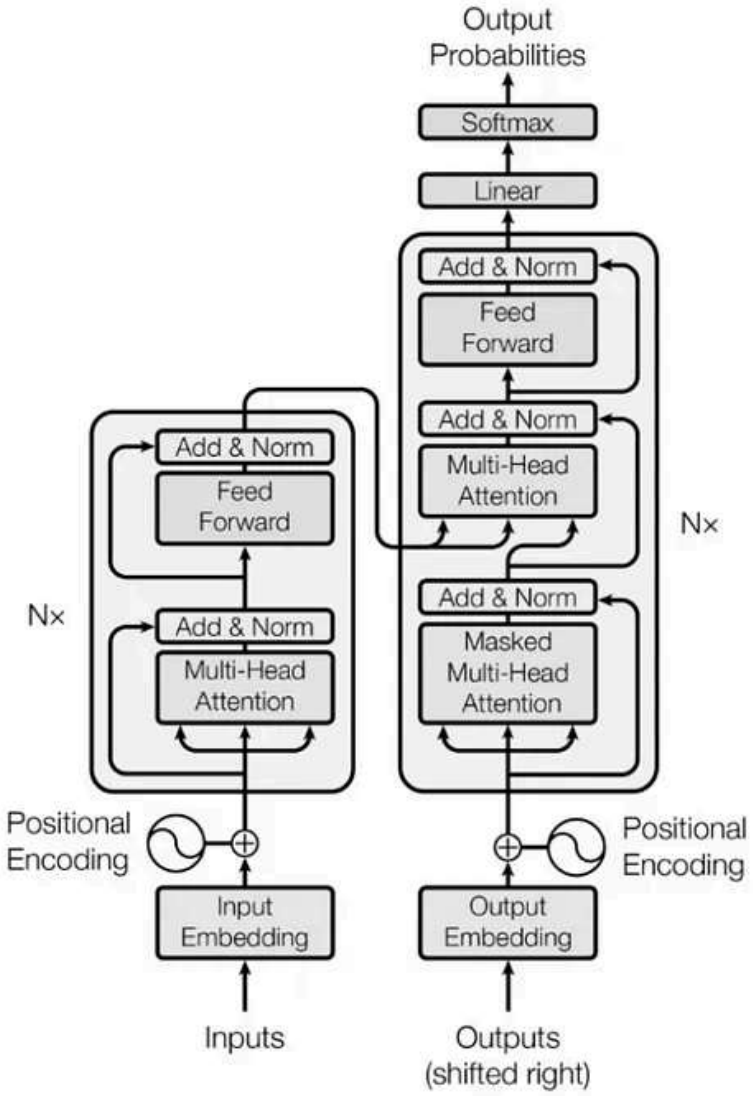
Training a Transformer Model from Scratch

Ebad Sayed · Follow

11 min read · Jun 11, 2024



https://cdn.analyticsvidhya.com/wp-content/uploads/2019/06/Screenshot-from-2019-06-17-19-53-10.png

In this article, we will see how to train the Transformer model that we built in the previous article on various tasks such as summarization, sentiment analysis, language translation, and more.

Previous Article :- Building a Transformer from Scratch

Dataset

I have chosen the translation task (English to Italian) to train my Transformer model on the **opus_books** dataset from Hugging Face.

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset

class TranslationDataset(Dataset):

    def __init__(self, ds, tokenizer_src, tokenizer_tgt, src_lang, tgt_lang, seq
        super().__init__()
        self.seq = seq

        self.ds = ds
        self.tokenizer_src = tokenizer_src
        self.tokenizer_tgt = tokenizer_tgt
        self.src_lang = src_lang
        self.tgt_lang = tgt_lang

        self.sos_token = torch.tensor([tokenizer_tgt.token_to_id("[SOS]")], dtyp
        self.eos_token = torch.tensor([tokenizer_tgt.token_to_id("[EOS]")], dtyp
        self.pad_token = torch.tensor([tokenizer_tgt.token_to_id("[PAD]")], dtyp

    def __len__(self):
        return len(self.ds)

    def __getitem__(self, idx):
        src_target_pair = self.ds[idx]
        src_text = src_target_pair['translation'][self.src_lang]
        tgt_text = src_target_pair['translation'][self.tgt_lang]

        # Transform the text into tokens
        enc_input_tokens = self.tokenizer_src.encode(src_text).ids
        dec_input_tokens = self.tokenizer_tgt.encode(tgt_text).ids

        # Add sos, eos and padding to each sentence
        enc_num_padding_tokens = self.seq - len(enc_input_tokens) - 2  # We will
        # We will only add <s>, and </s> only on the label
        dec_num_padding_tokens = self.seq - len(dec_input_tokens) - 1

        # Make sure the number of padding tokens is not negative. If it is, the
        if enc_num_padding_tokens < 0 or dec_num_padding_tokens < 0:
            raise ValueError("Sentence is too long")

        # Add <s> and </s> token
        encoder_input = torch.cat(
            [
                self.sos_token,
                torch.tensor(enc_input_tokens, dtype=torch.int64),
                self.eos_token,
                torch.tensor([self.pad_token] * enc_num_padding_tokens, dtype=to
```

```
            ],
            dim=0,
        )

        # Add only <s> token
        decoder_input = torch.cat(
            [
                self.sos_token,
                torch.tensor(dec_input_tokens, dtype=torch.int64),
                torch.tensor([self.pad_token] * dec_num_padding_tokens, dtype=to
            ],
            dim=0,
        )

        # Add only </s> token
        label = torch.cat(
            [
                torch.tensor(dec_input_tokens, dtype=torch.int64),
                self.eos_token,
                torch.tensor([self.pad_token] * dec_num_padding_tokens, dtype=to
            ],
            dim=0,
        )

        # Double check the size of the tensors to make sure they are all seq lon
        assert encoder_input.size(0) == self.seq
        assert decoder_input.size(0) == self.seq
        assert label.size(0) == self.seq

        return {
            "encoder_input": encoder_input,  # (seq)
            "decoder_input": decoder_input,  # (seq)
            "encoder_mask": (encoder_input != self.pad_token).unsqueeze(0).unsqu
            "decoder_mask": (decoder_input != self.pad_token).unsqueeze(0).int()
            "label": label,  # (seq)
            "src_text": src_text,
            "tgt_text": tgt_text,
        }

def causal_mask(size):
    mask = torch.triu(torch.ones((1, size, size)), diagonal=1).type(torch.int)
    return mask == 0
```

This dataset class prepares the data for training a Transformer model on a translation task by handling tokenization, padding, special tokens, and

masking, ensuring the model receives correctly formatted inputs. Let's take a look in more detail.

The **__init__** method initializes the dataset with the following parameters:
1. ds: The dataset containing source and target language pairs.
2. tokenizer_src: The tokenizer for the source language.
3. tokenizer_tgt: The tokenizer for the target language.
4. src_lang: The source language identifier (e.g., 'en' for English).
5. tgt_lang: The target language identifier (e.g., 'it' for Italian).
6. seq: The sequence length to which all sentences will be padded or truncated.
The method also initializes special tokens (**[SOS], [EOS], [PAD]**) for the target language.

The **__len__** method returns the length of the dataset.

The **__getitem__** method retrieves a data sample at a given index **idx** and processes it as follows:

1. **Retrieve Texts:**
   Extracts the source and target texts from the dataset.
2. **Tokenize Texts:**
   Tokenizes the source and target texts using their respective tokenizers.
3. **Padding Calculation:**
   Calculates the number of padding tokens needed to make the sequence length equal to self.seq.
4. **Sentence Length Validation:**
   Ensures the sentences are not too long to fit into the fixed sequence length after adding special tokens.

5. **Create Encoder Input:**
   Concatenates the [SOS] token, the encoded source tokens, the [EOS] token, and padding tokens.
6. **Create Decoder Input:**
   Concatenates the [SOS] token, the encoded target tokens, and padding tokens.
7. **Create Label:**
   Concatenates the encoded target tokens, the [EOS] token, and padding tokens.
8. **Masks:**
   Creates masks for the encoder and decoder inputs to distinguish actual tokens from padding tokens and applies a causal mask to the decoder input to prevent looking ahead.
9. **Return Dictionary:**
   Returns a dictionary containing the encoder input, decoder input, masks, label, and the original texts.

The **causal_mask** function generates an upper triangular matrix (excluding the diagonal) filled with ones, and then converts it into a boolean mask where zeros represent allowed positions (i.e., positions that can attend to previous positions in the sequence). This mask ensures that during training, each position in the decoder can only attend to previous positions, enforcing causality.

Configuration

```
from pathlib import Path

def get_config():
    return {
        "batch_size": 8,
        "num_epochs": 20,
        "lr": 10**-4,
        "seq": 350,
        "d_model": 512,
        "datasource": 'opus_books',
        "lang_src": "en",
        "lang_tgt": "it",
        "model_folder": "weights",
        "model_basename": "tmodel_",
        "preload": "latest",
        "tokenizer_file": "tokenizer_{0}.json",
        "experiment_name": "runs/tmodel"
    }

def get_weights_file_path(config, epoch: str):
    model_folder = f"{config['datasource']}_{config['model_folder']}"
    model_filename = f"{config['model_basename']}{epoch}.pt"
    return str(Path('.') / model_folder / model_filename)

# Find the latest weights file in the weights folder
def latest_weights_file_path(config):
    model_folder = f"{config['datasource']}_{config['model_folder']}"
    model_filename = f"{config['model_basename']}*"
    weights_files = list(Path(model_folder).glob(model_filename))
    if len(weights_files) == 0:
        return None
    weights_files.sort()
    return str(weights_files[-1])
```

This code provides utility functions for configuring and managing the training of a Transformer model.
The **Path** class from the pathlib module is used to handle file system paths in an object-oriented way.

The **get_config** function returns a dictionary containing the configuration parameters for the model training. Here's a breakdown of the parameters:

1. **batch_size:** The number of samples processed in one training iteration.
2. **num_epochs:** The number of times the entire training dataset will be passed through the model.
3. **lr:** The learning rate for the optimizer.
4. **seq:** The sequence length for input data.
5. **d_model:** The dimension of the model (e.g., the number of features in the input).
6. **datasource:** The name of the dataset source (e.g., 'opus_books').
7. **lang_src:** The source language code (e.g., 'en' for English).
8. **lang_tgt:** The target language code (e.g., 'it' for Italian).
9. **model_folder:** The folder where model weights will be saved.
10. **model_basename:** The base name for the saved model files.
11. **preload:** Specifies whether to load the latest weights file ('latest').
12. **tokenizer_file:** The template for the tokenizer file name.
13. **experiment_name:** The name for the experiment run (used for logging and tracking).

The **get_weights_file_path** function constructs the file path for the model weights based on the configuration and the epoch number.
config: The configuration dictionary returned by get_config.
epoch: A string representing the epoch number (e.g., '1', '2', etc.).

It constructs the file path by combining the current directory **(Path('.'))**, the model folder (constructed from the data source and model folder name), and

the model filename (constructed from the model base name and epoch number). The resulting file path is returned as a string.

The **latest_weights_file_path** function finds the latest weights file in the specified model folder.
config: The configuration dictionary returned by get_config.
It constructs the model folder and filename pattern (with a wildcard "*").
Then, it uses **Path.glob** to find all files matching the pattern in the model folder.
If no weights files are found, it returns **None**.
If weights files are found, it **sorts** them (which implicitly sorts by file name), and returns the path of the latest file (the last one in the sorted list) as a string.

Training

```
from model import build_transformer
from dataset import TranslationDataset, causal_mask
from config import get_config, get_weights_file_path, latest_weights_file_path

import torchtext.datasets as datasets
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from torch.optim.lr_scheduler import LambdaLR

import warnings
from tqdm import tqdm
import os
from pathlib import Path

# Huggingface datasets and tokenizers
from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer
from tokenizers.pre_tokenizers import Whitespace
```

```
import torchmetrics
from torch.utils.tensorboard import SummaryWriter
```

```
def get_all_sentences(ds, lang):
    for item in ds:
        yield item['translation'][lang]
```

The **get_all_sentences** function is a generator that iterates over a dataset (ds) and yields sentences in the specified language (lang). Each dataset item should have a 'translation' key with translations in multiple languages. The yield statement ensures memory efficiency by generating one sentence at a time instead of a complete list.

```
def get_or_build_tokenizer(config, ds, lang):
    tokenizer_path = Path(config['tokenizer_file'].format(lang))
    if not Path.exists(tokenizer_path):
        # Most code taken from: https://huggingface.co/docs/tokenizers/quicktour
        tokenizer = Tokenizer(WordLevel(unk_token="[UNK]"))
        tokenizer.pre_tokenizer = Whitespace()
        trainer = WordLevelTrainer(special_tokens=["[UNK]", "[PAD]", "[SOS]", "[
        tokenizer.train_from_iterator(get_all_sentences(ds, lang), trainer=train
        tokenizer.save(str(tokenizer_path))
    else:
        tokenizer = Tokenizer.from_file(str(tokenizer_path))
    return tokenizer
```

Tokenizer(models.WordLevel(unk_token="[UNK]")): Creates a new Tokenizer with a WordLevel model, specifying "[UNK]" as the token for unknown words.
tokenizer.pre_tokenizer = pre_tokenizers.Whitespace(): Sets the pre-

tokenizer to split text on whitespace.
WordLevelTrainer(special_tokens=[“[UNK]”, “[PAD]”, “[SOS]”, “[EOS]”], min_frequency=2): Creates a WordLevelTrainer with special tokens and a minimum frequency threshold of 2.
tokenizer.train_from_iterator(get_all_sentences(ds, lang), trainer=trainer): Trains the tokenizer using an iterator of sentences from the dataset for the specified language.
tokenizer.save(str(tokenizer_path)): Saves the trained tokenizer to the specified file path.
Tokenizer.from_file(str(tokenizer_path)): Loads the tokenizer from the specified file path if it already exists.
Returns the tokenizer, either newly built and trained or loaded from an existing file.

The **get_or_build_tokenizer** function checks if a tokenizer file exists for the specified language. If the file does not exist, it builds a new tokenizer using a WordLevel model, pre-tokenizes text by splitting on whitespace, and trains the tokenizer on the dataset with a WordLevelTrainer. The trained tokenizer is then saved to a file. If the tokenizer file already exists, it loads the tokenizer from the file. The function returns the tokenizer for use in subsequent text processing tasks.

```
def get_ds(config):
    # It only has the train split, so we divide it overselves
    ds_raw = load_dataset(f"{config['datasource']}", f"{config['lang_src']}-{con

    # Build tokenizers
    tokenizer_src = get_or_build_tokenizer(config, ds_raw, config['lang_src'])
    tokenizer_tgt = get_or_build_tokenizer(config, ds_raw, config['lang_tgt'])

    # Keep 90% for training, 10% for validation
    train_ds_size = int(0.9 * len(ds_raw))
    val_ds_size = len(ds_raw) - train_ds_size
```

```
    train_ds_raw, val_ds_raw = random_split(ds_raw, [train_ds_size, val_ds_size]

    train_ds = TranslationDataset(train_ds_raw, tokenizer_src, tokenizer_tgt, co
    val_ds = TranslationDataset(val_ds_raw, tokenizer_src, tokenizer_tgt, config

    # Find the maximum length of each sentence in the source and target sentence
    max_len_src = 0
    max_len_tgt = 0

    for item in ds_raw:
        src_ids = tokenizer_src.encode(item['translation'][config['lang_src']]).
        tgt_ids = tokenizer_tgt.encode(item['translation'][config['lang_tgt']]).
        max_len_src = max(max_len_src, len(src_ids))
        max_len_tgt = max(max_len_tgt, len(tgt_ids))

    print(f'Max length of source sentence: {max_len_src}')
    print(f'Max length of target sentence: {max_len_tgt}')

    train_dataloader = DataLoader(train_ds, batch_size=config['batch_size'], shu
    val_dataloader = DataLoader(val_ds, batch_size=1, shuffle=True)

    return train_dataloader, val_dataloader, tokenizer_src, tokenizer_tgt
```

Loading Dataset:
Load a dataset from the specified data source. Retrieve the name of the dataset source from the configuration and the language pair for the translation task (e.g., 'en-it' for English to Italian). **split='train'** specifies that only the training split of the dataset is loaded.

Build Tokenizer:
Call the **get_or_build_tokenizer** function to retrieve a tokenizer by passing the configuration dictionary and the raw dataset loaded previously and the source and target languages for the tokenizer.

Split the Dataset:
Randomly split the raw dataset into training and validation sets based on the calculated sizes.

Create Translation Datasets:
Call the TranslationDataset class from the **dataset.py** file and pass the raw training and validation datasets and the source and target language tokenizers. Also The source language, target language, and sequence length from the configuration.

Find Maximum Sentence Lengths:
Iterates over each item in the raw dataset. Then encode the source and target sentence and retrieves the token IDs. Update the maximum length if the current sentence is longer and output the maximum lengths of the source and target sentences.

Create Data Loaders:
Create iterable data loaders for the training and validation datasets. Retrieve the batch size for the training data loader from the configuration. Shuffle the datasets for better training performance.

Return:
Return the training and validation data loaders and the source and target language tokenizers.

```
def get_model(config, vocab_src_len, vocab_tgt_len):
    model = build_transformer(vocab_src_len, vocab_tgt_len, config["seq"], confi
    return model
```

This function is a simple wrapper around the **build_transformer** function, abstracting away the details of model creation. It takes in configuration

settings and vocabulary sizes as inputs and returns a transformer model ready for training and evaluation.

```
def train_model(config):
    # Define the device
    device = "cuda" if torch.cuda.is_available() else "mps" if torch.has_mps or
    print("Using device:", device)
    if (device == 'cuda'):
        print(f"Device name: {torch.cuda.get_device_name(device.index)}")
        print(f"Device memory: {torch.cuda.get_device_properties(device.index).t
    elif (device == 'mps'):
        print(f"Device name: <mps>")
    else:
        print("NOTE: If you have a GPU, consider using it for training.")
        print("      On a Windows machine with NVidia GPU, check this video: htt
        print("      On a Mac machine, run: pip3 install --pre torch torchvision
    device = torch.device(device)

    # Make sure the weights folder exists
    Path(f"{config['datasource']}_{config['model_folder']}").mkdir(parents=True,

    train_dataloader, val_dataloader, tokenizer_src, tokenizer_tgt = get_ds(conf
    model = get_model(config, tokenizer_src.get_vocab_size(), tokenizer_tgt.get_
    # Tensorboard
    writer = SummaryWriter(config['experiment_name'])

    optimizer = torch.optim.Adam(model.parameters(), lr=config['lr'], eps=1e-9)

    # If the user specified a model to preload before training, load it
    initial_epoch = 0
    global_step = 0
    preload = config['preload']
    model_filename = latest_weights_file_path(config) if preload == 'latest' els
    if model_filename:
        print(f'Preloading model {model_filename}')
        state = torch.load(model_filename)
        model.load_state_dict(state['model_state_dict'])
        initial_epoch = state['epoch'] + 1
        optimizer.load_state_dict(state['optimizer_state_dict'])
        global_step = state['global_step']
    else:
        print('No model to preload, starting from scratch')

    loss_fn = nn.CrossEntropyLoss(ignore_index=tokenizer_src.token_to_id('[PAD]'
```

```
    for epoch in range(initial_epoch, config['num_epochs']):
        torch.cuda.empty_cache()
        model.train()
        batch_iterator = tqdm(train_dataloader, desc=f"Processing Epoch {epoch:0
        for batch in batch_iterator:

            encoder_input = batch['encoder_input'].to(device) # (b, seq)
            decoder_input = batch['decoder_input'].to(device) # (B, seq)
            encoder_mask = batch['encoder_mask'].to(device) # (B, 1, 1, seq)
            decoder_mask = batch['decoder_mask'].to(device) # (B, 1, seq, seq)

            # Run the tensors through the encoder, decoder and the projection la
            encoder_output = model.encode(encoder_input, encoder_mask) # (B, seq
            decoder_output = model.decode(encoder_output, encoder_mask, decoder_
            proj_output = model.project(decoder_output) # (B, seq, vocab_size)

            # Compare the output with the label
            label = batch['label'].to(device) # (B, seq)

            # Compute the loss using a simple cross entropy
            loss = loss_fn(proj_output.view(-1, tokenizer_tgt.get_vocab_size()),
            batch_iterator.set_postfix({"loss": f"{loss.item():6.3f}"})

            # Log the loss
            writer.add_scalar('train loss', loss.item(), global_step)
            writer.flush()

            # Backpropagate the loss
            loss.backward()

            # Update the weights
            optimizer.step()
            optimizer.zero_grad(set_to_none=True)

            global_step += 1

        # Run validation at the end of every epoch
        run_validation(model, val_dataloader, tokenizer_src, tokenizer_tgt, conf

        # Save the model at the end of every epoch
        model_filename = get_weights_file_path(config, f"{epoch:02d}")
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'global_step': global_step
        }, model_filename)
```

This function is the core training logic for the transformer model. It handles device selection, model initialization, optimizer setup, training loop, and logging of training progress using Tensorboard.

Check and Set Device
Determines whether to use “cuda” (GPU), “mps” (Multi-Process Service), or “cpu” based on availability. Prints device information if using “cuda”.

Create Model Weights Folder
Creates a folder to save the model weights if it doesn’t exist.

Get Dataset
Loads the dataset and splits it into training and validation sets. Builds tokenizers for the source and target languages.

Initialize Model
Creates a transformer model using get_model function. Moves the model to the specified device.

Initialize Tensorboard
Creates a SummaryWriter to log training progress for visualization.

Initialize Optimizer
Uses Adam optimizer for training with the specified learning rate.

Load Pretrained Model (Optional)
If specified in the config, loads a pretrained model and optimizer state.

Define Loss Function
Uses CrossEntropyLoss with label smoothing.

Training Loop

Iterates over epochs. Within each epoch, iterates over batches in the training dataloader. Performs forward pass through the encoder, decoder, and projection layer.

Computes loss and performs backpropagation. Logs loss using Tensorboard. Runs validation at the end of each epoch. Saves model weights at the end of each epoch.

```
if __name__ == '__main__':
    warnings.filterwarnings("ignore")
    config = get_config()
    train_model(config)
```

When the script is run, it first sets up the environment to ignore warnings. It then retrieves the configuration settings and uses them to train the transformer model. This pattern allows the script to be used as a standalone program for training the model, while also being importable as a module for reuse in other scripts or projects.

After this we have four files namely:

1. Model.py: Contains the code for defining the transformer architecture (build_transformer, Encoder, Decoder, etc.) as well as any other related classes or functions.

2. Dataset.py: Contains the code for creating and managing datasets (TranslationDataset, get_ds, etc.), including tokenization and data loading logic.

3. Config.py: Contains the configuration settings (get_config) for the model, such as batch size, learning rate, etc. This allows for easy management and modification of hyperparameters.
4. Train.py: Contains the code (train_model) for training the transformer model using the provided dataset and configuration settings. This file typically includes the main training loop and related functions.

Run the `Train.py` file to start the training of the model. The training of this model was done on Kaggle using an **NVIDIA Tesla P100–16GB GPU.** It took **5 hours and 11 minutes** for training over **20 epochs** and each epoch has **3638 batches** to train on.

Result after Training

```
--------------------------------------------------------------------------------
    SOURCE: Her departure for the country suited Oblonsky in every way: it was good for the children, expenses would be cut down, and he would be freer.
    TARGET: Sotto tutti i riguardi a Stepan Arkad'ic piaceva molto che la moglie partisse; era salutare per i ragazzi, le spese sarebbero state minori,
ed egli sarebbe stato più libero.
 PREDICTED: Per Stepan Arkad ' ic piaceva molto che tutti quelli che Stepan Arkad ' ic si sarebbero buoni consigli , era ormai ben disposto , e lui sare
bbe stato sempre più libero .
--------------------------------------------------------------------------------
    SOURCE: From their conversation Levin gathered that he was not against new methods either.
    TARGET: Dalla conversazione col vecchio, Levin capì ch'egli non era alieno dalle innovazioni.
 PREDICTED: Dalla conversazione col vecchio , Levin capì ch ' egli non era dalle innovazioni .
--------------------------------------------------------------------------------
    SOURCE: But why does he not come?
    TARGET: Ma come mai non viene?
 PREDICTED: Ma perché non viene ?
--------------------------------------------------------------------------------
    SOURCE: When they had gone about three versts, Veslovsky suddenly missed his cigars and pocket-book, and did not know whether he had lost them or le
ft them on his table.
    TARGET: Allontanatisi di tre verste, Veslovskij a un tratto s'accorse che non aveva i sigari e il portafoglio, e non sapeva se li avesse perduti o l
asciati sul tavolo.
 PREDICTED: di tre verste , Veslovskij a un tratto si accorse che non aveva il portafoglio e non li sapeva se li avesse perduti o lasciati , per la tavo
la .
--------------------------------------------------------------------------------
    SOURCE: If it were not for that, one had better give up everything and flee to the ends of the earth.'
    TARGET: E se non ci fosse questo... lascia via tutto! Fuggi in capo al mondo!
 PREDICTED: Se non ci fosse questo ... lascia tutto ! a tutto !
--------------------------------------------------------------------------------
    SOURCE: 'I again offer you my arm if you wish to go,' said her husband, touching her arm.
    TARGET: — Vi offro ancora una volta il braccio, se volete andare — disse Aleksej Aleksandrovic, toccandole il braccio.
 PREDICTED: — Vi offro ancora una volta il braccio , se volete andare — disse il braccio , toccandole il braccio .
--------------------------------------------------------------------------------
    SOURCE: "Come back for it, then; I am your banker for forty pounds."
    TARGET: — Tornate a prenderle, sono il vostro banchiere per quaranta sterline.
 PREDICTED: — Tornate a prenderle , sono il vostro banchiere per quaranta sterline .
--------------------------------------------------------------------------------
    SOURCE: Levin followed, trying not to lag behind, but it became harder and harder until at last the moment came when he felt he had no strength lef
t, and then Titus again stopped and began whetting his scythe.
    TARGET: Levin lo seguiva, sforzandosi di non restare indietro, ma gli era sempre più difficile: veniva il momento in cui sentiva di non avere più fo
rze, ma proprio in quel momento Tit si fermava e si metteva ad affilare.
 PREDICTED: Levin lo seguiva , sforzandosi di non restare indietro , ma gli era sempre più difficile : aveva sentito proprio il momento in cui era stato
, e proprio in quel momento Tit si fermava a Tit e si metteva la falce .
--------------------------------------------------------------------------------
    SOURCE: THE DAY WHEN KOZNYSHEV ARRIVED at Pokrovsk was one of Levin's most distressing days.
    TARGET: Il giorno in cui Sergej Ivanovic giunse a Pokrovskoe, Levin era in una delle sue giornate più tormentose.
 PREDICTED: Il giorno in cui Sergej Ivanovic giunse a Pokrovskoe , Levin era un ' altra persona di quella sua età .
--------------------------------------------------------------------------------
    SOURCE: 'You are upset. I quite understand.
    TARGET: — Tu sei agitato, capisco.
 PREDICTED: — Tu sei agitato , capisco .
--------------------------------------------------------------------------------
```

Image by Author

You can get the code files form my GitHub.

Future Work

You can try to train this transformer model on other tasks like summarization, sentiment analysis, etc., and also experiment with the hyperparameters from the **config.py** file to observe the differences in the output. Additionally, you can evaluate the model on benchmarks like **BLEU score, ROUGE, etc** to assess its performance more formally.

Transformer Model | Ai Model Training | Large Language Models

Written by Ebad Sayed

142 Followers · 9 Following

I am currently a final year undergraduate at IIT Dhanbad, looking to help out aspiring AI/ML enthusiasts with easy AI/ML guides.

Follow

No responses yet

What are your thoughts?

Respond

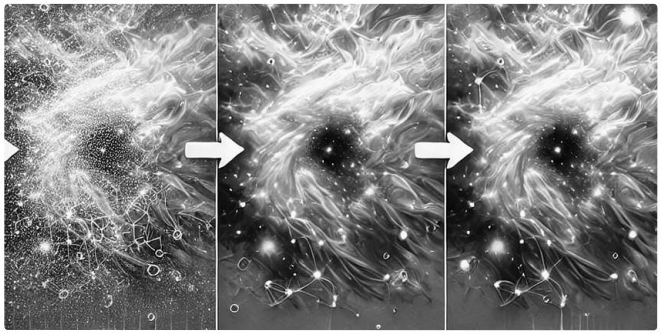
More from Ebad Sayed

Ebad Sayed

Scikit-LLM: Scikit-Learn Meets Large Language Models

As a beginner in Python and ML, I frequently relied on scikit-learn for almost all of my...

Sep 9 418 5

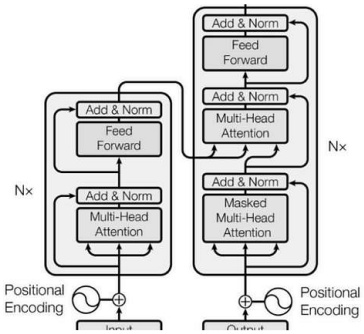


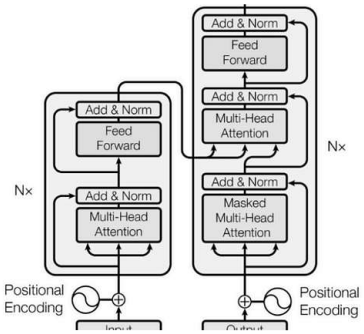
Ebad Sayed

Mastering Diffusion Probabilistic Models from Scratch

Paper: Denoising Diffusion Probabilistic Models

Dec 4 180 5





Ebad Sayed

Building a Transformer from Scratch: A Step-by-Step Guide

Introduction

Jun 9 42

Ebad Sayed

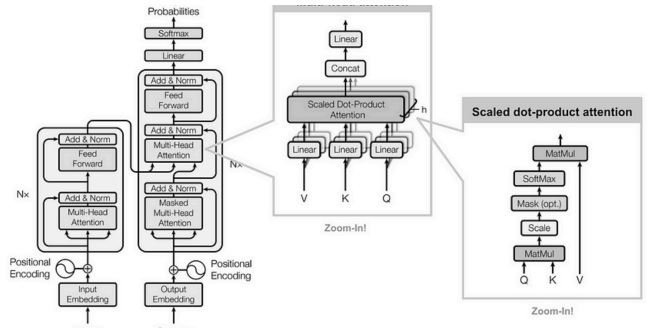
Mastering Transformer: Detailed Insights into Each Block

Before Transformers, Recurrent Neural Networks (RNNs) were used to handle...

Jun 8 43

See all from Ebad Sayed

Recommended from Medium



LM Po

Self-Attention and Transformer Network Architecture

The introduction of Transformer models in 2017 marked a significant turning point in th...

Oct 17 · 34 · 1

Amit Yadav

Few-Shot Learning vs. Semi-Supervised Learning

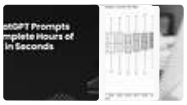
“In theory, there’s no difference between theory and practice. In practice, there is.”—...

Dec 4 · 1

Lists

Natural Language Processing

1857 stories · 1485 saves



ChatGPT prompts

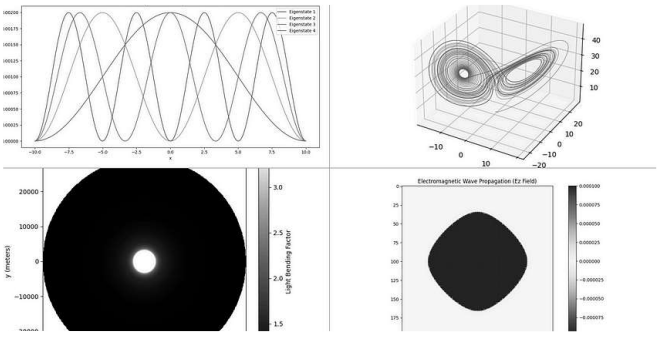
50 stories · 2360 saves

AI Regulation

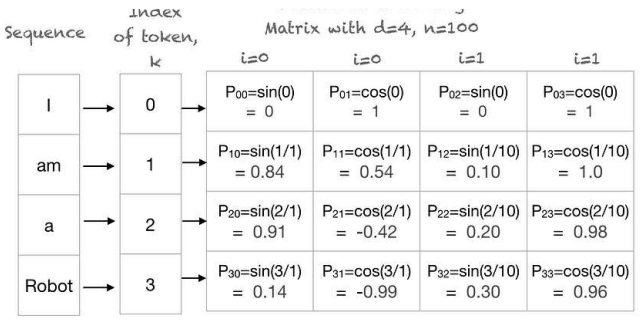
6 stories · 649 saves

Generative AI Recommended Reading

52 stories · 1552 saves



In Tech Spectrum by Aarafat Islam



Afrid Mondal

Simulating Complex Physics Equations with Python

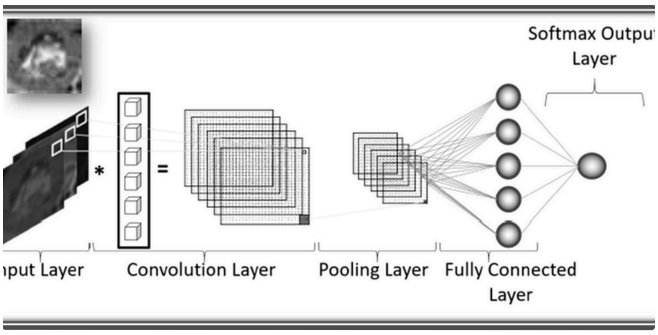
Simulating Black Holes, Schrödinger's Equation, Lorenz System & Electromagnetic...

Oct 18 150

Decoding Positional Encoding: The Secret Sauce Behind Transformer...

Positional encoding is a critical component of the Transformer architecture. Unlike recurre...

Nov 27

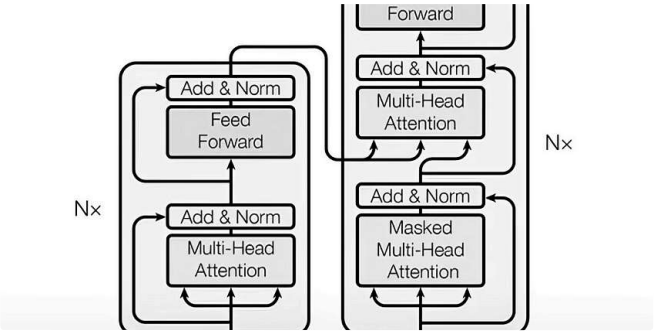


In Python in Plain English by Jyoti Dabass, Ph.D.

Friendly Introduction to Deep Learning Architectures (CNN, RN...

This blog aims to provide a friendly introduction to deep learning architectures...

Apr 1 1.3K 13



BavalpreetSinghh

Transformer from scratch using Pytorch

In today's blog we will go through the understanding of transformers architecture....

Jun 15 409 3

See more recommendations