Tutorial: Financial News Summarization using Transformer

Parth Mihir Patel

27/06/2022

Introduction

Transformer architecture, which is based on self-attention mechanism, revolutionised the field of NLP in 2017. It overcame many of the limitations of sequential and iterative approach of the previous popular architectures like LSTM. Generative Pretrained Transformer (GPT), which is a type of transformer model, is one of the most powerful neural network architectures for the purpose of text summarisation. This tutorial elaborates how and why GPT-2 can be used for financial news summarisation.

Summarization Techniques

Extractive

Extractive summarization is similar to how we highlight lines on paper. It calculates the importance of sentences in an article, and extracts them. It does not construct any new words or phrases.

Abstractive

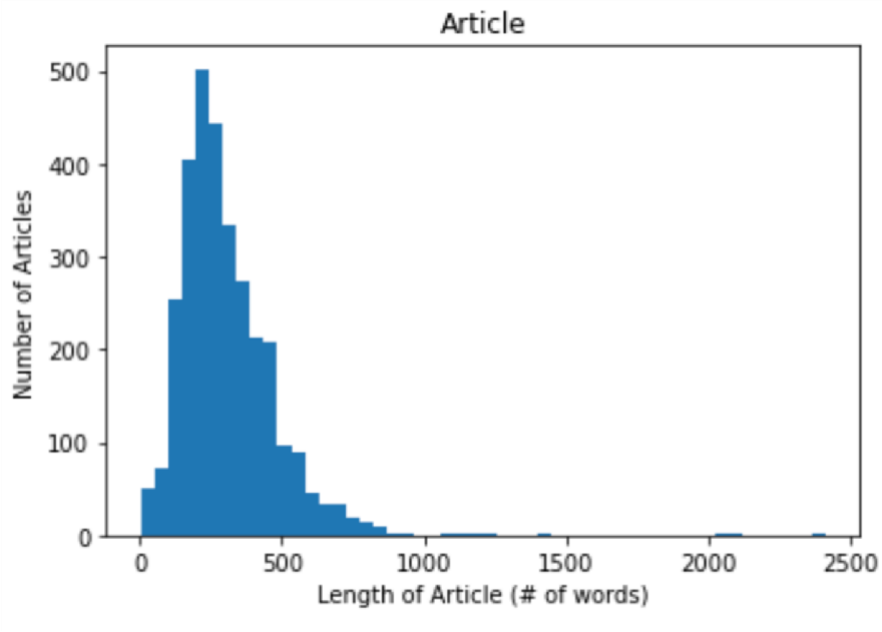
Abstractive summarization tries to understand meaning/point of each sentence and context of each word, in the article. Based on that, it creates new phrases in a meaningful way, such that they carry the same meaning in a rephrased shorter way.
It is more complex than extractive summarization. In this tutorial we focus on this type.

Dataset Overview

(Note: All the graphs have been created by me using matplotlib)

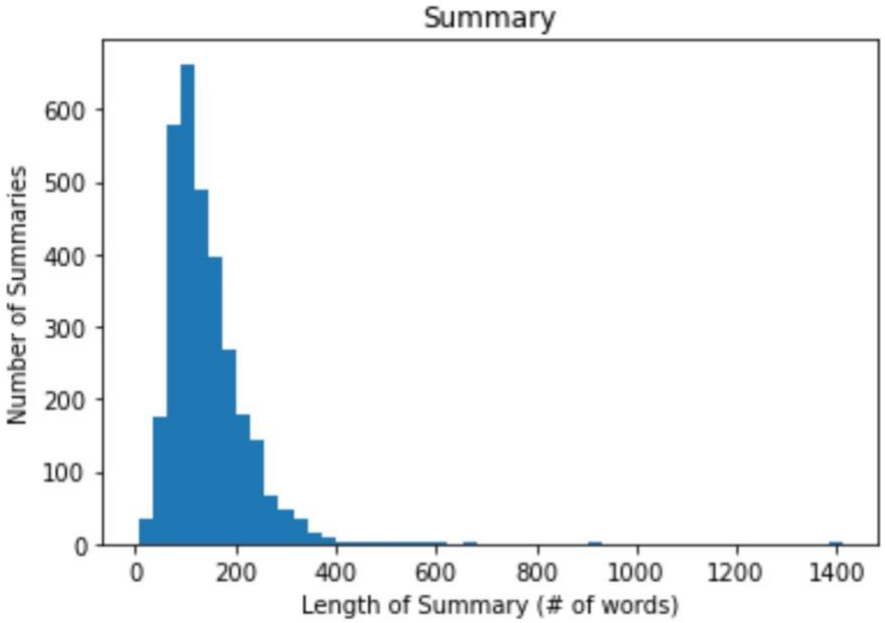
BBC News Dataset is a csv with 2224 tuples. It has 3 features - ArticleText, Class and Summary.

The length(word-count) of Article Texts is distributed as follows:



(fig 1.1 : created using matplotlib)

Summary will be the target variable. Length(Word-count) of summaries is distributed as follows:



(fig 1.2 : created using matplotlib)

Number of tuples, categorised by ‘Class' feature are as follows:

(Tuples which have empty Class feature are not counted)

```
business          510
politics          417
tech              401
entertainment     386
Name: Class, dtype: int64
```

(fig 1.3)

Text Preprocessing

Tokenization

Tokenization is the first step in pre-processing. It is the process of breaking paragraphs of texts into discrete elements. An array/list of discrete elements is returned. If we want our discrete elements to be words, we use whitespace as the delimiter, and if we want sentences, we use stopmark.

Example:

Input- “today is a good day. Today’s weather is good.”

Output- [“Today”, “is”, “a’, “good”, “day”, “.”, “Today’s”, “weather”, “is”, “good”]

In the tutorial I’ve used *spacy.tokenizer.Tokenizer* for tokenization.

```
from spacy.tokenizer import Tokenizer
from spacy.lang.en import English lang
= English() #Load language tokenizer =
Tokenizer(lang.vocab)#Create tokenizer
with Eng vocab article_tokens =
tokenizer(articleText)
```

Stemming/Lemmatization

Stemming and Lemmatization are processes for removing inflections(ing, er, etc) from tokens, thereby reducing words to their base forms.

Examples:

Book's -> Book

Books -> Book

Booking -> Book

Booker -> Book

What's the difference between stemming and lemmatization?

Stemming uses unsophisticated heuristic methods to simply chop off the ends of tokens. But this sometimes ends up removing base parts of words.
Lemmatization performs advanced vocabulary and morphological analysis of words. It also considers the context of the word. This makes sure that only the inflections are removed and not the base parts. The base parts are also known as the *lemma* .

For example, for the word *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw* depending on whether the use of the token was as a verb or a noun.

In the tutorial I've used *nltk.stem.WordNetLemmatizer* for lemmatization.

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatized_article_tokens = [] for tok
in article_tokens:
      lemmatized_article_tokens.append(lemmatizer.lemmatize(tok))
```

Encoding

Words are categorical data. Since we can't feed categorical data into our neural network, we need to convert each word into a vector.
One Hot Encoding is the simplest way to encode. Each word is represented with a vector of size n, where n is number of words in our vocabulary (vocabulary means list of all the unique words in our dataset). Values of all dimensions in vector are zero except the word which it is referring to (which is set to 1).

Problem with One Hit Encoding

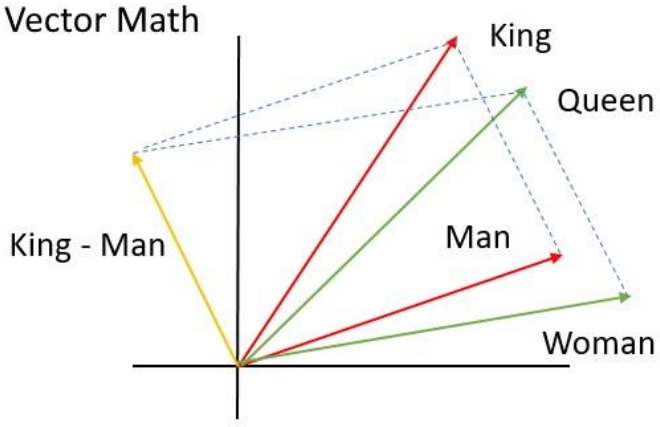
- The dimension of our vectors is as big is size of our vocabulary. This is a waste of space, and it increases the space and time complexity of our algorithms ("Curse of Dimensionality")
- Vectors aren't contextualised. Ideally, the vectors for words "Good" and "Great" should have very small cosine distance in vector space, but in 1HE, it is totally random.

To solve these issues, we use Word2Vec.

Word2Vec

Word2Vec algorithms create vectors (word embeddings) that are Contextualised. Embeddings have dimension of n, where n is usually between 100 to 300. Each dimension in vector has value between 0 and 1.
Word2Vec helps establish a word's association with other words (e.g. "queen" is to "female" what "king" is to "male"). This is because the embedding vectors are highly contextualised. Vectors of words that have similar meanings, have small cosine distance in the vector space.
Eg: Vectors of words like "Great" and "Awesome" will have the least cosine distance with the vector of word "Good":

One useful side effect of this is that we can calculate associations between words in interesting ways. For example,
Subtracting the embedding of Male from the embedding of King, and adding the embedding of Female to it, will give us the embedding of Queen.
V(King) – V(Male) + V(Female) = V(Queen)

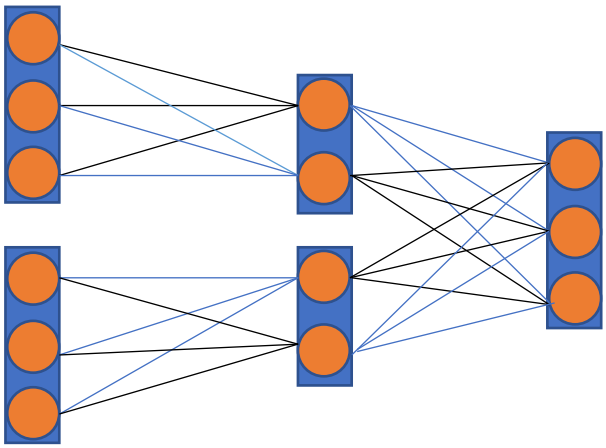


(fig 1.4, source: https://medium.com/analytics-vidhya/word-embeddings-in-nlp-word2vec-glove-fasttext-24d4d4286a73)

There are two main algorithms for Word2Vec: CBOW and SkipGram.

CBOW

It is based on a neural network which tries to predict the one hot encoding of target word based on one hot encodings of its surrounding words.
Consider the sentence 'GDP is on big decline'. The model converts this sentence into word pairs in the form (contextword, targetword). The word pairs would look like this: ([GDP, on], is), ([is, big], on),([on, decline], big). With these word pairs, the model tries to predict the target word considered the context words.



(fig 1.5, created in MS Word)

The input layer will be One Hot Encodings of 2 context vectors. Each of these will be connected to a hidden layer with k neurons each (k = 2 in the above figure). Note that these two hidden layers will be parallel and will share the same weights. These 2 parallel hidden layers will be concatenated and fed into the output dense layer which has softmax activation and n neurons where n is the size of our vocabulary (n = 3 in the above figure).

As we train this network on all the word pairs, the weights of the hidden layer will be learned, and the hidden layers will correspond to our learned word2vec embeddings. Embeddings will have k dimensions.

In the tutorial I've used *torch.nn.Embedding* for creating embeddings.

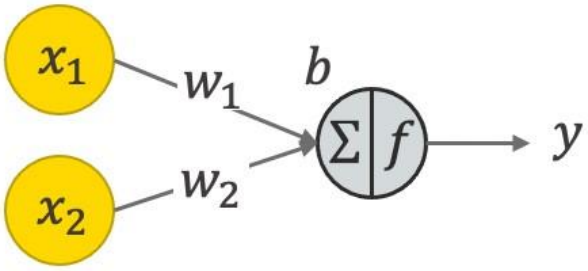
```
from torch.nn import Embedding
```

```
embeddings = Embedding(vocab_size, embedding_size
```

Modelling

Will first start with basic concepts of DL to build up to Transformer model which I've used.

Single Neuron



(fig 1.6, source: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks)

Let's first walk through how a single neuron works. Let's say there are n features. We multiply each feature by a corresponding weight, and add them up. The resultant value is called preactivation value.

$h = w_1x_1 + w_2x_2 + \ldots + w_nx_n$

Now, we pass h through an activation function, let's say Sigmoid. It would squash the preactivation value between 0 and 1, essentially giving us a probability score.

$a = 1 / (1+e^h)$

Why do we need activation functions?

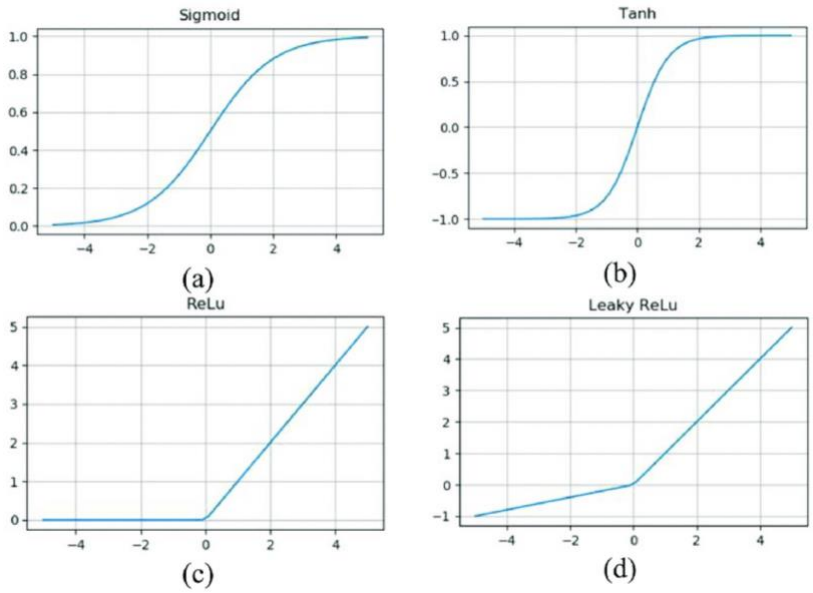
- To prevent super large outputs (to prevent exploding gradients)
- To map non-linearities.

Sigmoid isn't the only activation function. Other activation functions are TanH, ReLU, Leaky ReLU, etc.

ReLU – max(0,z) , that is, it outputs for all the inputs less than or equal to 0, and the same number if the input is > 0.

LeakyReLU – max(0.1x, x) , that is, if the input is <= 0, then the output would be 0.1 * input.

And if the input is > 0 , the output would be same as input.



(fig 1.7, source: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks)

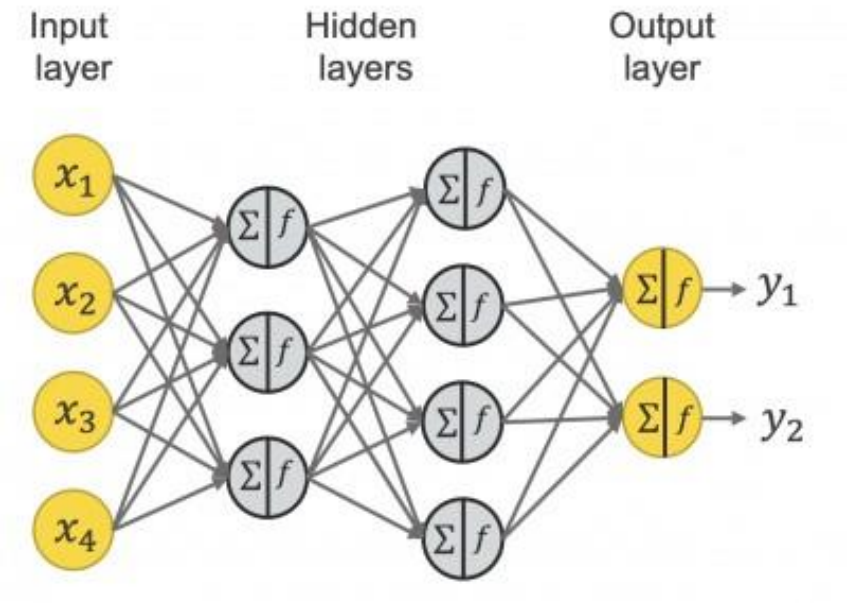
Gradient Descent

How do we decide the weights such that our neuron is able to map a relationship as accurately as possible? We learn them from training data.

- To begin with, we randomly initialize our weights (or use XavierInitialization).
- Now, we feed a training tuple's independent variables into our neuron. We compare the predicted output of neuron with the real target variable in our training tuple.
- We calculate squared error loss by $(y - ybar)^2$.
- Take the partial derivative of loss wrt each weight parameter using chain rule, and subtract them from weights.
- This process is repeated for n epochs.

Neural Network

A single neuron can't map extremely non-linear relationships. To achieve this, we need Neural Network.



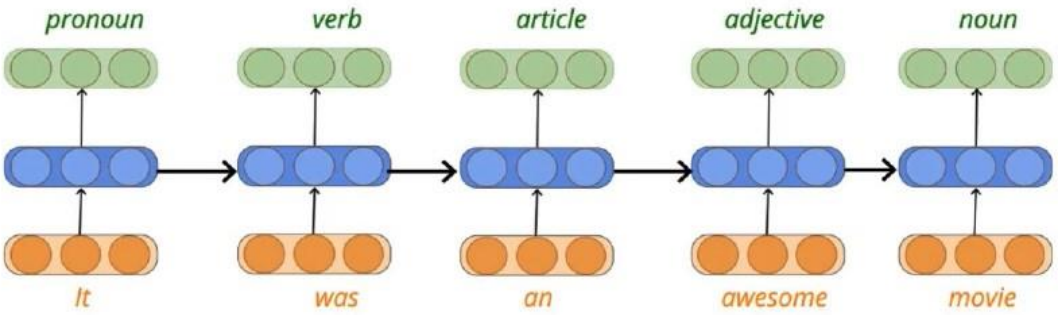
(fig 1.8, source: https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks)

A neural network is basically a stack of neutrons.

- There are multiple neurons in the 1st layer. Every neuron has different set of weights.
- Each neuron in hidden layer has a weighted connection to every neuron in previous layer.
- In the output layer, softmax activation is used to get the probability distribution of predicted classes.
- The loss is calculated by comparing predicted probabilities with ground truths in training dataset.
- The partial derivative is taken with every weight in every neuron using chain rule, and subtracted from current weight. This is done for n epochs.

Recurrent Neural Network

When we are dealing with a sequence of inputs (such as a sentence), we need to use recurrent neural networks.



(fig 1.9, source: https://www.pngfind.com/mpng/himiTxo_recurrent-neural-network-unfold-recurrent-neural-network-diagram/)

RNNs help us generate hidden representations of words that are context sensitive. We can't just straight out generate outputs for words, we need to contextualize them in order to generate correct outputs. For example, network's output generated for the word *"Bank"* needs to be different for "Money in bank", than for "Kids at river bank", even though it is the same word.
To achieve this, in RNNs, we iteratively feed the embedding of the current word of sentence as input, but the current iteration's hidden layer is also connected to the previous iteration's hidden layer. This way, the current iteration's output is affected by hidden representations of the words in previous iterations. It helps in contextualisation.

Problems with RNN

- It works iteratively, word by word. We can't feed the whole sentence all at once. This makes it slow.
- Can't model a future word's influence on the current word, as it is sequential.

To solve these problems, Vaswani et al. introduced Transformer model in 2018, in the paper "Attention is all you need". It uses the concept of Self Attention Mechanism.

Attention Mechanism

We know the kinds of embeddings that word2vec produces. When the vectors(embeddings) of words that have similar meanings are dot multiplied, the resultant magnitude would be greater than if the words were dissimilar. We can use this fact to calculate weights and contextualise all words in our input.

Let's say we have 3 vectors, A, B and C. We want to contextualize each of them.

Let's first contextualize the vector A.

-Take the dot product of A wrt A ($A.A = w_1$), dot product of A wrt B ($A.B = w_2$), and wrt C ($A.C = w_3$)

-Normalize w_1, w_2 and w_2 such that $w_1 + w_2 + w_3 = 1$

-Now, $A_{new} = A*w_1 + B*w_2 + C*w_3$

This process will be repeated for other vectors as well. In the end we will obtain contextualised vectors A_{new}, B_{new} and C_{new}.

The thing here is, as we can see, the weights (w_n) are pre-calculated, and don't require to be learned/updated during network training process. That's why it is called "self" attention.

But, to make our model more flexible and sophisticated, we need to introduce some weights in the system which are actually learned during training.

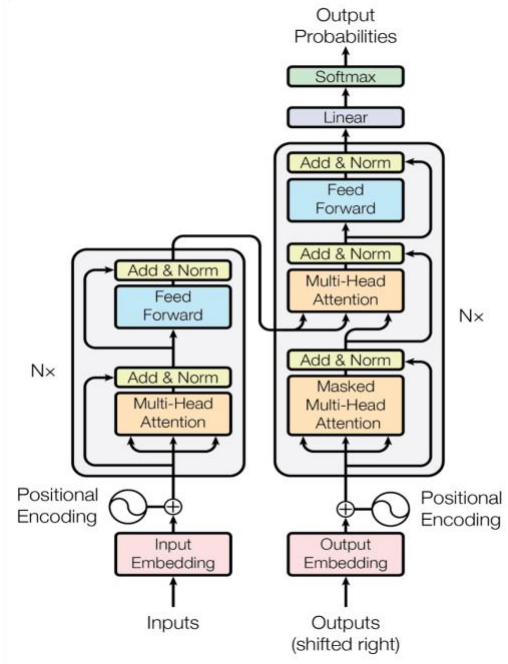
Let's introduce 3 weight matrices – Query(Q), Key(K) and Value(V). Each will have dimension k * k, where k is the dimension of embeddings. Now the updated process for obtaining contextualised vectors will be like this -

-Take the dot product of A wrt A ($Q.A.K.A = w_1$), dot product of A wrt B ($Q.A.K.B = w_2$), and wrt C ($Q.A.K.C = w_3$)

-Normalize w_1, w_2 and w_2 such that $w_1 + w_2 + w_3 = 1$

-$A_{new} = V*A*w_1 + V*B*w_2 + V*C*w_3$

Transformer Model



(fig 1.10, source: Vaswani et al. 2018)

It is divided into two components – Encoder and Decoder.

Encoder

Its job is to contextualise all the input embeddings.

In the input, we feed entire text (embeddings of all words in text) all at once. Since the embeddings don't have positional information, we add positional vectors to them. That's because positional information is important for contextualisation.

Positional vectors are based on the position of word in the text. The original paper uses this formula to get positional vectors for each position:

$$p_{i,j} = \begin{cases} \sin\left(\frac{i}{10000^{\frac{j}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{\frac{j-1}{d_{emb\_dim}}}}\right) & \text{if } j \text{ is odd} \end{cases}$$

(fig 1.11, source: "Attention is All You Need" paper)

Positional vectors are added to embeddings, hence we obtain positional embeddings. Next, pass positional embeddings though an Attention Mechanism Block. It will output contextualised embeddings. Next, we pass these contextual embeddings to a fully connected layer whose neurons use ReLU activation.

Decoder

For i iterations, where i = number of words that we want in our summary,

1) In the input, we feed **target** variable's entire text(embeddings of all words in text) all at once. We compute positional embeddings the way that we saw in encoder.
2) If we are currently at k^{th} iteration, we keep only first k positional embeddings and mask out the rest.
3) We pass these embeddings through attention mechanism block. It will give us contextualised embeddings.

4) The next attention mechanism block will computationally combine the contextualised embeddings from previous attention block in decoder, as well as the contextualized embeddings of encoder.
5) We pass the contextual embeddings outputted by it, to a fully connected layer whose neurons use softmax activation. This layer will have n neurons were n = number of words in our vocabulary.
6) The word corresponding to the output layer's neuron which has the highest value, will be appended to our summary.

Training

Of the 2224 tuples in our dataset, 2000 will be used for training. Since we can't feed all 2000 tuples of dataset all at once into the model in one iteration (due to RAM limitations), we need to perform batching.
PyTorch provides Dataset and DataLoader classes for performing batching. Dataset class provides an interface between the data source and data loader.
The first thing I did was to create a NewsData class which inherits Dataset. It contains a __get_item__ method which returns encoded x and y of ith tuple.

Next, I created an instance of class DataLoader, specifying Dataset (NewsData in our case) and batch size in its constructor.
This data loader would return random batches of size n whenever called.

Now we need to define loss function and optimiser (optimiser is used for performing gradient descent and updating weights).
I'm using CrossEntropyLoss as the loss function and Adam Optimiser as gradient descent algorithm. Adam optimiser is a more sophisticated version of gradient descent which introduces the concept of momentum. It takes a weighted average of gradients of previous n timesteps. This makes the descent process to accelerate when the slope is steep.

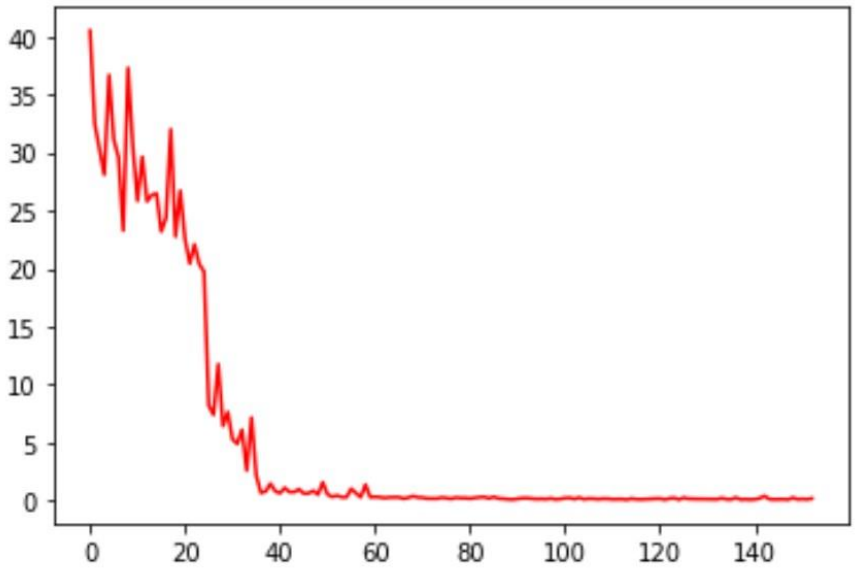
Now, For 150 iterations,

- DataLoader will load random batch
- The tuples of this batch would be fed into the model
- the model's outputs will be compared with real summaries, loss will be calculated
- gradient descent would be performed using optimizer, to update the weights

Outcome

The loss declined as follows during 150 epochs:

```
[<matplotlib.lines.Line2D at 0x7f7d0b9aa290>]
```



(image 1.12, made in matplotlib)

Sample generated text after 25 epochs:

```
generated_summary

 the the the Army the the of the the army the the army the the the the the army the the the the the army the the the the the
the the the the the the

actual_summary

"They are very much not for the good and will destroy Scotland's regiments by moulding them into a single super regiment whic
h will lead to severe recruitment problems, a loss of local connections to those regiments and a loss to Scotland of an impor
tant part of her heritage and, most importantly, her future - the regiments are the envy of armies around the world."The prop
osals to either merge or amalgamate the six regiments into a super regiment sparked a political outcry, with Labour backbench
ers
```

Sample generated text after 150 epochs:

```
generated_summary

Under their vision, it would be one of five in the new super regiment.The proposals to either merge or amalgamate the six reg
iments into a super regiment sparked a political outcry, with Labour backbenchers and opposition politicians opposing the pla
n.The proposals have faced stiff opposition from campaigners and politicians alike.The government and Army Board have spent t
he past four months attempting to trick serving soldiers and the public into thinking their planned changes for the Scottish
regiments are for the good of the Army and for that

actual_summary

"They are very much not for the good and will destroy Scotland's regiments by moulding them into a single super regiment whic
h will lead to severe recruitment problems, a loss of local connections to those regiments and a loss to Scotland of an impor
tant part of her heritage and, most importantly, her future - the regiments are the envy of armies around the world."The prop
osals to either merge or amalgamate the six regiments into a super regiment sparked a political outcry, with Labour backbench
ers
```