, 4-grams and 5-grams with dimensions of (dim of filter[2,3,4,5], length of embeddings[300]).
3. 4 max pooling layers with dimensions of (max_len +1 - dim of filter in convolution layer, 1) to reduce the vectors dimension and obtain a 1-d vector.
4. Relu activation functions in conv layers.
5. Concatenate to get 1D outputs
6. Add a dropout layer to avoid overfitting.
7. Sigmoid activation function in the last dense layer with 6 possible outputs. To model multilabel classification.
8. Adam optimizer, with binary cross entropy loss function and ROC AUC as choice of metric.

Results :

1. Time taken for training and evaluation ~ 15 minutes.
2. RoC-AUC based on 80:20 split of training and evaluation data ~ 0.987

Code snippets :

Below is the code snippet for CNN model.

```
filter_sizes = [2,3,4,5]
num_filters = 32

def get_model():
    inp = Input(shape=(maxlen, ))
    x = Embedding(max_features, embed_size, weights=[embedding_matrix])(inp)
    x = SpatialDropout1D(0.4)(x)
    x = Reshape((maxlen, embed_size, 1))(x)

    conv_0 = Conv2D(num_filters, kernel_size=(filter_sizes[0], embed_size), kernel_initializer='normal',
                    activation='relu')(x)
    conv_1 = Conv2D(num_filters, kernel_size=(filter_sizes[1], embed_size), kernel_initializer='normal',
                    activation='relu')(x)
    conv_2 = Conv2D(num_filters, kernel_size=(filter_sizes[2], embed_size), kernel_initializer='normal',
                    activation='relu')(x)
    conv_3 = Conv2D(num_filters, kernel_size=(filter_sizes[3], embed_size), kernel_initializer='normal',
                    activation='relu')(x)

    maxpool_0 = MaxPool2D(pool_size=(maxlen - filter_sizes[0] + 1, 1))(conv_0)
    maxpool_1 = MaxPool2D(pool_size=(maxlen - filter_sizes[1] + 1, 1))(conv_1)
    maxpool_2 = MaxPool2D(pool_size=(maxlen - filter_sizes[2] + 1, 1))(conv_2)
    maxpool_3 = MaxPool2D(pool_size=(maxlen - filter_sizes[3] + 1, 1))(conv_3)

    z = Concatenate(axis=1)([maxpool_0, maxpool_1, maxpool_2, maxpool_3])
    z = Flatten()(z)
    z = Dropout(0.1)(z)

    outp = Dense(6, activation="sigmoid")(z)

    model = Model(inputs=inp, outputs=outp)
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    return model
```


Part 2 : Multilabel classification with BERT and BERT + CNN type Architectures.

Having learnt how to use CNN's with word embeddings we decided to explore BERT which has been regarded as a major game changer for NLP tasks. Thus we first wanted to understand BERT and combine it with CNN first and then leverage BERT by itself for classification and some other tasks for project 2. Below are few high level details and graphs which explain how and why bert is very impactful.

Introduction to BERT (Bidirectional Encoder Representations from Transformers).

It is an attention based transfer learning model released in late 2018 by Google. It involves pretraining language representations that can be used to create models that NLP practitioners can download & use for free. These models can be used to extract high quality language features from text data, or fine-tuned on a specific task (classification, entity recognition, question answering, etc.) with custom data to produce state of art predictions.

- The best part is that only slight tweaking of inputs and keeping the same architecture can result in multiple applications. The picture from the original paper highlights this important feature about BERT.

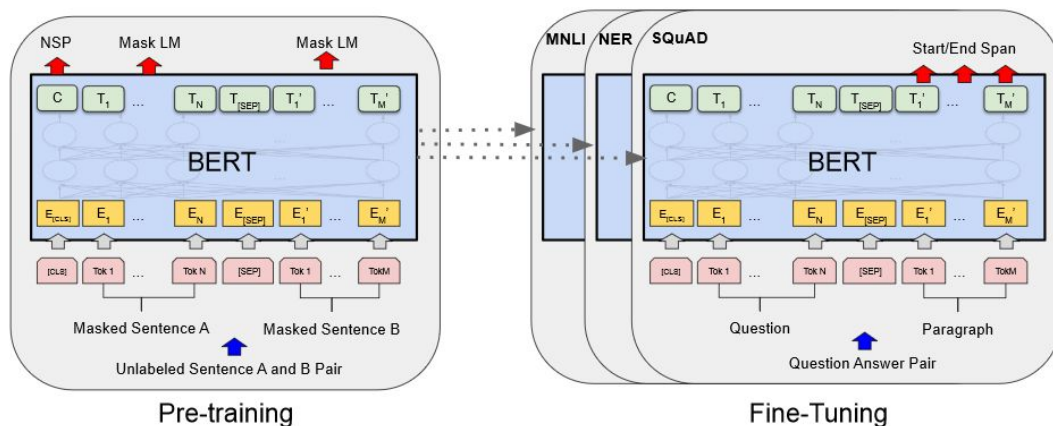


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

- Another advantage of BERT over other transformer models like ELMo and GPT is that it uses bidirectional transformer models very elegantly in all layers to achieve a great left and right context conditioning and allows fine tuning. Figure below from original paper shows BERT architecture & its comparison with other popular models.

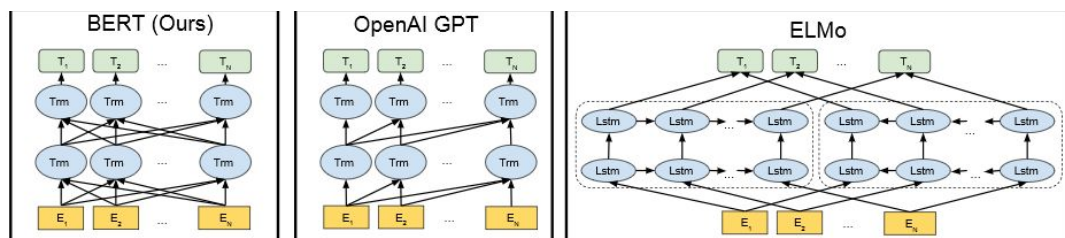
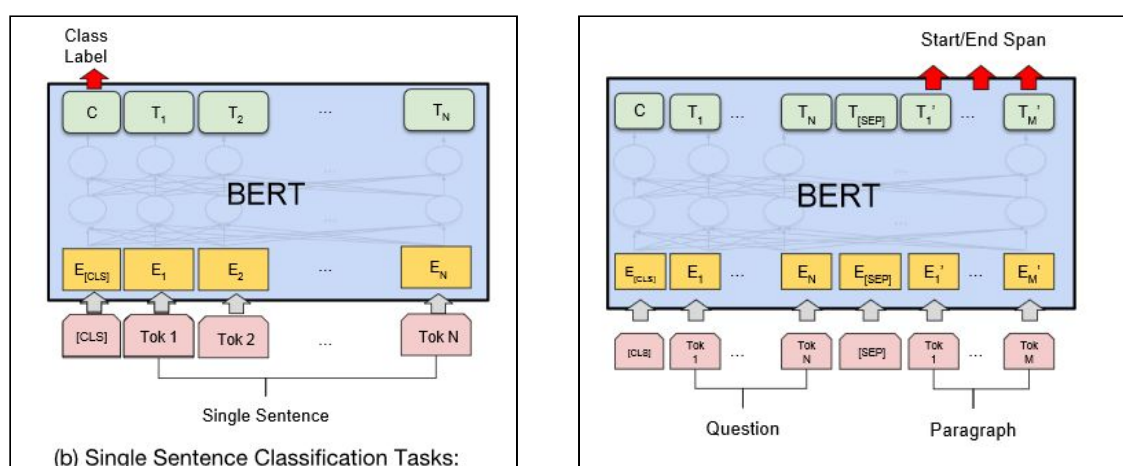


Figure 3: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach.

For this project we were mainly concerned about sentence classification and for the second part the Question Answer systems and hence the following two architectures were explored. The main differences in these two architectures are

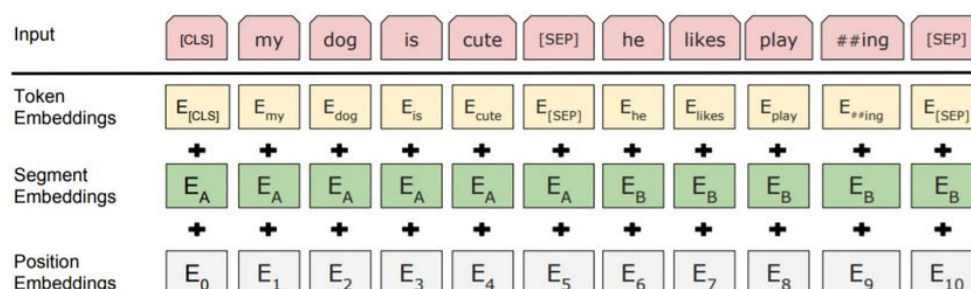
1. Output layers (for classification only one token in the last layer is used, whereas in QA tasks the last few tokens that determine start/end of sentence and questions are most useful.)
2. Use of tokens in input sequences also plays an important role and differences can occur in presence of separator tokens for questions, answers and paragraphs.



A main component of BERT is the usage of embedding layers.

Bert relies on embeddings and that too in a specific style that allows masking, classification and separation of key items. This specific styling allows bert to be tunable and modular for various tasks. The figure below shows how to obtain an input from raw text by combining

token embeddings, segment embeddings and position embedding which help in ensuring that proper context specific details are captured appropriately.



BERT input representation. The input embeddings is the sum of the token embeddings, the segmentation embeddings and the position embeddings.

BERT offers many ways to create word embeddings through its tunable encoder mechanism which offers a huge advantage over models like Word2Vec. This is because while each word has a fixed representation under Word2Vec regardless of the context within which the word

appears, BERT produces word representations that are dynamically informed by the words around them. For example, given two sentences:

“The man was accused of robbing a bank.” “The man went fishing by the bank of the river.”

Word2Vec would produce the same word embedding for the word “bank” in both sentences, while under BERT the word embedding for “bank” would be different for each sentence.

Aside from capturing obvious differences like polysemy, the context-informed word embeddings capture other forms of information that result in more accurate feature representations, which in turn results in better model performance.

These features of embedding layers can be exploited to train CNN type models just like we did with word embedding layers from FastText in part 1. This is being tried to be implemented in Part 2a of this project as shown in upcoming pages.

In part 2b we are exploring the BERT architecture mentioned in image (b) above and we used the library called simple transformers to achieve this goal. For this case we used the

BERT-Base case Uncased pretrained model that involves a combination of 12-transformer blocks, 768-hidden size, and 12 self-attention heads, totalling in around 110M parameters.

```
The BERT model has 201 different named parameters.

==== Embedding Layer ====

bert.embeddings.word_embeddings.weight          (30522, 768)
bert.embeddings.position_embeddings.weight       (512, 768)
bert.embeddings.token_type_embeddings.weight     (2, 768)
bert.embeddings.LayerNorm.weight               (768,)
bert.embeddings.LayerNorm.bias                 (768,)

==== First Transformer ====

bert.encoder.layer.0.attention.self.query.weight (768, 768)
bert.encoder.layer.0.attention.self.query.bias  (768,)
bert.encoder.layer.0.attention.self.key.weight  (768, 768)
bert.encoder.layer.0.attention.self.key.bias    (768,)
bert.encoder.layer.0.attention.self.value.weight (768, 768)
bert.encoder.layer.0.attention.self.value.bias  (768,)
bert.encoder.layer.0.attention.output.dense.weight (768, 768)
bert.encoder.layer.0.attention.output.dense.bias (768,)
bert.encoder.layer.0.attention.output.LayerNorm.weight (768,)
bert.encoder.layer.0.attention.output.LayerNorm.bias (768,)
bert.encoder.layer.0.intermediate.dense.weight  (3072, 768)
bert.encoder.layer.0.intermediate.dense.bias    (3072,)
bert.encoder.layer.0.output.dense.weight        (768, 3072)
bert.encoder.layer.0.output.dense.bias          (768,)
bert.encoder.layer.0.output.LayerNorm.weight    (768,)
bert.encoder.layer.0.output.LayerNorm.bias      (768,)

==== Output Layer ====

bert.pooler.dense.weight          (768, 768)
bert.pooler.dense.bias            (768,)
classifier.weight                 (6, 768)
classifier.bias                   (6,)
```


To further understand the concepts behind the attention heads and transformer blocks please refer to the original paper by vasvani et. al. titled "Attention is all you need". Some key concepts can be understood from following visuals taken from the original papers. The figure on the left shows how attention models keep a track of long distance dependencies in a sentence and how use of embedding tokens like <EOS>, "<EOS>, "." is used by layers. The figure on right shows basic transformer architecture. The figure at bottom shows how different attention heads can learn different tasks related to structure of sentences.

Attention Visualizations

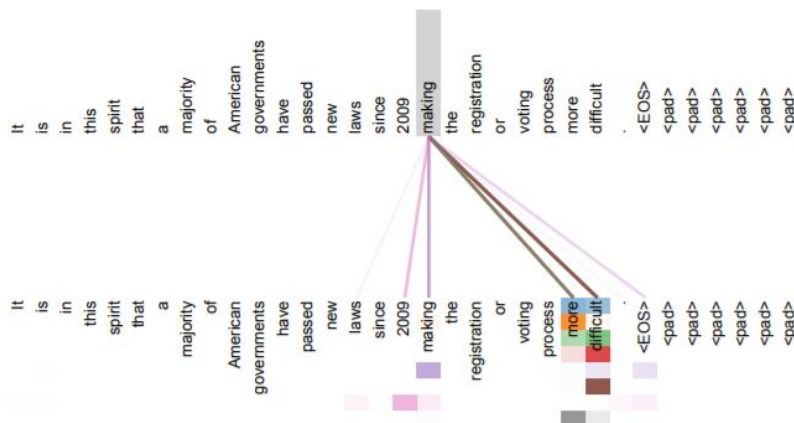


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

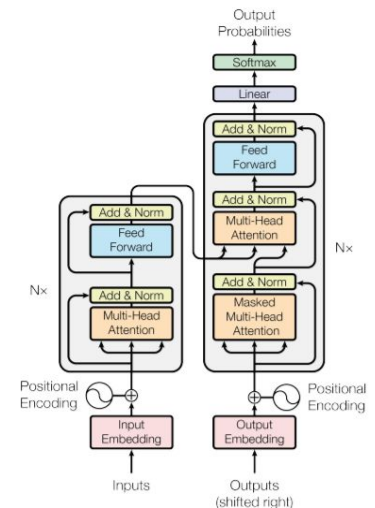


Figure 1: The Transformer - model architecture.

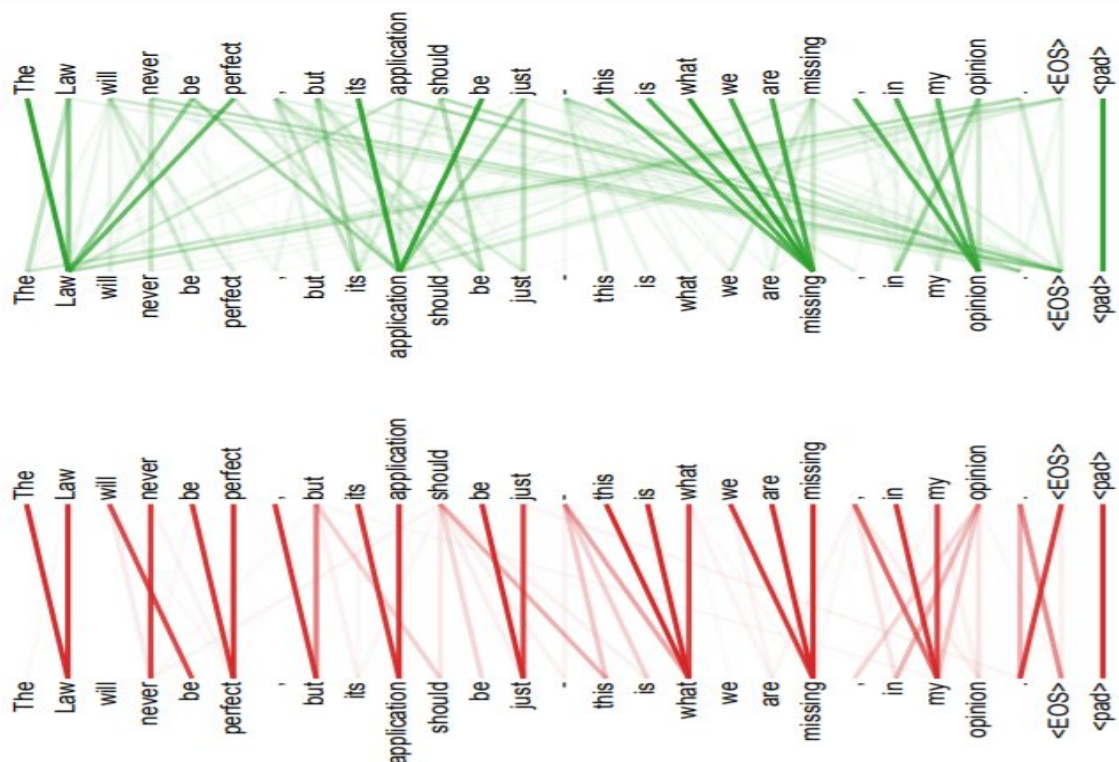


Figure 5: Many of the attention heads exhibit behaviour that seems related to the structure of the sentence. We give two such examples above, from two different heads from the encoder self-attention at layer 5 of 6. The heads clearly learned to perform different tasks.

Implementation NOTES :

As mentioned earlier in order to get familiar with BERT architecture we first tried to replicate a study that used BERT based fine tuned embeddings as inputs for a CNN based classifier. This helped us get more familiar with what happens to the text inputs we provide to Bert Model. This also helped us in exploring the lower level implementation details that one should be familiar with while implementing bert like

1. Impact of batching to avoid memory overflow errors.
2. Impact of sequence length on model training in terms of time to train and overall performance
3. How embeddings are generated using padding, truncation and use of specialized tokens.
- 4.

Based on our learnings from this replication study and after more research on the topic we decided to use a much simplified library that allows us to leverage BERT in a simpler manner and more efficiently without having to clean datasets once we obtain the initial dataset in specified format.

Step 1: Getting familiar with BERT & CNN by mimicking steps from a medium article.

Our initial goal was to mimic the steps in a well written article titled “Identifying Hate Speech with BERT and CNN” by Roman Orac. The article title is available at the link below.

<https://towardsdatascience.com/identifying-hate-speech-with-bert-and-cnn-b7aa2cddd60d>

This article was intended to be an introduction to bert embeddings and encoder layers for fine tuning bert embedding to adapt to a new dataset and then leverage these encodings to train a CNN model called “KimCNN” that seems to work really well on text data.

In this article the training dataset is kept intentionally very small so that the model can work on a CPU. We tried using a similar dataset which consisted of only 10,000 records from train.csv but unfortunately we kept getting out of memory errors. Hence we needed to dig a little deeper into issues and came across another good resource by Chris McCormick. This new article introduced us to concepts like “batching”, “dataloader” and “encoding”. There was one useful function called tokenizer.encode_plus function which significantly helped in creating suitable word embedding for BERT.

The blog post article by Chris McCormick was called “Bert fine tuning” that is available at the link below. <http://mccormickml.com/2019/07/22/BERT-fine-tuning/>

The main issue was that of batching and we needed to use data loader classes in pytorch. However we also came across another good tutorial type article about word embeddings in bert and that was useful for getting familiar with how embeddings were generated in BERT. The article is available at the following link.

<https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/>

Below are some key summaries and images based on these articles that we found very useful.

Getting BERT embeddings from words in a comment.

With BERT each word of a comment is transformed into a vector of size [1 x 768] (768 is the length of a BERT embedding). In fasttext example above the length of embedding was 300.

Now because a comment consists of multiple words, so we get a matrix [n x 768], where n is the number of words in a comment. There are less than n words as BERT inserts [CLS] token at the beginning of the first sentence and a [SEP] token at the end of each sentence. BERT-base-uncased language model, has 12 attention layers and uses a vocabulary of 30522 words. BERT uses a tokenizer to split the input text into a list of tokens that are available in the vocabulary. It learns words that are not in the vocabulary by splitting them into subwords.

For example a sentence "Here is the sentence I want embeddings for."

Shows up as

```
['[CLS]', 'here', 'is', 'the', 'sentence', 'i', 'want', 'em', '##bed', '##ding', '##s', 'for', '.', '[SEP]']
```

Notice presence of "[CLS]" token, "[SEP]" token and "##*". The tokens like [CLS] and [SEP] are used for classification and sentence ending detection. The [CLS] token is finally replaced with class scores in the final layer of bert for classification tasks. More information on this is available at this link. (<https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/>)

The ## based words are created by splitting the word embeddings into 'em', '##bed', '##ding', '##s' this is done because in the 30522 words in bert model we don't have embeddings as words but we do have "em", "bed", "ding" and "s". Now ## is used for characters or words that come after the first word in the split up main word to highlight that these are broken segments from the word.

To obtain the embeddings for a comment we transform each comment into a 2D matrix. Matrices have a predefined size, but some comments have more words than others. To transform a comment to a matrix, we need to:

- limit the length of a comment to 100-500 words (512 is max size of tokens allowed in bert-base-uncased),
- pad a comment with less than 100-500 words (add 0 vectors to the end).

BERT doesn't simply map each word to an embedding like it is the case with some context-free pre-trained language models (Word2Vec, FastText or GloVe). To calculate the

context, we need to feed the comments to the BERT model. So we need to tokenize, pad and convert comments to PyTorch Tensors which are fed into BERT to transform the text to embeddings.

We can use the `tokenizer.encode` function mentioned above from `bert tokenizer` which combines the following multiple steps for us:

1. Split the sentence into tokens.
2. Add the special [CLS] and [SEP] tokens.
3. Map the tokens to their IDs.
4. Pad or truncate all sentences to the same length.
5. Create the attention masks which explicitly differentiate real tokens from [PAD] tokens.

The image below shows how the first comment in `train.csv` shows up as in terms of bert embeddings. This is a tensor of size `n * 768` (`n = max_seq_length` allowed)

```
tensor([[ 0.1020, -0.1540, -0.1991, ..., -0.0927,  0.9687,  0.1253],
        [ 0.5115,  0.6774,  1.4377, ...,  1.0570,  0.3752, -0.3614],
        [ 0.0124,  0.1622,  1.1159, ...,  0.8882,  0.6164, -0.2798],
        ...,
        [ 0.1322,  0.0337,  1.0933, ..., -0.6233,  0.1783, -1.1930],
        [ 0.0658,  0.0356,  1.0270, ..., -0.6100,  0.0813, -1.1758],
        [ 0.2795,  0.3124,  0.8268, ..., -0.6755, -0.0943, -1.2319]])
```

This tensor of comments is then fed into a CNN that can help in multilabel classification of comments.

NOTE : since we were out of luck on memory issues with pytorch based implementation of bert even after batching, we had to limit ourselves to using only 10,000 rows as training dataset and 5000 rows each as validation and test datasets.

Below are some key code snippets for the word embedding process.

Code for Creation of model and tokenizer.

```
model_class = transformers.BertModel
tokenizer_class = transformers.BertTokenizer
pretrained_weights='bert-base-uncased'

tokenizer = tokenizer_class.from_pretrained(pretrained_weights)
bert_model = model_class.from_pretrained(pretrained_weights)
```

Code for Determining max_length of comments.

```
max_len = 0
# For every sentence...
for sent in df_train['comment_text']:
    # Tokenize the text and add `[CLS]` and `[SEP]` tokens.
    input_ids = tokenizer.encode(sent, add_special_tokens=True)
    # Update the maximum sentence length.
    max_len = max(max_len, len(input_ids))
print('Max sentence length: ', max_len)
```

Max sentence length: 1989

```
max_seq = 100
```

Note: even though the max_sentence length was 1989 we were limited by memory issues and had to use max_length as 100 for our case with tensorflow. We updated this to 512 in our bert only implementation as that was done using apex and tensorflow back end and didn't have memory issues.

Usage of encode_plus function to create bert embeddings with special tokens, attention mask and padding:

```
##### extra code #####
# Tokenize all of the sentences and map the tokens to their word IDs.
input_ids = []
attention_masks = []
# For every sentence...
for sent in df_train.comment_text:
    # '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,                      # Sentence to encode.
        add_special_tokens=True,    # Add '[CLS]' and '[SEP]'
        max_length=max_seq,        # 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'])
# Convert the lists into tensors.
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(df_train['labels'])
# Print sentence 0, now as a list of IDs.
print('Original: ', df_train.comment_text[0])
print('Token IDs:', input_ids[0])

Original: Geez, are you forgetfull! We've already discussed why Marx was not an anarchist, i.e. he wanted to use a State to mold his 'socialist man.' Ergo, he is a statist - the opposite of an anarchist.
Token IDs: tensor([ 101, 20277, 2480, 1010, 2024, 2017, 5293, 3993, 999, 2057,
1005, 2310, 2525, 6936, 2339, 13518, 2001, 2025, 2019, 18448,
1010, 1045, 1012, 1041, 1012, 2002, 2359, 2000, 2224, 1037,
2110, 2000, 18282, 2010, 1005, 6102, 2158, 1012, 1005, 9413,
3995, 1010, 2002, 2003, 1037, 28093, 2923, 1011, 1996, 4500,
1997, 2019, 18448, 1012, 1045, 2113, 1037, 3124, 2040, 2758,
2000, 1010, 2043, 2002, 4152, 2214, 1990, 2010, 4091, 2991,
2041, 1010, 2002, 1005, 2222, 8046, 5983, 6240, 1012, 2055,
2017, 2655, 2032, 1037, 23566, 1029, 102, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0])
```


Usage of data loader to create batches for training, validating and testing the model.

```
from torch.utils.data import DataLoader, RandomSampler, SequentialSampler
from torch.utils.data import TensorDataset

# 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 = 8
train_dataset = TensorDataset(input_ids, attention_masks, labels)

# 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.
)

val_dataset = TensorDataset(val_input_ids, val_attention_masks, val_labels)

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

test_dataset = TensorDataset(test_input_ids, test_attention_masks, test_labels)

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

model = bert_model
model.cuda()
```

Code to showcase the model parameters for the BERT model being used.

```
# Get all of the model's parameters as a list of tuples.
params = list(model.named_parameters())
print('The BERT model has {:} different named parameters.\n'.format(len(params)))
print('==== Embedding Layer ==== \n')
for p in params[0:5]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
print('\n==== First Transformer ==== \n')
for p in params[5:21]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
print('\n==== Output Layer ==== \n')
for p in params[-4:]:
    print("{:<55} {:>12}".format(p[0], str(tuple(p[1].size()))))
```

➤ The BERT model has 199 different named parameters.

==== Embedding Layer ====

embeddings.word_embeddings.weight	(30522, 768)
embeddings.position_embeddings.weight	(512, 768)
embeddings.token_type_embeddings.weight	(2, 768)
embeddings.LayerNorm.weight	(768,)
embeddings.LayerNorm.bias	(768,)

==== First Transformer ====

encoder.layer.0.attention.self.query.weight	(768, 768)
encoder.layer.0.attention.self.query.bias	(768,)
encoder.layer.0.attention.self.key.weight	(768, 768)
encoder.layer.0.attention.self.key.bias	(768,)
encoder.layer.0.attention.self.value.weight	(768, 768)
encoder.layer.0.attention.self.value.bias	(768,)
encoder.layer.0.attention.output.dense.weight	(768, 768)
encoder.layer.0.attention.output.dense.bias	(768,)
encoder.layer.0.attention.output.LayerNorm.weight	(768,)
encoder.layer.0.attention.output.LayerNorm.bias	(768,)
encoder.layer.0.intermediate.dense.weight	(3072, 768)
encoder.layer.0.intermediate.dense.bias	(3072,)
encoder.layer.0.output.dense.weight	(768, 3072)
encoder.layer.0.output.dense.bias	(768,)
encoder.layer.0.output.LayerNorm.weight	(768,)
encoder.layer.0.output.LayerNorm.bias	(768,)

==== Output Layer ====

encoder.layer.11.output.LayerNorm.weight	(768,)
encoder.layer.11.output.LayerNorm.bias	(768,)
pooler.dense.weight	(768, 768)
pooler.dense.bias	(768,)

Code for generating word embeddings in the training batch.

All the batches are appended to obtain a single embedding tensor. The same step is done for test and validation batches to get correct embeddings.

```
import random
import numpy as np
total_t0 = time.time()
embed_xtrain = []

# For each batch of training data...
for step, batch in enumerate(train_dataloader):
    with torch.no_grad():
        t0 = time.time()
        # Progress update every 40 batches.
        if step % 40 == 0 and not step == 0:
            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)
            # Report progress.
            print(' Batch {:>5}, of {:>5},. Elapsed: {}'.format(step, len(train_dataloader), elapsed))

        # Unpack this training batch from our dataloader.
        #
        # As we unpack the batch, we'll also copy each tensor to the GPU using the
        # `to` method.
        #
        # `batch` contains three pytorch tensors:
        # [0]: input ids
        # [1]: attention masks
        # [2]: labels
        b_input_ids = batch[0].to(device)
        x_train = model(b_input_ids)[0]
        embed_xtrain.append(x_train)

print("")
print("Training complete!")

print("Total training took {} (h:mm:ss)".format(format_time(time.time()-total_t0)))
```

Batch 40 of 1,250. Elapsed: 0:00:00.
Batch 80 of 1,250. Elapsed: 0:00:00.
Batch 120 of 1,250. Elapsed: 0:00:00.
Batch 160 of 1,250. Elapsed: 0:00:00.
Batch 200 of 1,250. Elapsed: 0:00:00.
Batch 240 of 1,250. Elapsed: 0:00:00.
Batch 280 of 1,250. Elapsed: 0:00:00.
Batch 320 of 1,250. Elapsed: 0:00:00.
Batch 360 of 1,250. Elapsed: 0:00:00.
Batch 400 of 1,250. Elapsed: 0:00:00.
Batch 440 of 1,250. Elapsed: 0:00:00.
Batch 480 of 1,250. Elapsed: 0:00:00.
Batch 520 of 1,250. Elapsed: 0:00:00.
Batch 560 of 1,250. Elapsed: 0:00:00.
Batch 600 of 1,250. Elapsed: 0:00:00.

Code for Combining all the batches and displaying first comments word embeddings.

```
b = torch.stack(embed_xtrain).view(-1,100,768)
x_train2 = b
x_train2[0].shape
```

torch.Size([100, 768])

[32] x_train2[0]

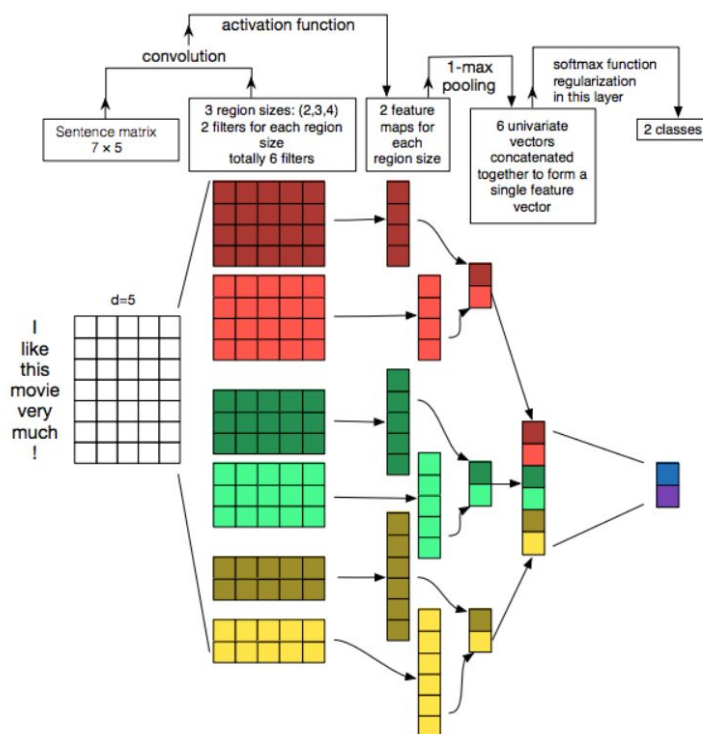
tensor([[0.3509, -0.2748, 0.1016, ..., 0.2375, 0.1749, 0.6065],
 [0.9436, -0.0820, 0.4454, ..., 0.4683, 0.7480, -0.2255],
 [0.4853, -0.0717, 0.2793, ..., -0.3621, 1.5902, -0.2867],
 ...,
 [0.3040, -0.5154, -0.6047, ..., -0.4600, -0.2937, 0.1232],
 [-0.8843, -0.5824, -0.5844, ..., 0.8551, 0.1752, 0.1995],
 [0.5226, 0.1167, -0.0986, ..., 0.9840, 0.0062, -0.0716]],
 device='cuda:0')

Note: The size of x_train[0] is 100,768 which shows only 100 tokens are used from the whole comment.

Once the tensor of word embeddings are obtained the next step is to create a CNN model.

This will be used to train and test the model using bert generated word embeddings.

We use the architecture suggested by Yoon Kim from New York University in a paper published in 2014. This offers a slight improvement over the simple 2D text cnn that we implemented in part 1 as it allows filter layers in addition to bigram, trigram and 4-gram convolutions. Thus each of those convolutions is repeated. The rest of the architecture is pretty similar to earlier CNN except that all steps in convolution and pooling are duplicated due to two filters.



The architecture of KimCNN [2].

Note : In our model since we have 6 labels we use a sigmoid function instead of softmax as shown in diagram

The main steps used for Kim-CNN here are as below.

1. Take a word embedding on the input $[n \times m]$, where n represents the maximum number of words in a sentence and m represents the length of the embedding.
2. Apply convolution operations on embeddings. It uses multiple convolutions of different sizes $[2 \times m]$, $[3 \times m]$ and $[4 \times m]$. The intuition behind this is to model combinations of 2 words, 3 words, etc. Note, that convolution width is m - the size of the embedding. This is because $[1 \times m]$ represents a whole word and it doesn't make sense to run a convolution with a smaller kernel size (eg. a convolution on half of the word).
3. Apply Rectified Linear Unit (ReLU) to add the ability to model nonlinear problems.

4. Apply 1-max pooling to down-sample the input representation and to help to prevent overfitting. Fewer parameters also reduce computational cost.
5. Concatenate vectors from previous operations to a single vector.
6. Add a dropout layer to deal with overfitting.
7. Apply sigmoid function to predict multiple labels. This scales logits between 0 and 1 for each class. This means that multiple classes can be predicted at the same time.
8. We use adam optimizer and binary cross entropy loss function along with roc_auc metrics for evaluating the model

The code for KimCNN implementation is as below.

1. Code for model definition.

```
class KimCNN(nn.Module):
    def __init__(self, embed_num, embed_dim, class_num, kernel_num, kernel_sizes, dropout, static):
        super(KimCNN, self).__init__()

        V = embed_num
        D = embed_dim
        C = class_num
        Co = kernel_num
        Ks = kernel_sizes

        self.static = static
        self.embed = nn.Embedding(V, D)
        self.convs1 = nn.ModuleList([nn.Conv2d(1, Co, (K, D)) for K in Ks])
        self.dropout = nn.Dropout(dropout)
        self.fc1 = nn.Linear(len(Ks) * Co, C)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        if self.static:
            x = Variable(x)

        x = x.unsqueeze(1) # (N, Ci, W, D)

        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs1] # [(N, Co, W), ...]*len(Ks)

        x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x] # [(N, Co), ...]*len(Ks)

        x = torch.cat(x, 1)
        x = self.dropout(x) # (N, len(Ks)*Co)
        logit = self.fc1(x) # (N, C)
        output = self.sigmoid(logit)
        return output

embed_num = x_train2.shape[1]
embed_dim = x_train2.shape[2]
class_num = y_train.shape[1]
kernel_num = 4
kernel_sizes = [2, 3, 4, 5]
dropout = 0.9
static = True
```

Key parameters are

- **embed_num** = 100 (max length of words), **embed_dim** = 768 (embedding dimension from bert), **class_num** = 6 (number of labels we need prediction for)
- **kernel_num** is the number of filters for each convolution (eg. 4 filters for [2 x m] convolution)
- **kernel_sizes of convolutions**. Eg. look at combinations 2 words, 3 words, etc.
- **dropout** is the percentage of randomly set hidden units to 0 at each update in training.
- **static parameter** : True means that we don't calculate gradients of embeddings.

Code for model generation and training parameters.

```
model2 = KimCNN(
    embed_num=embed_num,
    embed_dim=embed_dim,
    class_num=class_num,
    kernel_num=kernel_num,
    kernel_sizes=kernel_sizes,
    dropout=dropout,
    static=static,
)
model2.to(device)

KimCNN(
  (embed): Embedding(100, 768)
  (conv1): ModuleList(
    (0): Conv2d(1, 4, kernel_size=(2, 768), stride=(1, 1))
    (1): Conv2d(1, 4, kernel_size=(3, 768), stride=(1, 1))
    (2): Conv2d(1, 4, kernel_size=(4, 768), stride=(1, 1))
    (3): Conv2d(1, 4, kernel_size=(5, 768), stride=(1, 1))
  )
  (dropout): Dropout(p=0.9, inplace=False)
  (fc1): Linear(in_features=16, out_features=6, bias=True)
  (sigmoid): Sigmoid()
)

n_epochs = 50
batch_size = 1000
lr = 0.001
optimizer = torch.optim.Adam(model2.parameters(), lr=lr)
loss_fn = nn.BCELoss()
```

Code for batch generation :

```
def generate_batch_data(x, y, batch_size):
    i, batch = 0, 0
    for batch, i in enumerate(range(0, len(x) - batch_size, batch_size), 1):
        x_batch = x[i : i + batch_size]
        y_batch = y[i : i + batch_size]
        yield x_batch, y_batch, batch
    if i + batch_size < len(x):
        yield x[i + batch_size :], y[i + batch_size :], batch + 1
    if batch == 0:
        yield x, y, 1
```

Code for training and evaluation along with graphs of loss function

```
train_losses, val_losses = [], []

for epoch in range(n_epochs):
    start_time = time.time()
    train_loss = 0

    model2.train(True)
    for x_batch, y_batch, batch in generate_batch_data(x_train2, y_train, batch_size):
        x_batch= x_batch.to(device)
        y_pred = model2(x_batch).to(device)
        optimizer.zero_grad()
        y_batch = y_batch.to(device)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= batch
    train_losses.append(train_loss)
    elapsed = time.time() - start_time

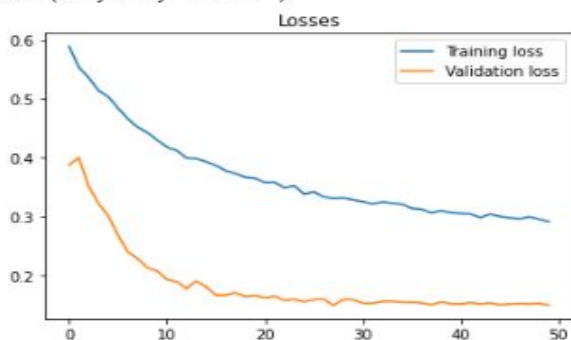
    model2.eval() # disable dropout for deterministic output
    with torch.no_grad(): # deactivate autograd engine to reduce memory usage and speed up computations
        val_loss, batch = 0, 1
        for x_batch, y_batch, batch in generate_batch_data(x_val2, y_val, batch_size):
            x_batch= x_batch.to(device)
            y_pred = model2(x_batch).to(device)
            y_batch = y_batch.to(device)
            loss = loss_fn(y_pred, y_batch)
            val_loss += loss.item()
        val_loss /= batch
        val_losses.append(val_loss)

    print(
        "Epoch %d Train loss: %.2f. Validation loss: %.2f. Elapsed time: %.2fs."
        % (epoch + 1, train_losses[-1], val_losses[-1], elapsed)
    )
```

```
Epoch 1 Train loss: 0.59. Validation loss: 0.39. Elapsed time: 0.63s.
Epoch 2 Train loss: 0.55. Validation loss: 0.40. Elapsed time: 0.63s.
Epoch 3 Train loss: 0.54. Validation loss: 0.35. Elapsed time: 0.64s.
Epoch 4 Train loss: 0.51. Validation loss: 0.32. Elapsed time: 0.63s.
Epoch 5 Train loss: 0.50. Validation loss: 0.30. Elapsed time: 0.63s.
Epoch 6 Train loss: 0.48. Validation loss: 0.27. Elapsed time: 0.63s.
Epoch 7 Train loss: 0.47. Validation loss: 0.24. Elapsed time: 0.63s.
Epoch 8 Train loss: 0.45. Validation loss: 0.23. Elapsed time: 0.63s.
Epoch 9 Train loss: 0.44. Validation loss: 0.21. Elapsed time: 0.63s.
Epoch 10 Train loss: 0.43. Validation loss: 0.21. Elapsed time: 0.63s.
Epoch 11 Train loss: 0.42. Validation loss: 0.19. Elapsed time: 0.63s.
Epoch 12 Train loss: 0.41. Validation loss: 0.19. Elapsed time: 0.63s.
```

```
plt.plot(train_losses, label="Training loss")
plt.plot(val_losses, label="Validation loss")
plt.legend()
plt.title("Losses")
```

Text(0.5, 1.0, 'Losses')



Note: The loss functions didn't completely stabilize for training data and this might mean that our training sample may be pretty small. However for validation set the loss is pretty low.

Code for prediction.

```
model2.eval() # disable dropout for deterministic output
with torch.no_grad(): # deactivate autograd engine to reduce memory usage and speed up computations
    y_preds = []
    batch = 0
    for x_batch, y_batch, batch in generate_batch_data(x_test2, y_test, batch_size):
        y_pred = model2(x_batch)
        y_preds.extend(y_pred.cpu().numpy().tolist())
    y_preds_np = np.array(y_preds)
```

Results and Code for generating results :

We see that this model doesn't get very good Auc-Roc values for any class. This is probably because the dataset is very unbalanced and we provided very few examples for training.

```
auc_scores = roc_auc_score(y_test_np, y_preds_np, average=None)
df_accuracy = pd.DataFrame({"label": target_columns, "auc": auc_scores})
df_accuracy.sort_values('auc')[::-1]
```

	label	auc
0	toxic	0.487205
2	obscene	0.483615
4	insult	0.478866
5	identity_hate	0.478558
3	threat	0.473858
1	severe_toxic	0.452424

```
positive_labels = df_train[target_columns].sum().sum()
positive_labels
```

2201

```
all_labels = df_train[target_columns].count().sum()
all_labels
```

60000

```
positive_labels/all_labels
```

0.03668333333333333

```
df_test_targets.sum()
```

toxic	474
severe_toxic	36
obscene	261
threat	8
insult	242
identity_hate	58
dtype: int64	

```
df_pred_targets.sum()
```

toxic	639.0
severe_toxic	0.0
obscene	3.0
threat	0.0
insult	2.0
identity_hate	0.0
dtype: float64	

The predictions are also never showing up threat, identity hate or severe toxicity thus highlighting the limitations of this model with limited training dataset.

Other results and observations :

- Time for training and evaluation with bert embedding + CNN is pretty small ~ 5 minutes for training on 10,000 comments.
- The major issue is due to memory errors as we can't train on a complete training set without going out of memory.
- This was a good experience to learn about BERT and KimCNN architecture on a toy dataset that works but isn't very accurate due to data limitations.

Therefore after having learned enough about CNN and BERT we now move on to exploring BERT on its own and a few other architectures in project 2 like RNN, LSTM, GRUS and another application of BERT.

Step 2: Implementation after getting familiar with BERT using simple transformers library

Having understood the finer details of Bert with the above project we wanted to explore a scalable and less complicated way to implement bert. Hence we decided to use a library that takes care of trivial tasks like creation of embeddings by default and we found out that libraries like simple transformers were made recently to do exactly that.

The implementation of BERT on the training dataset was made really easy by a library called simple transformers which allows tasks like QA systems modelling, classification, named entity recognition etc. in a very simplified manner. The aim was to get familiar with basic implementation of BERT for multiple tasks using the same base models.

The code for training and evaluation dataset creation is as below.

```
import pandas as pd
from simpletransformers.classification import MultiLabelClassificationModel
df = pd.read_csv('/content/drive/My Drive/train.csv')

df['labels'] = list(zip(df.toxic.tolist(), df.severe_toxic.tolist(), df.obscene.tolist(),
                      df.threat.tolist(), df.insult.tolist(), df.identity_hate.tolist()))
df['text'] = df['comment_text'].apply(lambda x: x.replace('\n', ' '))
df = df[['text', 'labels']]

from sklearn.model_selection import train_test_split
train_df, eval_df = train_test_split(df, test_size=0.2, random_state=42)
```

The model creation is as below where we use multilabelclassification module from simple transformers library and use pretrained bert model called “bert-base-uncased” as mentioned above with 6 labels. We use training batch size of 2 to avoid memory errors in gpu. The learning rate of 3e-5 seems to show decent results in reasonable time without causing gradient overflow errors which are caused by lower learning rates as we get “Nan” losses. We used 1 epoch to ensure we get the model trained in 2.5 hours. More epochs can help us get better model for evaluate. We use the max sequence length allowed by BERT i.e. 512 to ensure we take maximum features from the comment_text.

```
model = MultiLabelClassificationModel('bert', 'bert-base-uncased', num_labels=6,
    args={"reprocess_input_data": True,
          "overwrite_output_dir": True,
          "train_batch_size": 2,
          "gradient_accumulation_steps": 16,
          "learning_rate": 3e-5,
          "num_train_epochs": 1,
          "max_seq_length": 512})
```

Main parameters :

- **Model** : MultiLabelClassificationModel (this model can take in inputs as text and labels from a text where labels are tuples or arrays of numbers corresponding to values for each label)
- **Type of pre-trained model** :

WE use Bert with pre-embeddings and trained models based on bert-base-uncased. Other options include 'xlnet', 'xlm', 'roberta', 'distilbert' etc. we use Bert as we were slightly familiar with underlying architecture and also because this has fewer parameters compared to others.

- **Num_labels** : we have 6 labels that we wanted to work with for multilabel classification.

- **max_seq_length**: we use 512 to allow each sentence to be represented with a vector of 512 tokens. This is the maximum limit for bert-base-uncased models.

Results:

- Time : it took around 2.5 hours to train and evaluate the model with just one epoch using high RAM colab instances with GPU's.
- The overall AUC_ROC score was really good ~ 0.9827 on the evaluation dataset and this was pretty similar to values we got using CNN + word embeddings based architecture.

```
{'LRAP': 0.9957381586505826, 'roc_auc': 0.9827030551511439, 'eval_loss': 0.04881058259155802}
```

- The label specific ROC scores were also very good and pretty high on evaluation dataset.

	label	auc
3	threat	0.989509
1	severe_toxic	0.984375
2	obscene	0.983872
4	insult	0.983363
0	toxic	0.979992
5	identity_hate	0.975107

- For sanity check we compared the sum of all labels in the actual dataset and in the model prediction and found pretty close matches.

df_eval_targets.sum()	
toxic	3056
severe_toxic	321
obscene	1715
threat	74
insult	1614
identity_hate	294
dtype: int64	

df_pred_targets.sum()	
toxic	3104
severe_toxic	140
obscene	1611
threat	59
insult	1553
identity_hate	99
dtype: int64	

Observation and overall results from CNN and Bert Explorations:

- We saw that using a simple 2D- CNN architecture with fastText based word embeddings we can get pretty good ROC values.
- The BERT architecture is really good for multilabel classification and can be leveraged along with CNN.
- BERT alone can be used for text classification and is a really good model based on our dataset as it was able to predict all labels pretty effectively.
- CNN's can be used for text analysis pretty effectively if we use 2D architecture with word embeddings.

Limitations in current studies:

- We were limited by software limitations of pytorch.
- We were unfamiliar with pytorch and hence it was difficult to debug memory issues.
- We were limited by hardware as we couldn't fine tune various models as the models training and validation especially for bert took 2-3 hours for just one epoch in very basic settings.

Project 2: Using RNNs, LSTMs, GRUs and BERT for creation of applications like chatbots and QA systems.

Part 1 : Models comparisons and Chatbot implementation using Attention, GRU, LSTM, RNN recurrent sequence-to-sequence models.

Introduction to Dataset:

The dataset is obtained from cornell.edu website containing Cornell Movie-Dialogs Corpus

Link- https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

The Cornell Movie-Dialogs Corpus is a rich dataset of movie character dialogue:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- 9,035 characters from 617 movies
- 304,713 total utterances

movie_lines.txt

- contains the actual text of each utterance
- fields:
- lineID
- characterID (who uttered this phrase)
- movieID
- character name
- text of the utterance

movie_conversations.txt

- the structure of the conversations
- fields
- characterID of the first character involved in the conversation
- characterID of the second character involved in the conversation
- movieID of the movie in which the conversation occurred
- list of the utterances that make the conversation.

Step 1: - Preparing Data for Models

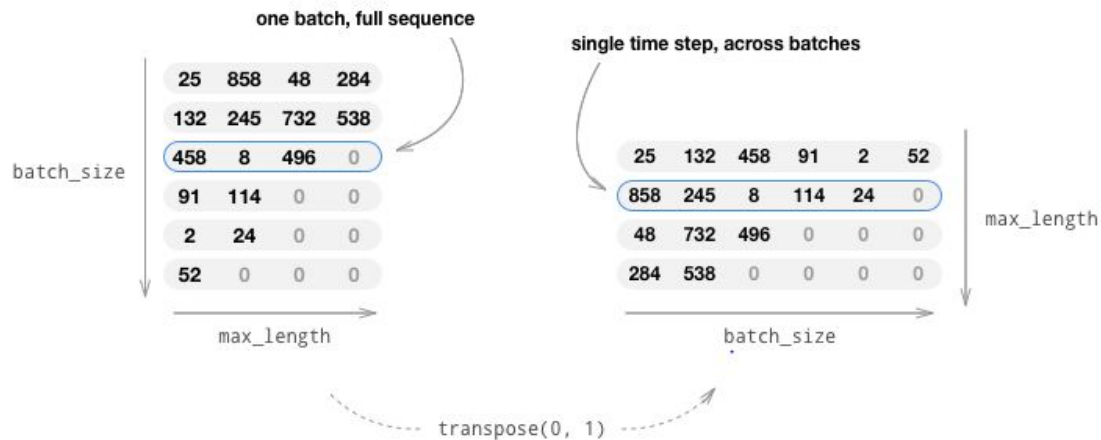
Data Cleaning and Pre-processing

The data files from movie_lines.txt are read to file splitting lines by '+++\$+++' and Groups fields of lines into conversations based on movie_conversations.txt and Extracts pairs of sentences from conversations

- Formatting into a data file in which each line contains a tab-separated query sentence and a response sentence pair matching the order of (lineID, characterID, movieID, character, text) according to movie_conversations.txt file.
- Trimming and creating a vocabulary and load query/response sentence pairs into memory with proper padding.

- Normalizing for special characters and trimming rare words for model efficiency purpose

we use a batch size of 1 and convert the words in our sentence pairs to their corresponding indexes from the vocabulary and feed this to the models. Data is padded to make it uniform across varying sentence lengths



We will generate pairs of conversational texts between character ids for all the corresponding conversational ids. Here question and response of characters are separated by a comma ',' separator.

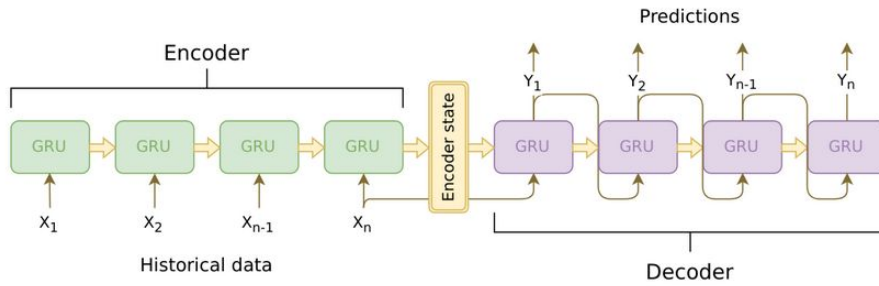
```
pairs:
['there .', 'where ?']
['you have my word . as a gentleman', 'you re sweet .']
['hi .', 'looks like things worked out tonight huh ?']
['you know chastity ?', 'i believe we share an art instructor']
['have fun tonight ?', 'tons']
['well no . . .', 'then that s all you had to say .']
['then that s all you had to say .', 'but']
['but', 'you always been this selfish ?']
['do you listen to this crap ?', 'what crap ?']
['what good stuff ?', 'the real you .']
```

Defining Models

Sequence-Sequence Models:

The goal of a seq2seq model is to take a variable-length sequence as an input and return a variable-length sequence as an output using a fixed-sized model.

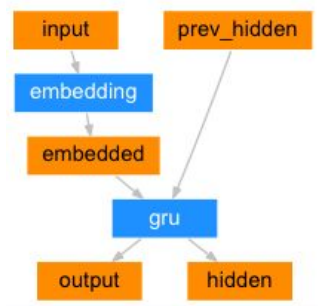
One RNN acts as an **encoder**, which encodes a variable length input sequence to a fixed-length context vector. The second RNN is a **decoder**, which takes an input word and the context vector, and returns a guess for the next word in the sequence and a hidden state to use in the next iteration.



Stages of the model preparation

Stage 1 – Encoder

The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word.



Encoders takes input of padded, embedded sentences with the following configuration and generates output along with a context vector at each stage of input.

```

input_variable: tensor([[ 25, 266, 45, 33, 23],
                        [197, 4, 469, 685, 6],
                        [117, 219, 115, 35, 2],
                        [ 24, 101, 473, 2, 0],
                        [ 25, 102, 4, 0, 0],
                        [200, 631, 2, 0, 0],
                        [965, 4, 0, 0, 0],
                        [ 4, 2, 0, 0, 0],
                        [ 2, 0, 0, 0, 0]])
lengths: tensor([9, 8, 6, 4, 3])
target_variable: tensor([[ 64, 12, 1498, 1493, 197],
                        [ 6, 201, 37, 4, 117],
                        [ 2, 428, 82, 2, 242],
                        [ 0, 302, 511, 0, 64],
                        [ 0, 7319, 98, 0, 1531],
                        [ 0, 115, 53, 0, 4],
                        [ 0, 95, 127, 0, 2],
                        [ 0, 4, 4, 0, 0],
                        [ 0, 2, 2, 0, 0]])
mask: tensor([[1, 1, 1, 1, 1],
              [1, 1, 1, 1, 1],
              [1, 1, 1, 1, 1],
              [0, 1, 1, 0, 1],
              [0, 1, 1, 0, 1],
              [0, 1, 1, 0, 1],
              [0, 1, 1, 0, 1],
              [0, 1, 1, 0, 1],
              [0, 1, 1, 0, 0],
              [0, 1, 1, 0, 0]], dtype=torch.uint8)
max_target_len: 9

```

```

[159] # Configure models
      model_name = 'cb_model'
      attn_model = 'dot'
      #attn_model = 'general'
      #attn_model = 'concat'
      hidden_size = 500
      encoder_n_layers = 2
      decoder_n_layers = 2
      dropout = 0.1
      batch_size = 64

```

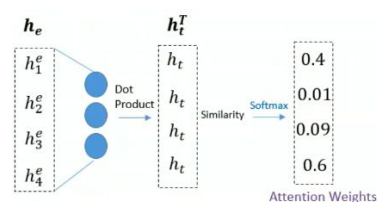
```

self.gru = nn.RNN(hidden_size, hidden_size, n_layers,
                  dropout=(0 if n_layers == 1 else dropout), bidirectional=True)

```

Stage 2- Attention Decoder Layer:

This encoder hidden state output(context) vectors are fed to an attention layer. It calculates and selects the highest weighted hidden state vector by using Attention operation of Dot product. So we mainly get flexibility of selecting most important hidden states for processing data.



For e.g. If the above Attention weights correspond to [what=0.4, do=0.01, you=0.09, work=0.6]. When generating next word attention gives importance to word ‘work’ and ‘what’.


```

class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.RNN(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size*2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

```

Input:

- `input_step`: one word of input sequence batch; shape= (1, batch_size)
- `last_hidden`: final hidden layer shape= (n_layers x num_directions, batch_size, hidden_size)
- `encoder_outputs`: encoder model's output; shape= (max_length, batch_size, hidden_size)

Output:

- `output`: softmax normalized tensor probabilities of each word being the correct next word in the decoded sequence; shape=(batch_size, voc.num_words)
- `hidden`: final hidden state of GRU; shape=(n_layers x num_directions, batch_size, hidden_size)

Stage 3- Training Model

The models are trained on 40 sequences, each containing 40000 iterations. The data is provided to the models as a single batch. The LSTM and GRU cells are implemented using the default settings of the CuDNNLSTM and CuDNNGRU layers in Keras with Tensorflow 1.12.0 as the back end. Each model contains a single cell with 16 neurons. The Adam optimizer is used with a constant learning rate of 0.001.

Model Configuration

- A (10,5)-dimensional word embedding layer was used to represent the input words.
- Softmax was used on the output layer.
- The model was fit for 4000 iterations of batches where some learning rate decay was performed.
- A batch-size of 64 sequences was used during training.
- Batches were comprised of sentences with roughly the same length to speed-up computation.
- `teacher_forcing_ratio` = 1.0
- `learning_rate` = 0.0001

- decoder_learning_ratio = 5.0
- n_iteration = 4000
- attn_model = 'dot'
- hidden_size = 500
- encoder_n_layers = 2
- decoder_n_layers = 2
- dropout = 0.1
- batch_size = 64
- small_batch_size = 5
- max_target_len: 10

The model was run on google colab with high disk allocation and GPU machine where each layer was run on a different GPU. Training took around 2 hours and 20 minutes.

Teacher forcing ratio is used for the current target word as the decoder's next input rather than using the decoder's current guess. This has an advantage as the model selects the decoder next input from input model because if the decoder's current guess goes wrong it disturbs the complete next decoder outputs as they are passed as inputs.

```
if use_teacher_forcing:
    for t in range(max_target_len):
        decoder_output, decoder_hidden = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )
        # Teacher forcing: next input is current target
        decoder_input = target_variable[t].view(1, -1)
        # Calculate and accumulate loss
        mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
        loss += mask_loss
        print_losses.append(mask_loss.item() * nTotal)
        n_totals += nTotal
```

We use one of the models (RNN/LSTM/GRU) decoder as the decoder mechanism which takes hidden state context vector from Attention and produces the next word as output.

```
# Define layers
self.embedding = embedding
self.embedding_dropout = nn.Dropout(dropout)
self.gru = nn.RNN(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers == 1 else dropout))
self.concat = nn.Linear(hidden_size*2, hidden_size)
self.out = nn.Linear(hidden_size, output_size)

self.attn = Attn(attn_model, hidden_size)
```

In batches of padded sequences, we cannot simply consider all elements of the tensor when calculating loss. We define `maskNLLLoss` to calculate our loss based on our decoder's output tensor, the target tensor, and a binary mask tensor describing the padding of the target tensor.

```
[182] def maskNLLLoss(inp, target, mask):
    nTotal = mask.sum()
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

This loss function calculates the average negative log likelihood of the elements that correspond to a 1 in the mask tensor.

We run the data using n-iterations of the training using `trainIters` function for training our data in batches and build encoder and decoder.

```
print('Building encoder and decoder ...')
# Initialize word embeddings
embedding = nn.Embedding(voc.num_words, hidden_size)
if loadFilename:
    embedding.load_state_dict(embedding_sd)
# Initialize encoder & decoder models
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words, decoder_n_layers, dropout)
if loadFilename:
    encoder.load_state_dict(encoder_sd)
    decoder.load_state_dict(decoder_sd)
# Use appropriate device
encoder = encoder.to(device)
decoder = decoder.to(device)
print('Models built and ready to go!')
```

```
Building encoder and decoder ...
Models built and ready to go!
```

Now after building the model, we run our training in the form of training Iterations.

```
[162] # Start training
res_lstm=trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer, decoder_optimizer,
                    embedding, encoder_n_layers, decoder_n_layers, save_dir, n_iteration, batch_size,
                    print_every, save_every, clip, corpus_name, loadFilename)
```

```
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
Iteration: 3996; Percent complete: 99.9%; Average loss: 2.2212
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
/pytorch/aten/src/ATen/native/LegacyDefinitions.cpp:70: UserWarning: masked_select received a mask with dtype torch.FloatTensor
Iteration: 3997; Percent complete: 99.9%; Average loss: 2.0496
```

Stage 4 Evaluating Text- Chatbot

`evaluateInput` acts as the user interface for the chatbot. When called, an input text field will spawn in where we can enter our query sentence. text is normalized in the same way as our training data. This data is ultimately fed to the `evaluate` function to obtain a decoded output sentence. We loop this process, so we can keep chatting with our bot until we enter either “q” or “quit”.

```
[179] # Set dropout layers to eval mode
      encoder.eval()
      decoder.eval()

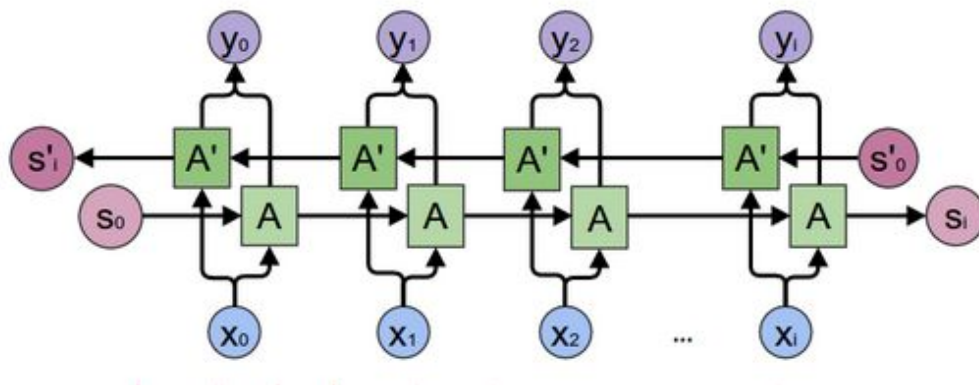
      # Initialize search module
      searcher = GreedySearchDecoder(encoder, decoder)

      # Begin chatting (uncomment and run the following line to begin)
      evaluateInput(encoder, decoder, searcher, voc)
```

```
> Hi
Bot: hi .
> Name?
Bot: olive .
> How's it going?
Bot: i m not .
> Are you serious?
Bot: yes .
> q
```

Model 1 – RNN Encoder + Attention RNN Decoder

Bidirectional RNN:



The encoder RNN iterates through the input sentence one token (e.g. word) at a time, at each time step outputting an “output” vector and a “hidden state” vector. The hidden state vector is then passed to the next time step, while the output vector is recorded. The encoder transforms the context it saw at each point in the sequence into a set of points in a high-dimensional space, which the decoder will use to generate a meaningful output for the given task.

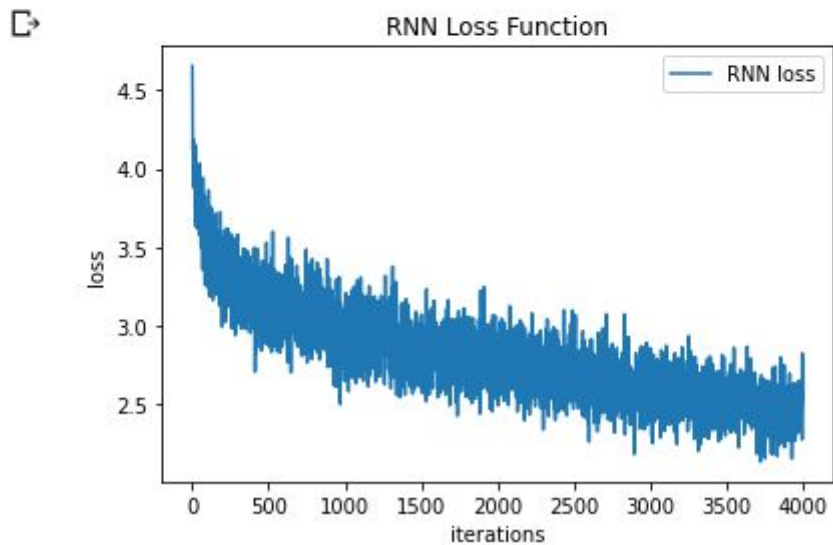
Training Model: -

The models are trained on sequences, each containing 4000 iterations. The data is provided to the models as a single batch. The RNN cells are implemented using the default settings of the `torch.nn.RNN(*args, **kwargs)` layers in pytorch as the back end. The Adam optimizer is used with a constant learning rate of 0.001.

Model Configuration

- A (10,5)-dimensional word embedding layer was used to represent the input words.
- Softmax was used on the output layer.
- The model was fit for 4000 iterations of batches where some learning rate decay was performed.
- A batch-size of 64 sequences was used during training.
- Batches were comprised of sentences with roughly the same length to speed-up computation.
- `teacher_forcing_ratio` = 1.0
- `learning_rate` = 0.0001
- `decoder_learning_ratio` = 5.0
- `n_iteration` = 4000
- `attn_model` = 'dot'
- `hidden_size` = 500
- `encoder_n_layers` = 2
- `decoder_n_layers` = 2
- `dropout` = 0.1
- `batch_size` = 64
- `small_batch_size` = 5
- `max_target_len`: 10


```
[224] plt.plot(res_rnn, label='RNN loss')
plt.title('RNN Loss Function')
plt.ylabel('loss')
plt.xlabel("iterations")
plt.legend(loc = 'best')
plt.show()
```

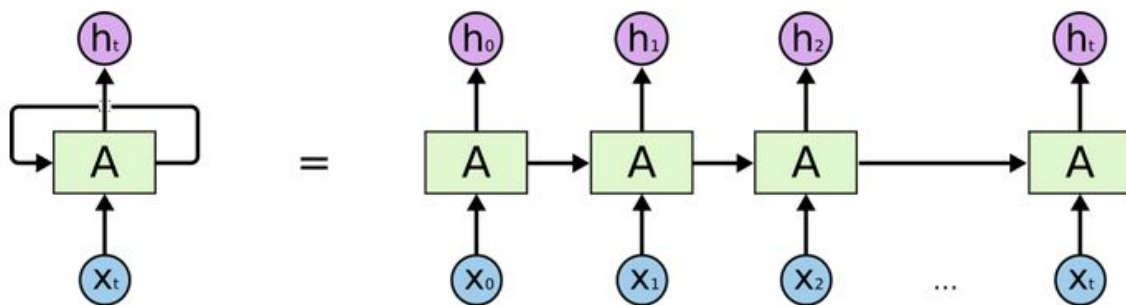


Average loss value of this model

```
sum(res_rnn)/4000
```

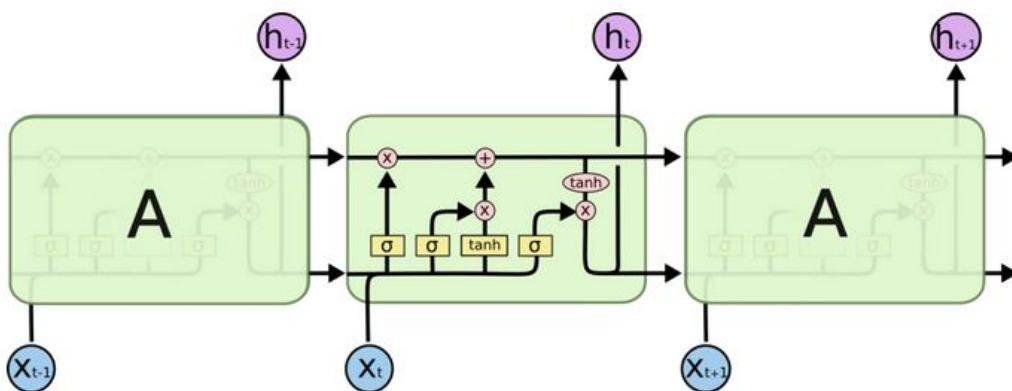
2.815537046418692

Model 2- LSTM + ATTENTION



Sequential processing in RNN, from: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

RNN and LSTM are mainly used for sequential processing over time. Long-term information has to sequentially travel through all cells before getting to the present processing cell. This means it can be easily corrupted by being multiplied by small numbers < 0 . This is called as vanishing gradients. To the rescue, LSTMs can bypass units and thus remember for longer time steps. Thus, have a way to remove some of the vanishing gradients problems. The LSTM cell maintains a cell state that is read from and written to. There are 4 gates that regulate the reading, writing, and outputting values to and from the cell state, dependent upon the input and cell state values. The first gate determines what the hidden state forgets. The next gate is responsible for determining what part of the cell state is written to. The third gate decides the contents that are written. Finally, the last gate reads from the cell state to produce an output.



Sequential processing in LSTM, from: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Training Model: -

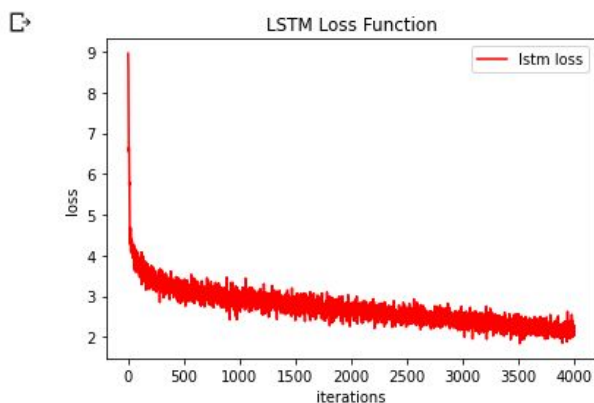
The models are trained on sequences, each containing 4000 iterations. The data is provided to the models as a single batch. The LSTM cells are implemented using the default settings of the `torch.nn.LSTM*args, **kwargs` layers in pytorch as the back end. The Adam optimizer is used with a constant learning rate of 0.001.

Model Configuration

- A (10,5)-dimensional word embedding layer was used to represent the input words.
- Softmax was used on the output layer.

- The model was fit for 4000 iterations of batches where some learning rate decay was performed.
- `input_size` – The number of expected features in the input= 10
- `hidden_size` – The number of features in the hidden state= 10
- A batch-size of 64 sequences was used during training.
- Batches were comprised of sentences with roughly the same length to speed-up computation.
- `teacher_forcing_ratio` = 1.0
- `learning_rate` = 0.0001
- `decoder_learning_ratio` = 5.0
- `n_iteration` = 4000
- `attn_model` = 'dot'
- `hidden_size` = 500
- `encoder_n_layers` = 2
- `decoder_n_layers` = 2
- `dropout` = 0.1
- `batch_size` = 64
- `small_batch_size` = 5
- `max_target_len`: 10

```
[225] plt.plot(res_lstm, label='lstm loss',color="r")
plt.title('LSTM Loss Function')
plt.ylabel('loss')
plt.xlabel("iterations")
plt.legend(loc = 'best')
plt.show()
```



Average loss value of this model

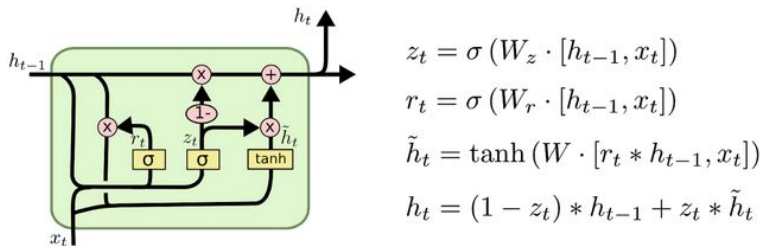
```
[ ] sum(res_lstm)/4000
```

```
↳ 2.725111623162019
```

Model 3: - GRU + Attention

GRU is related to LSTM as both are utilizing different way if gating information to prevent vanishing gradient problem. Here are some pin-points about GRU vs LSTM-

- The GRU controls the flow of information like the LSTM unit, but without having to use a **memory unit**. It just exposes the full hidden content without any control.
- GRU is relatively new, and from my perspective, the performance is on par with LSTM, but computationally **more efficient** (*less complex structure as pointed out*).



IN GRU there is no hidden state. The cell state adopts the functionality of the hidden state from the LSTM cell design. Next, the processes of determining what the cell states forgets and what part of the cell state is written to are consolidated into a single gate. Only the portion of the cell state that has been erased is written to. Finally, the entire cell state is given as an output. This is different from the LSTM cell which chooses what to read from the cell state to produce an output. All of these changes together provide a simpler design with less parameters than the LSTM.

Training Model: -

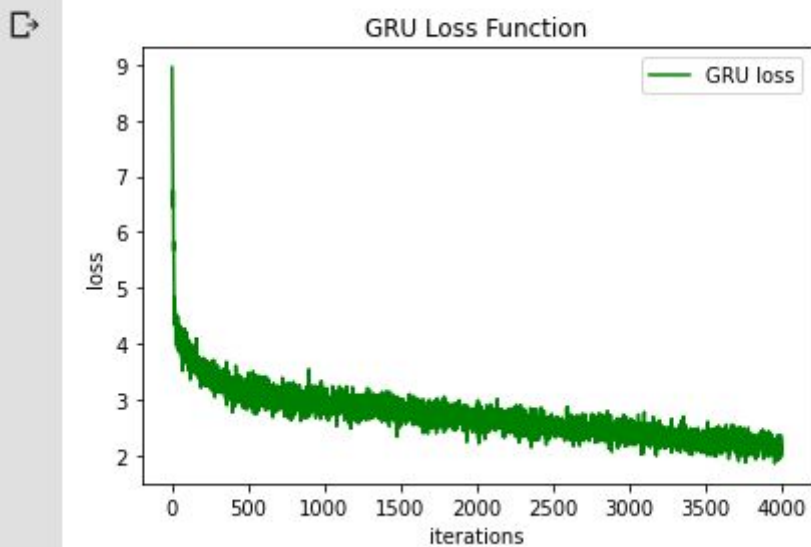
The models are trained on sequences, each containing 4000 iterations. The data is provided to the models as a single batch. The GRU cells are implemented using the default settings of the `torch.nn.GRU(*args, **kwargs)` layers in pytorch as the back end. The Adam optimizer is used with a constant learning rate of 0.001.

Model Configuration

- A (10,5)-dimensional word embedding layer was used to represent the input words.
- Softmax was used on the output layer.
- The model was fit for 4000 iterations of batches where some learning rate decay was performed.
- `input_size` – The number of expected features in the input= 10
- `hidden_size` – The number of features in the hidden state= 10
- A batch-size of 64 sequences was used during training.
- Batches were comprised of sentences with roughly the same length to speed-up computation.
- `teacher_forcing_ratio` = 1.0
- `learning_rate` = 0.0001
- `decoder_learning_ratio` = 5.0
- `n_iteration` = 4000
- `attn_model` = 'dot'
- `hidden_size` = 500
- `encoder_n_layers` = 2
- `decoder_n_layers` = 2
- `dropout` = 0.1

- batch_size = 64
- small_batch_size = 5
- max_target_len: 10

```
plt.plot(res_gru, label='GRU loss',color="g")  
plt.title('GRU Loss Function')  
plt.ylabel('loss')  
plt.xlabel("iterations")  
plt.legend(loc = 'best')  
plt.show()
```



Average loss value of this model

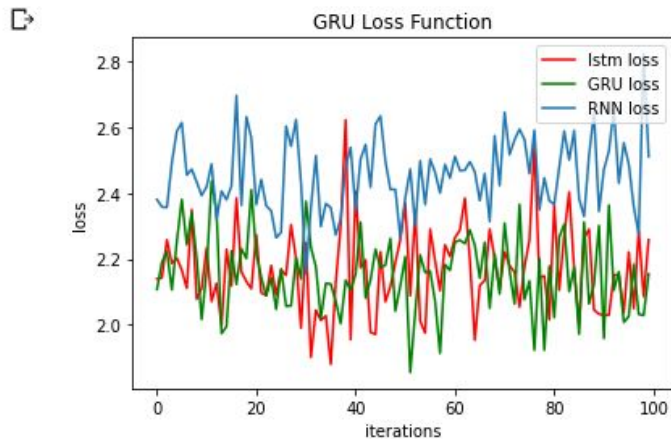
```
[ ] sum(res_gru)/4000
```

```
2.725182879057899
```

Models Comparison – Loss Function

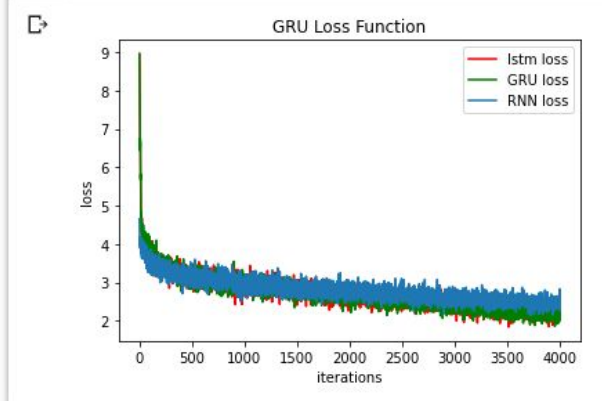
Plotting last 100 iterations loss function values for faster analysis

```
plt.plot(res_lstm[3900:], label='lstm loss',color="r")
plt.plot(res_gru[3900:], label='GRU loss',color="g")
plt.plot(res_rnn[3900:], label='RNN loss')
plt.legend(loc = 'best')
plt.title('GRU Loss Function')
plt.ylabel('loss')
plt.xlabel("iterations")
plt.show()
```



Plotting loss function values vs iterations

```
plt.plot(res_lstm, label='lstm loss',color="r")
plt.plot(res_gru, label='GRU loss',color="g")
plt.plot(res_rnn, label='RNN loss')
plt.legend(loc = 'best')
plt.title('GRU Loss Function')
plt.ylabel('loss')
plt.xlabel("iterations")
plt.show()
```



Comparing average loss function values across models and plot analysis:

- RNN has the highest loss function values compared to LSTM, GRU
- LSTM has high loss function values compared to GRU

- GRU has the lowest loss function value.

Reasons for the expected results:

1. Vanishing gradient is the main reason for high loss function value in RNN encoder and attention RNN decoder mechanism compared to other models.
2. LSTMS can bypass units and thus remember for longer time steps. Thus, have a way to remove some of the vanishing gradients problems. So has less loss function value than RNN
3. The GRU controls the flow of information like the LSTM unit, but without having to use a *memory unit and more efficiently with less complex architecture*. It just exposes the full hidden content without any control. Thus, having the lowest loss value compared to others.

Challenges:

- System crashes when executed in my local system due to memory issues.
- High RAM Allocation is required for fast training. It took around 2.30 hours to train a model with default allocation in google colab.

Part 2: Using BERT to create a QA system to answer covid 19 related questions based on a specific context.

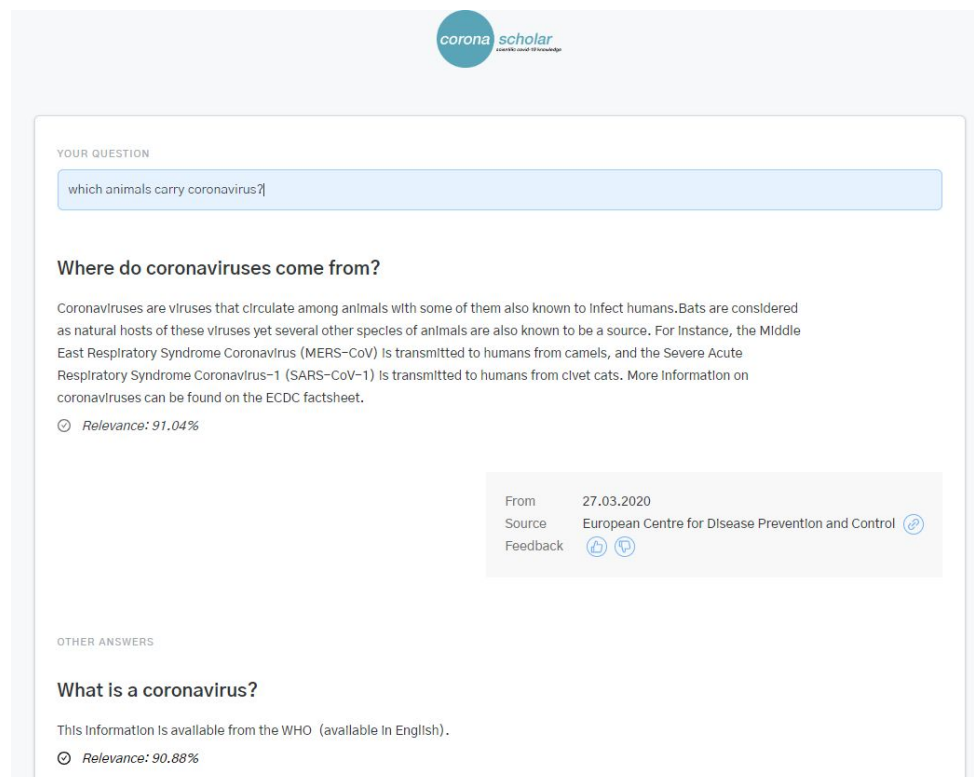
For this task we again go back to our old friend library called simple transformers and now use the `QuestionAnsweringModel`.

Note : This task requires the dataset in a specific format which is compatible to SQUAD styles. Hence we obtained the dataset here from this link.

https://raw.githubusercontent.com/deepset-ai/COVID-QA/master/data/question-answering/200421_covidQA.json

This dataset is created by the deepset-ai team and they have made an interesting product called corona scholar that can answer covid-19 related questions using bert based models. They created a challenge and a dataset called COVID-QA around this challenge and open sourced it in the squad format that can be used with the simple transformers implementation.

This corona scholar is our inspiration and something that we are trying to mimic using a familiar library of simple transformers. The eventual product will be something like one shown in the image below. But for this project we are just going to have a terminal based input and output in google colab as we are not using any api's.



This task is mostly dataset dependent and involves training on a specific context which in our case is corona virus related articles. The SQUAD type data format requires the following

specification from training data. The input data can be in JSON files or in a Python list of dicts in the correct format.

The file should contain a single list of dictionaries. A dictionary represents a single context and its associated questions.

Each such dictionary contains two attributes, the "context" and "qas".

- context: The paragraph or text from which the question is asked.
- qas: A list of questions and answers.

Questions and answers are represented as dictionaries. Each dictionary in qas has the following format.

- id: (string) A unique ID for the question. Should be unique across the entire dataset.
- question: (string) A question.
- is_impossible: (bool) Indicates whether the question can be answered correctly from the context.
- answers: (list) The list of correct answers to the question.

A single answer is represented by a dictionary with the following attributes.

- answer: (string) The answer to the question. Must be a substring of the context.
- answer_start: (int) Starting index of the answer in the context.

Example of the list of dictionaries that can be used as inputs.

```
train_data = [
  {
    'context': "This is the first context",
    'qas': [
      {
        'id': "00001",
        'is_impossible': False,
        'question': "Which context is this?",
        'answers': [
          {
            'text': "the first",
            'answer_start': 8
          }
        ]
      }
    ]
  },
  {
    'context': "Other legislation followed, including the Migratory Bird Conservation Act of 1929, a 1937 treaty prohibiting the hunting of right and gray whales, and the Bald Eagle Protection Act of 1940. These later laws had a low cost",
    'qas': [
      {
        'id': "00002",
        'is_impossible': False,
        'question': "What was the cost to society?",
        'answers': [
          {
            'text': "low cost",
            'answer_start': 225
          }
        ]
      },
      {
        'id': "00003",
        'is_impossible': False,
        'question': "What was the name of the 1937 treaty?",
        'answers': [
          {
            'text': "Bald Eagle Protection Act",
            'answer_start': 167
          }
        ]
      }
    ]
  }
],
...
```

The context in case of COVID-QA is a collection of web scraped and manually collected lists of scientific research articles. The questions and answers are some manually filled in questions and answers by volunteers that can be used to fine tune bert like models.

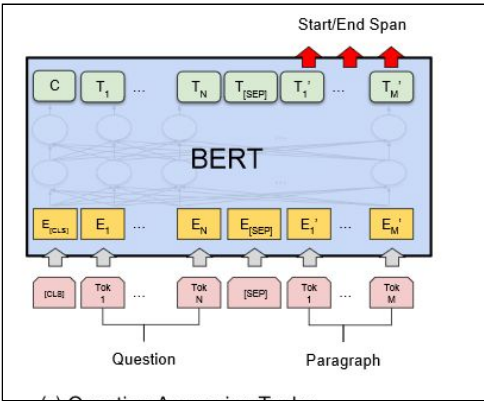
A snapshot of contents of covid-qa dataset is as below.

```
{
  "data": [
    {
      "paragraphs": [
        {
          "qas": [
            {
              "question": "What is the main cause of HIV-1 infection in children?",
              "id": 262,
              "answers": [
                {
                  "text": "Mother-to-child transmission (MTCT) is the main cause of HIV-1 infection in children worldwide. ",
                  "answer_start": 370
                }
              ],
              "is_impossible": false
            },
            {
              "question": "What plays the crucial role in the Mother to Child Transmission of HIV-1 and what increases the risk",
              "id": 276,
              "answers": [
                {
                  "text": "DC-SIGNR plays a crucial role in MTCT of HIV-1 and that impaired placental DC-SIGNR expression increases risk of transmission.",
                  "answer_start": 2003
                }
              ],
              "is_impossible": false
            },
            {
              "question": "How many children were infected by HIV-1 in 2008-2009, worldwide?",
              "id": 278,
              "answers": [
                {
                  "text": "more than 400,000 children were infected worldwide, mostly through MTCT and 90% of them lived in sub-Saharan Africa. ",
                  "answer_start": 2291
                }
              ],
              "is_impossible": false
            },
            {
              "question": "What is the role of C-C Motif Chemokine Ligand 3 Like 1 (CCL3L1) in mother to child transmission of HIV-1?",
              "id": 316,
              "answers": [
                {
                  "text": "High copy numbers of CCL3L1, a potent HIV-1 suppressive ligand for CCR5, are associated with higher chemokine production and lower risk of MTCT of HIV-1 among South African infants",
                  "answer_start": 28143
                }
              ],
              "is_impossible": false
            }
          ]
        }
      ]
    }
  ]
}
```

This image shows a dict containing “data” which has a key called paragraph which contains the data in required format. The part of image above shows only “qas”: section. The image below shows a part of the context from the same paragraph.

```
{
  "question": "How can CCR5's effect in HIV-1 transmission be reduced?",
  "id": 321,
  "answers": [
    {
      "text": "The 32-pb deletion polymorphism in CCR5 has be shown to protect from vertical transmission of HIV-1",
      "answer_start": 27966
    }
  ],
  "is_impossible": false
},
{
  "context": "Functional Genetic Variants in DC-SIGNR Are Associated with Mother-to-Child Transmission of HIV-1\n\nhttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC2752805/\n\nBoily-Larouche, Geneviève; Iscache, Anne-Laure; Zijenah, Lynn S.; Humphrey, Jean H.; Moulard, Andrew J.; Ward, Brian J.; Roger, Michel\n2009-10-07\nDOI:10.1371/journal.pone.0007211\n\nlicense:cc-by\n\nAbstract: Mother-to-child transmission (MTCT) is the main cause of HIV-1 infection in children worldwide. Given that the C-type lectin receptor, dendritic cell-specific ICAM-grabbing non-integrin-related (DC-SIGNR, also known as CD209L or liver/lymph node-specific ICAM-grabbing non-integrin (L-SIGN)), can interact with pathogens including HIV-1 and is expressed at the maternal-fetal interface, we hypothesized that it could influence MTCT of HIV-1. METHODS AND FINDINGS: To investigate the potential role of DC-SIGNR in MTCT of HIV-1, we carried out a genetic association study of DC-SIGNR in a well-characterized cohort of 197 HIV-infected mothers and their infants recruited in Harare, Zimbabwe. Infants harbouring two copies of DC-SIGNR H1 and/or H3 haplotypes (H1-H1, H1-H3, H3-H3) had a 3.6-fold increased risk of in utero (IU) (P = 0.013) HIV-1 infection and a 5.7-fold increased risk of intrapartum (IP) (P = 0.025) HIV-1 infection after adjusting for a number of maternal factors. The implicated H1 and H3 haplotypes share two single nucleotide polymorphisms (SNPs) in promoter region (p-150A) and Intron 2 (Int2-180A) that were associated with increased risk of both IU (P = 0.045 and P = 0.003, respectively) and IP (P = 0.025, for Int2-180A) HIV-1 infection. The promoter variant reduced transcriptional activity in vitro. In homozygous H1 infants bearing both the p-150A and Int2-180A mutations, we observed a 4-fold decrease in the level of placental DC-SIGNR transcripts, disproportionately affecting the expression of membrane-bound isoforms compared to infant noncarriers (P = 0.011). CONCLUSION: These results suggest that DC-SIGNR plays a crucial role in MTCT of HIV-1 and that impaired placental DC-SIGNR expression increases risk of transmission.\n\nNext: Without specific interventions, the rate of HIV-1 mother-to-child transmission (MTCT) is approximately 15-45% [1] . UNAIDS estimates that last year alone, more than 400,000 children were infected worldwide, mostly through MTCT and 90% of them lived in sub-Saharan Africa. In the most heavilyaffected countries, such as Zimbabwe, HIV-1 is responsible for one third of all deaths among children under the age of five. MTCT of HIV-1 can occur during pregnancy (in utero, IU), delivery (intrapartum, IP) or breastfeeding (postpartum, PP). High maternal viral load, low CD4 cells count, vaginal delivery, low gestational age have all been identified as independent factors associated with MTCT of HIV-1 [1] . Although antiretrovirals can reduce MTCT to 2%, limited access to timely diagnostics and drugs in many developing world countries limits the potential impact of this strategy. A better understanding of the mechanisms acting at the maternal-fetal interface is crucial for the design of alternative interventions to antiretroviral therapy for transmission prevention.\n\nDendritic cell-specific ICAM-grabbing non-integrin-related (DC-SIGNR, also known as CD209L or liver/lymph node-specific ICAM-grabbing non-integrin (L-SIGN)) can interact with a plethora of pathogens including HIV-1 and is expressed in placental capillary endothelial cells [2] . DC-SIGNR is organized in three distinct domains, an N-terminal cytoplasmic tail, a repeat region containing seven repeat of 23 amino acids and a C-terminal domain implicated in pathogen binding. Alternative splicing of DC-SIGNR gene leads to the production of a highly diversify isoforms repertoire which includes membrane-bound and soluble isoforms [3] . It has been proposed that interaction between DC-SIGNR and HIV-1 might enhance viral transfer to other susceptible cell types [2] but DC-SIGNR can also internalize and mediate proteasome-dependant degradation of viruses [4] that may differently affect the outcome of infection.\n\nGiven the presence of DC-SIGNR at the maternal-fetal interface and its interaction with HIV-1, we hypothesized that it could influence MTCT of HIV-1. To investigate the potential role of DC-SIGNR in MTCT of HIV-1, we carried out a genetic association study of DC-SIGNR in a well-characterized cohort of HIV-infected mothers and their infants recruited in Zimbabwe, and identified specific DC-SIGNR variants associated with increased risks of HIV transmission. We further characterized the functional impact of these genetic variants on DC-SIGNR expression and show that they affect both the level and type of DC-SIGNR transcripts produced in the placenta.\n\nSamples consisted of stored DNA extracts obtained from 197 mother-child pairs co-enrolled immediately postpartum in the ZVITAMBO Vitamin A supplementation trial (Harare, Zimbabwe) and followed at 6 weeks, and 3-monthly intervals up to 24 months. The ZVITAMBO project was a randomized placebocontrolled clinical trial that enrolled 14,110 mother-child pairs, between November 1997 and January 2008, with the main objective of investigating the impact of immediate postpartum vitamin A supplementation on MTCT of HIV-1. The samples used in the present study were from mother-child pairs randomly assigned to the placebo group of the ZVITAMBO project. Antiretroviral prophylaxis for HIV-1-positive antenatal women was not available in the Harare public-sector during ZVITAMBO patient recruitment. The samples were consecutively drawn from two groups: 97 HIV-1-positive mother/HIV-1-positive child pairs and 100 HIV-1-positive mother/HIV-negative child pairs.
```

Based on these context and QA pairs we can train the bert model shown in image below.



The data from COVID-QA can be converted in required format as shown in code below.

Code for installing necessary libraries and generation of logs for debugging.

```
# install prereqs
!pip install transformers
!pip install seqeval
!pip install tensorboardx

# install apex
!git clone https://github.com/NVIDIA/apex
%cd apex
!pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_ext" ./

# install simple transformers
!pip install simpletransformers
```

```
from simpletransformers.question_answering import QuestionAnsweringModel
import json
import os
import logging

logging.basicConfig(level=logging.INFO)
transformers_logger = logging.getLogger("transformers")
transformers_logger.setLevel(logging.WARNING)
```

Below is the code for creating a list of dictionaries in squad format that can be used by the model.

```
with open('/content/drive/My Drive/200421_covidQA.json') as f:
    data = json.load(f)

#train_data = data['data'][0]['paragraphs']
train_data = []
for i in range(len(data['data'])):
    train_data.append(data['data'][i]['paragraphs'][0])
```

Below is a paragraph from one context that we will use to ask questions from this model.

Creation of a sample context on which we can ask questions.

```
context = train_data[77]['context']
print(context)
#for i in range(len(data['data'])):
#    print(data['data'][i]['paragraphs'][0])
```

Potential Maternal and Infant Outcomes from (Wuhan) Coronavirus 2019-nCoV Infecting Pregnant Women: Lessons from SARS, MERS, and Other Human Coronavirus Infections

The text from this context is as shown below. The highlighted section where our answer lies. The question will be a variant of **which animals carry coronavirus?**

```
context = train_data[77]['context']
print(context)
#for i in range(len(data['data'])):
#    print(data['data'][i]['paragraphs'][0])
```

Potential Maternal and Infant Outcomes from (Wuhan) Coronavirus 2019-nCoV Infecting Pregnant Women: Lessons from SARS, MERS, and Other Human Coronavirus Infections

<https://doi.org/10.3390/v12020194>

SHA: 779c1b5cb3afe3d50219aa2af791014a22eb355a

Authors: Schwartz, David A.; Graham, Ashley L.
Date: 2020
DOI: 10.3390/v12020194
License: cc-by

Abstract: In early December 2019 a cluster of cases of pneumonia of unknown cause was identified in Wuhan, a city of 11 million persons in the People's Republic of China. Further in
Text: Coronaviruses are spherical, enveloped, and the largest of positive-strand RNA viruses. They have a wide host range, including birds, farm animals, pets, camels, and bats, in which
In humans, they are a cause of mild illnesses including the common colds occurring in children and adults, and were believed to be of modest medical importance. However, two zoonotic coron
Viruses 2020, 12, 194 3 of 16 epidemic. An additional 307 cases of 2019-nCoV infection have occurred among 24 other countries outside of China [12] . (Figure 1) At the meeting of the Emer
Pneumonia arising from any infectious etiology is an important cause of morbidity and mortality among pregnant women. It is the most prevalent non-obstetric infectious condition that occur
Pneumonia arising from any infectious etiology is an important cause of morbidity and mortality among pregnant women. It is the most prevalent non-obstetric infectious condition that occur
The SARS epidemic began quietly at the turn of the 21st century. In November 2002, a cook in Guangdong Province, China, died from an unidentified illness. He had worked at a restaurant in
Although there were relatively few documented cases of SARS occurring during pregnancy, several case reports and small clinical studies have described the clinical effects in pregnant wom
The clinical outcomes among pregnant women with SARS in Hong Kong were worse than those occurring in infected women who were not pregnant [32] . Wong et al. [29] evaluated the obstetrical
A case-control study to determine the effects of SARS on pregnancy compared 10 pregnant and 40 non-pregnant women with the infection at the Princess Margaret Hospital in Hong Kong [27, 33].
Maxwell et al. [32] reported 7 pregnant women infected with SARS-CoV who were followed at a designated SARS unit-2 of the 7 died (CFR of 28%), and 4 (57%) required ICU hospitalization and
Zhang et al. [34] described SARS-CoV infections in 5 primagravidas from Guangzhou, China at the height of the SARS epidemic. Two of the mothers became infected in the 2nd trimester, and 3
Two pregnant women with SARS were reported from the United States. In a detailed case report, Robertson et al. [35] described a 36-year-old pregnant woman with an intermittent cough of app
From Canada, Yudin et al. [38] reported a 33-year-old pregnant woman who was admitted to the hospital at 31 weeks gestation with a fever, dry cough, and abnormal chest radiograph demonstri

Below is the actual question and answer that was provided in training set.

Checking on some questions on this sample context that obert model is being fine tuned on.

```
[4] train_data[77]['qas']
```

```
[{'answers': [{'answer_start': 1930,
  'text': 's are spherical, enveloped, and the largest of positive-strand RNA v'}],
  'id': 2199,
  'is_impossible': False,
  'question': 'What are coronaviruses?'},
 {'answers': [{'answer_start': 2039,
  'text': 'uding birds, farm animals, pets, camels, an'}],
  'id': 2200,
  'is_impossible': False,
  'question': 'What animals can carry coronavirus?'}]
```

Below is a code to store the data for further use in models and also to create a basic bert based model.

```
#!/usr/bin/env python
#%cd ..
os.makedirs('data', exist_ok=True)
# Save as a JSON file
with open('data/train.json', 'w') as f:
    json.dump(train_data, f)

# Create the QuestionAnsweringModel
model = QuestionAnsweringModel('bert', 'bert-base-uncased', args={'reprocess_input_data': True, 'overwrite_output_dir': True})

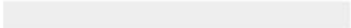
# Train the model with JSON file
model.train_model('data/train.json')
```


Below is the output after training has started.

It displays any settings for optimizer and amount of time for each epochs. In this example case we are just using default settings and it takes around 25 minutes for this model to train.

```
INFO:simpletransformers.question_answering.question_answering_model: Converting to features started.
100%|██████████| 1279/1279 [05:20<00:00, 3.99it/s]
Selected optimization level 01: Insert automatic casts around Pytorch functions and Tensor methods.

Defaults for this optimization level are:
enabled                : True
opt_level              : O1
cast_model_type        : None
patch_torch_functions  : True
keep_batchnorm_fp32    : None
master_weights         : None
loss_scale             : dynamic
Processing user overrides (additional kwargs that are not None)...
After processing overrides, optimization options are:
enabled                : True
opt_level              : O1
cast_model_type        : None
patch_torch_functions  : True
keep_batchnorm_fp32    : None
master_weights         : None
loss_scale             : dynamic


Epoch: 0%  0/1 [00:00<?, ?it/s]

Current iteration: 45%  4042/8919 [09:57<11:48, 6.88it/s]
```

The tuned model can then be used to answer questions. Below is some code based on the context introduced earlier. The model is able to give a very good answer when the question matches exactly. But the answer changes to birds, farm animals, pets, camels and bats.


Even slight variation in question leads to no answer. This offers an opportunity for tuning the model a little better and this will be tried up next.

```
[17] # Making predictions using the model.
      to_predict = [{'context': context, 'qas': [{'question': 'what animals can carry coronavirus?', 'id': '0'}]}]
      to_predict
      print(model.predict(to_predict))
```

```
↳ INFO:simpletransformers.question_answering.question_answering_model: Converting to features started.
100%|██████████| 1/1 [00:00<00:00, 2.28it/s]
100%  11/11 [00:00<00:00, 43.88it/s]

[{'id': '0', 'answer': 'birds, farm animals, pets, camels, and bats'}]
```

```
[19] # Making predictions using the model.
      to_predict = [{'context': context, 'qas': [{'question': 'what animals carry coronavirus?', 'id': '0'}]}]
      to_predict
      print(model.predict(to_predict))
```

```
↳ INFO:simpletransformers.question_answering.question_answering_model: Converting to features started.
100%|██████████| 1/1 [00:00<00:00, 2.37it/s]
100%  11/11 [00:00<00:00, 44.68it/s]

[{'id': '0', 'answer': ''}]
```

Overall results and observations from QA system:

- It was very easy to implement a QA system for covid related data using simple transformers. The quality of predictions were not very good but offered an opportunity for improvement.
- In simple bert and distilbert based models the predictions were not good.
- However we saw that using a large bert model led to improvements in answering questions which were similar. Unfortunately we missed the output due to timeout on colab but we will update the report once we are able to get the answers again.
- The result was the same if the following questions were asked.
 1. What animals carry viruses?
 2. What animals can carry viruses?
 3. What animals can carry coronavirus?
 4. Which animals can carry coronavirus?

This meant that the model was able to understand that “which” and “what” can be used interchangeably.

The model also understood that “can” may be omitted and answered the question appropriately. The model was also able to understand that virus and coronavirus meant similar things.

Overall observations and conclusions from all implementations.

- We got familiar with Pytorch, CNNs, RNNs, LSTMs, GRUs, and BERT models for applications in NLP.
- Major limitation was computing infrastructure as most of the models took 2-5 hours for training just one epoch
- CNNs are faster to implement and can give pretty good output for proof of concept type applications in NLP when combined with word embeddings.
- KimCNN is an interesting architecture that we explored for text data and seems to work really well even with non-context specific embeddings like fastText.
- Simple transformers is an amazing library that allows us to make applications like sentence classification, QA system and NER type systems with just a few lines of code and allows decent customization options which work flawlessly with big datasets.
- GRUs are faster in implementation compared to LSTMs and RNNs as they solve vanishing gradient problem
- Attention mechanism helps to access all hidden context state vectors from encoders to select the best probable word from hidden state encoders as next input.
- Pytorch has issues with memory management as it stores a lot of tensor for backpropagation.
- Creation of batches helps reduce memory issues during training.
- We can indeed just start with attention models and implement many tasks in NLP.
- When we move from RNN to LSTM, we are introducing more & more controlling knobs, which control the flow and mixing of Inputs as per trained **Weights**. And thus, bringing in more flexibility in controlling the outputs.

Acknowledgements:

We would like to thank Dr. Apurva Narayan for providing an opportunity to explore these areas of NLP and deep learning. We are also thankful to the work done by kaggle teams, Conversation AI team, pytorch team, hugging face team, simple transformers team, deep set-ai team, chris mccormick, roman orac, ThilinaRajapakse and Vladimir Demidov for sharing their work on CNNs, RNNs, LSTMs, textCNN, BERT, covid-data, and transformers etc.

References:

1. <https://www.kaggle.com/yekenot/textcnn-2d-convolution> (toxicity data classification with cnn and word embeddings - Vladimir Demidov's kaggle submission)
2. <https://towardsdatascience.com/identifying-hate-speech-with-bert-and-cnn-b7aa2cddd60d> (classification of hate speech using bert and CNN blogpost by roman rac)
3. <https://arxiv.org/pdf/1706.03762.pdf> (attention is all you need - original article)
4. <https://arxiv.org/pdf/1810.04805.pdf> (bert - original article)
5. <https://github.com/deepset-ai/COVID-QA/tree/master/data/question-answering> (covid QA dataset by deepset-ai)
6. <https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/notebooks> (toxicity dataset by kaggle)
7. <https://github.com/ThilinaRajapakse/simpletransformers> (simple transformers library - ThilinaRajapakse)
8. <https://covid.deepset.ai/home> (corona scholar page for covid related questions)
9. <https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/> (bert word embeddings article by chris mccormick)
10. <http://mccormickml.com/2019/07/22/BERT-fine-tuning/> (bert fine tuning article for sentence classification by chris mccormick)
11. <https://pytorch.org/docs/stable/nn.html> (Pytorch official document)
12. https://pytorch.org/tutorials/beginner/chatbot_tutorial.html (chatbot tutorial)
13. <https://www.udemy.com/course/applied-deep-learning-build-a-chatbot-theory-application/learn/lecture/17090528#overview> (udemy course ref)