Build a Natural Language Generation (NLG) System using PyTorch

Prateek Joshi — Published On August 4, 2020 and Last Modified On August 4th, 2020

Advanced Algorithm Deep Learning NLP Text Unstructured Data Unsupervised

Overview

- Introduction to Natural Language Generation (NLG) and related things-
  - Data Preparation
  - Training Neural Language Models
- Build a Natural Language Generation System using PyTorch

Introduction

In the last few years, Natural language processing (NLP) has seen quite a significant growth thanks to advancements in deep learning algorithms and the availability of sufficient computational power. However, feed-forward neural networks are not considered optimal for modeling a language or text. This is because the feed-forward network does not take into consideration the word order in the text.

Hence, to capture the sequential information present in the text, recurrent neural networks are used in NLP. In this article, we will see how we can use a recurrent neural network (LSTM), using PyTorch for Natural Language Generation.

If you need a quick refresher on PyTorch then you can go through the article below:

- A Beginner-Friendly Guide to PyTorch and How it Works from Scratch

And if you are new to NLP and wish to learn it from scratch, then check out our course:

- Natural Language Processing (NLP) Using Python

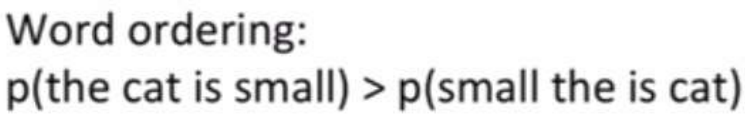
Build a Natural Language Generation (NLG) System using PyTorch

Table of Contents

A Brief Overview of Natural Language Generation

Natural Language Generation (NLG) is a subfield of Natural Language Processing (NLP) that is concerned with the automatic generation of human-readable text by a computer. NLG is used across a wide range of NLP tasks such as Machine Translation, Speech-to-text, chatbots, text auto-correct, or text auto-completion.

We can model NLG with the help of Language Modeling. Let me explain the concept of language models – A language model learns to predict the probability of a sequence of words. For example, consider the sentences below:



We can see that the first sentence, "the cat is small", is more probable than the second sentence, "small the is cat", because we know that the sequence of the words in the second sentence is not correct. This is the fundamental concept behind language modeling. A language model should be able to distinguish between a more probable and a less probable sequence of words (or tokens).

Types of Language Models

The following are the two types of Language Models:

1. **Statistical Language Models:** These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM), and certain linguistic rules to learn the probability distribution of words.
2. **Neural Language Models:** These models have surpassed the statistical language models in their effectiveness. They use different kinds of Neural Networks to model language.

In this article, we will focus on RNN/LSTM based neural language models. If you want a quick refresher on RNN or LSTM then please check out these articles:

- Introduction to Recurrent Neural Networks
- Introduction to LSTM

Build a Natural Language Generation (NLG) System using PyTorch

Text Generation using Neural Language Modeling

Text Generation using Statistical Language Models

First of all let's see how we can generate text with the help of a statistical model, like an N-Gram model. To understand how an N-Gram language model works then do check out the first half of the below article:

- A Comprehensive Guide to Build your own Language Model in Python

Suppose we have to generate the next word for the below sentence:

"There she built a ? "

Let's say that our N-Gram model considers a context of 3 previous only to predict the next word. So, the model will try to maximize the probability P(w | "she built a"), where 'w' represents each and every word in the text dataset. Whichever word that maximizes that probability will be generated as the next word for the sentence "There she built a…".

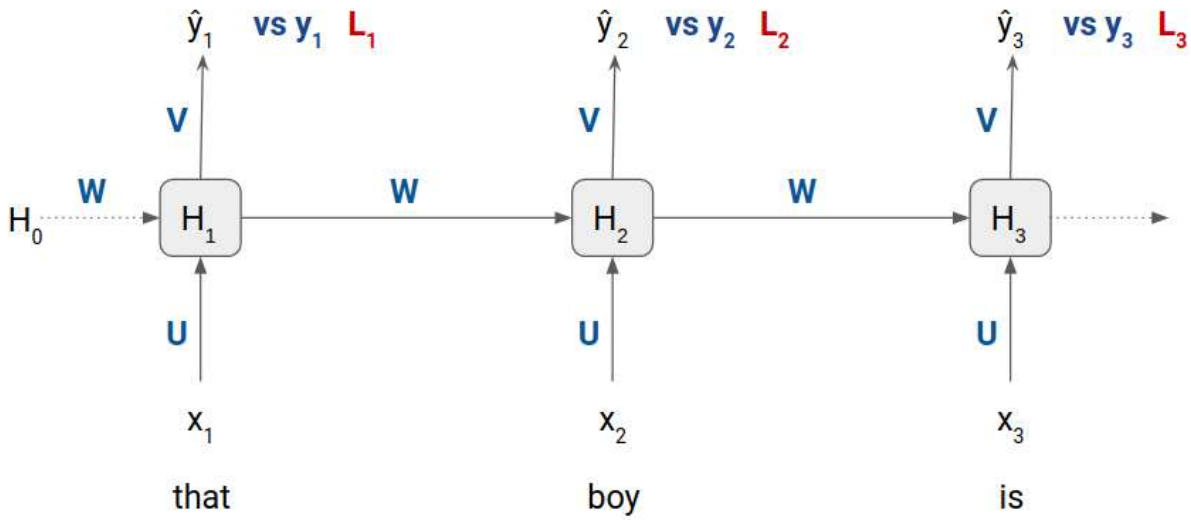
However, there are certain drawbacks of using such statistical models that use the immediate previous words as context to predict the next word. Let me give you some extra context.

"Alice went to the beach. There she built a sandcastle "

Now we have some more information about what's going on. The term "sandcastle" is very likely as the next word because it has a strong dependency on the term "beach" because people build sandcastles on beaches mostly right. So, the point is that "sandcastle" does not depend on the immediate context ("she built a") as much as it depends on "beach".

Text Generation using Neural Language Models

To capture such unbounded dependencies among the tokens of a sequence we can use an RNN/LSTM based language model. The following is a minimalistic representation of the language model that we will use for NLG:



- x1, x2, and x3 are the inputs word embeddings at timestep 1, timestep 2, and timestep 3 respectively
- ŷ1, ŷ2, and ŷ3 are the probability distribution of all the distinct tokens in the training dataset

- U, V, and W are the weight matrices
- and H0, H1, H2, and H3 are the hidden states

We will cover the working of this neural language model in the next section.

Understanding the Functioning of Neural Language Models

We will try to understand the functioning of a neural language model in three phases:

- Data Preparation
- Model Training
- Text Generation

1. Data Preparation

Let's assume that we will use the sentences below as our training data.

['alright that is perfect',
'that sounds great',
'what is the price difference']

The first sentence has 4 tokens, the second has 3 and the third has 5 tokens. So, all these sentences have varying lengths in terms of tokens. An LSTM model accepts sequences of the same length only as inputs. Therefore, we have to make the sequences in the training data have the same length.

There are multiple techniques to make sequences of equal length.

One technique is padding. We can pad the sequences with padding tokens wherever required. However, if we use this technique then we will have to deal with the padding tokens during loss calculation and text generation.

So, we will use another technique that involves splitting a sequence into multiple sequences of equal length without using any padding token. This technique also increases the size of the training data. Let me apply it to our training data.

Let's say we want our sequences to have exactly three tokens. Then the first sequence will be split into the following sequences:

['alright that is',
'that is perfect']

The second sequence is of length three only so it will not be split. However, the third sequence of the training data has five tokens and it will be broken down into multiple sequences of tokens:

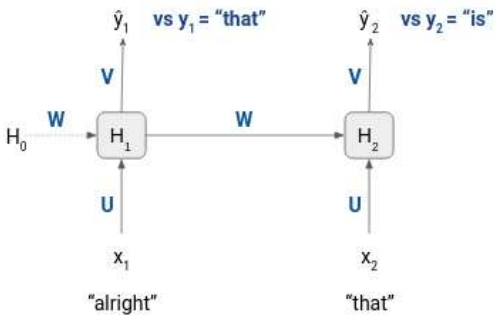
['what is the',
'is the price',
'the price difference']

Now the new dataset will look something like this:

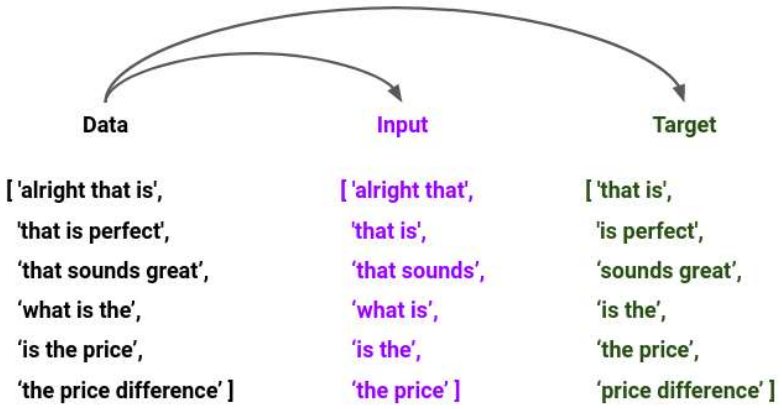
['alright that is',
'that is perfect',
'that sounds great',
'what is the',
'is the price',
'the price difference']

2. Model Training

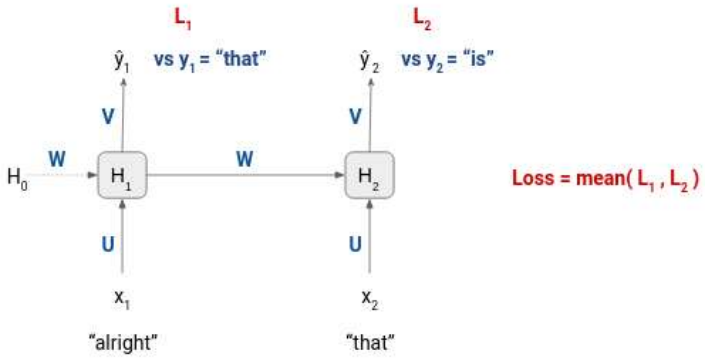
Since we want to solve the next word generation problem, the target should be the next word to the input word. For example, consider the first text sequence “alright that is”.



As you can see, with respect to the first sequence of our training data, the inputs to the model are “alright” and that”, and the corresponding target tokens are “that” and “is”. Hence, before starting the training process, we will have to split all the sequences in the dataset to inputs and targets as shown below:



So, these pairs of sequences under Input and Target are the training examples that will be passed to the model, and the loss for a training example will be the mean of losses at each timestep.



Let’s see how this model can then be used for text generation.

3. Text Generation

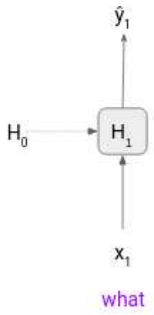
Once our language model is trained, we can then use it for NLG. The idea is to pass a text string as input along with a number of tokens you the model to generate after the input text string. For example, if the user passes “what is” as the input text and specifies that the model should generate 2 tokens, then the model might generate “what is going on” or “what is your name” or

Let me show you how it happens with the help of some illustrations:

input text = "what is"

n = 2

Step 1 – The first token ("what") of the input text is passed to the trained LSTM model. It generates an output ŷ1 which we will ignore because we already know the second token ("is"). The model also generates the hidden state H1 that will be passed to the next timestep.



Step 2 – Then the second token ("is") is passed to the model at timestep 2 along with H1. The output at this timestep is a probability distribution in which the token "going" has the maximum value. So, we will consider it as the first generated or predicted token by our model. Now we have one more token left to generate.

Step 3 – In order to generate the next token we need to pass an input token to the model at timestep 3. However, we have run out of the input tokens, "is" was the last token that generated "going". So, what do we pass next as input? In such a case we will pass the previously generated token as the input token.

The final output of the model would be "what is going on". That is the text generation strategy that we will use to perform NLG. Next, we will train our own language model on a dataset of movie plot summaries.

Natural Language Generation using PyTorch

Now that we know how a neural language model functions and what kind of data preprocessing it requires, let's train an LSTM language model to perform Natural Language Generation using PyTorch. I have implemented the entire code on Google Colab, so I suggest you should use it too.

Let's quickly import the necessary libraries.

```
import re
import pickle
import random

import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
```

nlg_import_lib.py hosted with ❤ by GitHub view raw

Build a Natural Language Generation (NLG) System using PyTorch

We will work with a sample of the CMU Movie Summary Corpus. You can download the pickle file of the sample data from this link.

```
# read pickle file
pickle_in = open("plots_text.pickle","rb")
movie_plots = pickle.load(pickle_in)

# count of movie plot summaries
len(movie_plots)
```

nlg_get_data.py hosted with ❤ by GitHub view raw

You can use the code below to print five summaries, sampled randomly.

```
# sample random summaries
random.sample(movie_plots, 5)
```

2. Data Preparation

First of all, we will clean our text a bit. We will keep only the alphabets and the apostrophe punctuation mark and remove the rest of the other elements from the text.

```
# clean text
movie_plots = [re.sub("[^a-z' ]", "", i) for i in movie_plots]
```

It is not mandatory to perform this step. It is just that I want my model to focus only on the alphabet and not worry about punctuation marks or numbers or other symbols.

Next, we will define a function to prepare fixed-length sequences from our dataset. I have specified the length of the sequence as five. It is a hyperparameter, you can change it if you want.

```
# create sequences of length 5 tokens
def create_seq(text, seq_len = 5):

    sequences = []

    # if the number of tokens in 'text' is greater than 5
    if len(text.split()) > seq_len:
      for i in range(seq_len, len(text.split())):
        # select sequence of tokens
        seq = text.split()[i-seq_len:i+1]
        # add to the list
        sequences.append(" ".join(seq))

      return sequences

    # if the number of tokens in 'text' is less than or equal to 5
    else:

      return [text]
```

nlg_seq_prep_func.py hosted with ❤ by GitHub view raw

So, we will pass the movie plot summaries to this function and it will return a list of fixed-length sequences for each input.

```
seqs = [create_seq(i) for i in movie_plots]

# merge list-of-lists into a single list
seqs = sum(seqs, [])

# count of sequences
```

nlg_get_seqs.py hosted with ❤ by GitHub view raw

Output: 152644

Once we have the same length sequences ready, we can split them further into input and target sequences.

```
# create inputs and targets (x and y)
x = []
y = []

for s in seqs:
  x.append(" ".join(s.split()[:-1]))
  y.append(" ".join(s.split()[1:]))
```

nlg_ip_op.py hosted with ❤ by GitHub view raw

Now we have to convert these sequences (x and y) into integer sequences, but before that, we will have to map each distinct word in the dataset to an integer value. So, we will create a token to integer dictionary and an integer to the token dictionary as well.

```
# create integer-to-token mapping
int2token = {}
cnt = 0

for w in set(" ".join(movie_plots).split()):
  int2token[cnt] = w
  cnt+= 1

# create token-to-integer mapping
token2int = {t: i for i, t in int2token.items()}

token2int["the"], int2token[14271]
```

nlg_tok_int_mapping.py hosted with ❤ by GitHub view raw

Output: (14271, 'the')

```
# set vocabulary size
vocab_size = len(int2token)
vocab_size
```

Output: 16592

The size of the vocabulary is 16,592, i.e., there are over 16,000 distinct tokens in our dataset.

Once we have the token to integer mapping in place then we can convert the text sequences to integer sequences.

```
def get_integer_seq(seq):
  return [token2int[w] for w in seq.split()]

# convert text sequences to integer sequences
x_int = [get_integer_seq(i) for i in x]
y_int = [get_integer_seq(i) for i in y]

# convert lists to numpy arrays
x_int = np.array(x_int)
y_int = np.array(y_int)
```

nlg_text_to_int.py hosted with ❤ by GitHub view raw

Build a Natural Language Generation (NLG) System using PyTorch

We will pass batches of the input and target sequences to the model as it is better to train batch-wise rather than passing the entire data to the model at once. The following function will create batches from the input data.

```
def get_batches(arr_x, arr_y, batch_size):

    # iterate through the arrays
    prv = 0
    for n in range(batch_size, arr_x.shape[0], batch_size):
      x = arr_x[prv:n,:]
      y = arr_y[prv:n,:]
      prv = n
      yield x, y
```

nlg_get_batch.py hosted with ❤ by GitHub view raw

Now we will define the architecture of our language model.

```
class WordLSTM(nn.Module):

    def __init__(self, n_hidden=256, n_layers=4, drop_prob=0.3, lr=0.001):
        super().__init__()

        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

        self.emb_layer = nn.Embedding(vocab_size, 200)

        ## define the LSTM
        self.lstm = nn.LSTM(200, n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)

        ## define a dropout layer
        self.dropout = nn.Dropout(drop_prob)

        ## define the fully-connected layer
        self.fc = nn.Linear(n_hidden, vocab_size)

    def forward(self, x, hidden):
        ''' Forward pass through the network.
            These inputs are x, and the hidden/cell state `hidden`. '''

        ## pass input through embedding layer
        embedded = self.emb_layer(x)

        ## Get the outputs and the new hidden state from the lstm
        lstm_output, hidden = self.lstm(embedded, hidden)

        ## pass through a dropout layer
        out = self.dropout(lstm_output)

        #out = out.contiguous().view(-1, self.n_hidden)
        out = out.reshape(-1, self.n_hidden)

        ## put "out" through the fully-connected layer
        out = self.fc(out)

        # return the final output and the hidden state
        return out, hidden


    def init_hidden(self, batch_size):
        ''' initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x n_hidden,
```

```
51
52          # if GPU is available
53          if (torch.cuda.is_available()):
54            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda(),
55                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda())
56
57          # if GPU is not available
58          else:
59            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_(),
60                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_())
61
62          return hidden
```

nlg_model_arch.py hosted with ❤ by GitHub view raw

The input sequences will first pass through an embedding layer, then through an LSTM layer. The LSTM layer will give a set of outputs equal to the sequence length, and each of these outputs will be passed to a linear (dense) layer on which softmax will be applied.

```
1  # instantiate the model
2  net = WordLSTM()
3
4  # push the model to GPU (avoid it if you are not using the GPU)
5  net.cuda()
6
7  print(net)
```

nlg_instantiate_model.py hosted with ❤ by GitHub view raw

Output:

WordLSTM(
(emb_layer): Embedding(16592, 200)
(lstm): LSTM(200, 256, num_layers=4, batch_first=True, dropout=0.3)
(dropout): Dropout(p=0.3, inplace=False)
(fc): Linear(in_features=256, out_features=16592, bias=True)
)

Let's now define a function that will be used to train the model.

```
1  def train(net, epochs=10, batch_size=32, lr=0.001, clip=1, print_every=32):
2
3      # optimizer
4      opt = torch.optim.Adam(net.parameters(), lr=lr)
5
6      # loss
7      criterion = nn.CrossEntropyLoss()
8
9      # push model to GPU
10     net.cuda()
11
12     counter = 0
13
14     net.train()
15
16     for e in range(epochs):
17
18         # initialize hidden state
19         h = net.init_hidden(batch_size)
20
21         for x, y in get_batches(x_int, y_int, batch_size):
22             counter+= 1
```

```
            inputs, targets = torch.from_numpy(x), torch.from_numpy(y)

            # push tensors to GPU
            inputs, targets = inputs.cuda(), targets.cuda()

            # detach hidden states
            h = tuple([each.data for each in h])

            # zero accumulated gradients
            net.zero_grad()

            # get the output from the model
            output, h = net(inputs, h)

            # calculate the loss and perform backprop
            loss = criterion(output, targets.view(-1))

            # back-propagate error
            loss.backward()

            # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
            nn.utils.clip_grad_norm_(net.parameters(), clip)

            # update weigths
            opt.step()

            if counter % print_every == 0:

              print("Epoch: {}/{}...".format(e+1, epochs),
                    "Step: {}...".format(counter))
```

nlg_train.py hosted with ❤ by GitHub view raw

```
# train the model
train(net, batch_size = 32, epochs=20, print_every=256)
```

I have specified the batch size of 32 and will train the model for 20 epochs. The training might take a while.

4. Text Generation

Once the model is trained, we can use it for text generation. Please note that this model can generate one word at a time along with a hidden state. So, to generate the next word we will have to use this generated word and the hidden state.

```
# predict next token
def predict(net, tkn, h=None):

  # tensor inputs
  x = np.array([[token2int[tkn]]])
  inputs = torch.from_numpy(x)

  # push to GPU
  inputs = inputs.cuda()

  # detach hidden state from history
  h = tuple([each.data for each in h])

  # get the output of the model
  out, h = net(inputs, h)

  # get the token probabilities
  p = F.softmax(out, dim=1).data

```

```
    p = p.numpy()
    p = p.reshape(p.shape[1],)

    # get indices of top 3 values
    top_n_idx = p.argsort()[-3:][::-1]

    # randomly select one of the three indices
    sampled_token_index = top_n_idx[random.sample([0,1,2],1)[0]]

    # return the encoded value of the predicted char and the hidden state
    return int2token[sampled_token_index], h


# function to generate text
def sample(net, size, prime='it is'):

    # push to GPU
    net.cuda()

    net.eval()

    # batch size is 1
    h = net.init_hidden(1)

    toks = prime.split()

    # predict next token
    for t in prime.split():
      token, h = predict(net, t, h)

    toks.append(token)

    # predict subsequent tokens
    for i in range(size-1):
        token, h = predict(net, toks[-1], h)
        toks.append(token)

    return ' '.join(toks)
```

nlg_text_gen.py hosted with ❤ by GitHub view raw

The function *sample( )* takes in an input text string (“prime”) from the user and a number (“size”) that specifies the number of tokens to generate. *sample( )* uses the *predict( )* function to predict the next word given an input word and a hidden state. Given below are a few text sequences generated by the model.

```
sample(net, 15)
```

Output:

‘it is now responsible by the temple where they have the new gospels him and is held’

```
sample(net, 15, prime = "one of the")
```

Output:

‘one of the team are waiting by his rejection and throws him into his rejection of the sannokai’

```
sample(net, 15, prime = "as soon as")
```

Output:

‘as soon as he is sent not to be the normal warrior caused on his mouth he has’

```
sample(net, 15, prime = "they")
```

Build a Natural Language Generation (NLG) System using PyTorch

'they find that their way into the ship in his way to be thrown in the'

End Notes

Natural Language Generation is a rapidly maturing field and increasingly active field of research. The methods used for NLG have also come a long way from N-Gram models to RNN/LSTM models and now transformer-based models are the new state-of-the-art models in this field.

To summarize, in this tutorial, we covered a lot of things related to NLG such as dataset preparation, how a neural language model is trained, and finally Natural Language Generation process in PyTorch. I suggest you try to build a language model on a bigger dataset and see what kind of text it generates.

In case you are looking for a roadmap to becoming an expert in NLP read the following article-

- A Comprehensive Learning Path to Understand and Master NLP in 2020

Please feel free to comment below if you have any queries.

Language modeling natural language generation Natural language processing pytorch text generation

About the Author

Prateek Joshi

Data Scientist at Analytics Vidhya with multidisciplinary academic background. Experienced in machine learning, NLP, graphs & networks. Passionate about learning and applying data science to solve real world problems.

Our Top Authors

Previous Post

Selecting the Right Bounding Box Using Non-Max Suppression (with implementation)

Next Post

Playing with YOLO v1 on Google Colab

3 thoughts on "Build a Natural Language Generation (NLG) System using PyTorch"

Wei Chuang says:
August 09, 2020 at 5:54 pm

Where can i get the resource code about the tutorial?

Reply

Prateek Joshi says:
August 19, 2020 at 4:45 pm

Hey Wei, Entire code is given in the article. You can use that.

Reply

chinmay says:
November 02, 2020 at 4:52 pm

This is the entire code in case you want it to be located in a single place
https://colab.research.google.com/drive/1M1dASeH9zj1fuDqMJ5Z_YKcooBSDNryJ?usp=sharing

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name*

Email*

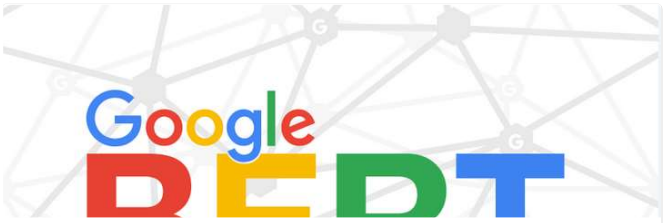
Website

☑ Notify me of follow-up comments by email.

☑ Notify me of new posts by email.

Submit

Top Resources



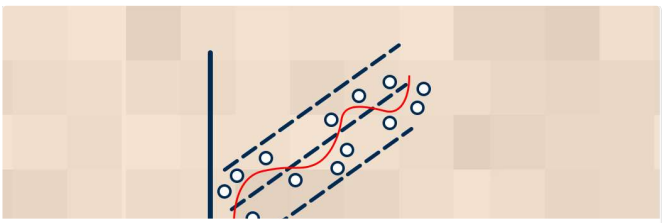
Building Language Models: A Step-by-Step BERT Implementation Guide

H1B Visa Data Analysis: Unveiling Patterns of H1B Visa Approval

Build a Natural Language Generation (NLG) System using PyTorch

Understand Random Forest Algorithms With Examples (Updated 2023)

Sruthi E R - JUN 17, 2021



Everything you need to Know about Linear Regression!

KAVITA MALI - OCT 04, 2021