Chatbot Implementation in Customer Service Industry
Natural Language Processing

Project Report

I. Introduction

Chatbots are computer programs designed to simulate conversation with human users, especially over the Internet. They can be integrated into messaging apps, websites, mobile apps, and other platforms.
Chatbots are used for a variety of purposes, including customer service, information gathering, and entertainment. Some chatbots are programmed to answer questions and provide information to users, while others are designed to facilitate transactions or help users with tasks such as booking a hotel room or ordering food.

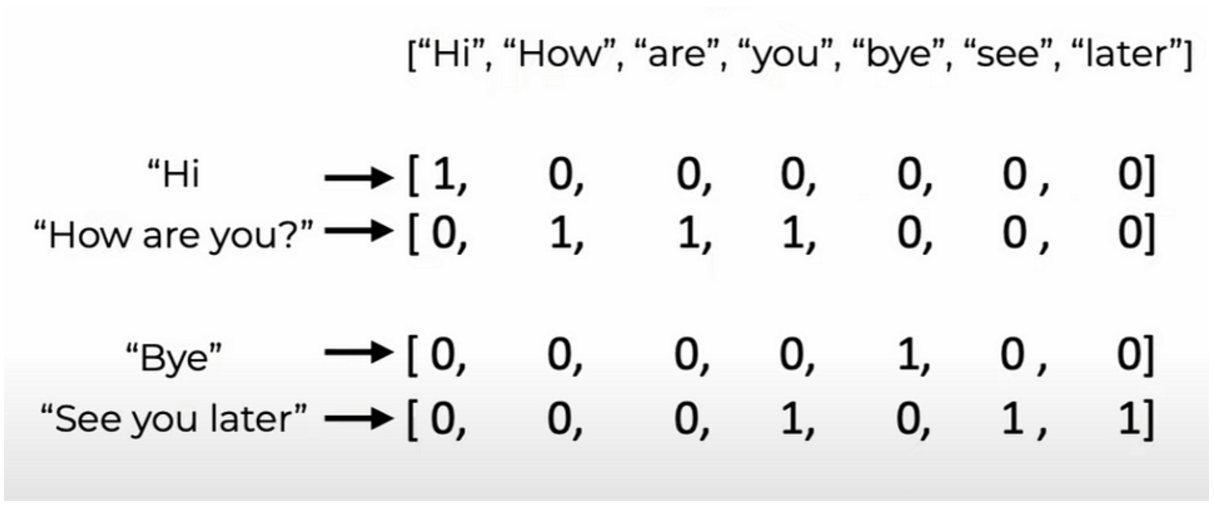
NLP has a wide range of applications, including language translation, text summarization, sentiment analysis, and chatbot development. An example of NLP that you use every day is Google Search. Google takes the keywords in your request and searches the internet for matching resources. Other examples include Google Translate, Grammarly, and the most advanced application, ChatGPT.

NLP algorithms and models are trained on large datasets of human language data and use techniques such as machine learning and deep learning to recognize patterns and make informed decisions. This is exactly how chatbots function. They're provided with a set of customer requests and corresponding responses. They're then trained on this data and are able to recognize similar sentences to the ones they're trained on.

The bag of words model is a way of representing text data as numbers. It involves converting a collection of text documents into numerical feature vectors that can be input into an ML model.
To create a bag of words model, the text is first preprocessed to remove unwanted characters and words, such as punctuation and stop words. Then, an array of words is created by selecting a set of words that appear most frequently in the text. In our case, the array will contain all words from our data, since it's a relatively small amount of data.
Each sentence is then represented as a numerical vector, with each element of the vector corresponding to a word in the vocabulary. The value of each element in the vector determines whether the word is frequently used or not.

Here's an example:
Let's say our dataset of customer input contains the following inputs:
"Hi", "How are you?", "Bye", and "See you later".

The set of all words would contain “Hi”, “How” “are”, “you”, “bye”, “see”, and “later”. Note that we don’t repeat words. The word vector would look like this: [“Hi”, “How”, “are”, “you”, “bye”, “see”, “later”].
If we use the bag of words model here, we would create four vectors with the same dimensions, since we have four inputs. Here’s how we would do it:



["Hi", "How", "are", "you", "bye", "see", "later"]

“Hi → [1, 0, 0, 0, 0, 0, 0]
“How are you?” → [0, 1, 1, 1, 0, 0, 0]
“Bye” → [0, 0, 0, 0, 1, 0, 0]
“See you later” → [0, 0, 0, 1, 0, 1, 1]

We first make four vectors with the same dimensions as the “all words” vector. All vectors contain zeros at first. Then, for each vector, we change every element corresponding to a word that’s present in the “all words” vector to “1”. This creates a unique vector to represent each sentence.
Now that we have all of our sentences as vectors, we’re ready to train them with the neural network.

II. Dataset

The dataset is a custom dataset made especially at Yale University. The sample conversations are based upon the personal student assistant chatbot, for which the student is from Yale University.

The text data in the question-and-answer columns of the pandas DataFrame are cleaned up in the data preparation stage described above. It specifically uses the pandas Series.apply() method to apply the preprocessing procedure to each member of these columns. Preprocessing () performs the following processes on a phrase after receiving it as input: Using the. lower() function, it lowercases every character. Using the regular phrase '[,.]', it eliminates all commas and periods. This standardised the text data's format and gets rid of any extra punctuation that can get in the way of model training or assessment. The matching columns in the original DataFrame df are then given new assignments to the updated question and response columns.

```
In [4]: # data preprocessing

        def preprocessing(sentence):
            return re.sub('[,.]', '', sentence.lower())
        df['question'] = df['question'].apply(preprocessing)
        df['answer'] = df['answer'].apply(preprocessing)
```

```
In [5]: df.head()

Out[5]:
                              question                               answer
        0          hi how are you doing?          i'm fine how about yourself?
        1   i'm fine how about yourself?      i'm pretty good thanks for asking
        2 i'm pretty good thanks for asking    no problem so how have you been?
        3 no problem so how have you been?       i've been great what about you?
        4 i've been great what about you?  i've been good i'm in school right now
```

III. Transfer Learning

a. Implementation

```
In [6]: class tokenGenerator:
            def __init__(self, conversation):
                self.conversation = conversation
                self.tokens_list = []
                self.word_freq = {}
                self.unique_words = set()
                self.W2I = {}
                self.I2W = {}

            def counter(self):
                self.tokens_list = [word for sentence in self.conversation for word in sent
                self.word_freq = Counter(self.tokens_list)
                self.unique_words = set(self.tokens_list)
                self.W2I = {word: i for i, word in enumerate(self.unique_words)}
                self.I2W = {i: word for word, i in self.W2I.items()}
```

```
In [7]: quest_tokenizer = tokenGenerator(df['question'])
        quest_tokenizer.counter()
        ans_tokenizer = tokenGenerator(df['answer'])
        ans_tokenizer.counter()
```

```
In [8]: # Printing out results
        print(f"Number of words in questions: {len(quest_tokenizer.tokens_list)}")
        print(f"Number of unique words in questions: {len(quest_tokenizer.unique_words)}")
        print(f"Most common words in questions: {quest_tokenizer.word_freq.most_common(10)}
        print(f"Index of 'the' in questions: {quest_tokenizer.W2I.get('the')}")
        print(f"Word at index 100 in questions: {quest_tokenizer.I2W.get(100)}")

        print(f"Number of words in answers: {len(ans_tokenizer.tokens_list)}")
        print(f"Number of unique words in answers: {len(ans_tokenizer.unique_words)}")
        print(f"Most common words in answers: {ans_tokenizer.word_freq.most_common(10)}")
        print(f"Index of 'the' in answers: {ans_tokenizer.W2I.get('the')}")
        print(f"Word at index 100 in answers: {ans_tokenizer.I2W.get(100)}")
```

```
Number of words in questions: 1193
Number of unique words in questions: 356
Most common words in questions: [('the', 54), ('yale', 40), ('in', 35), ('you', 3
4), ('is', 30), ('a', 28), ('for', 19), ('what', 19), ('to', 19), ('it', 19)]
Index of 'the' in questions: 29
Word at index 100 in questions: attend?
Number of words in answers: 1021
Number of unique words in answers: 343
Most common words in answers: [('the', 39), ('you', 33), ('in', 26), ('yale', 25),
('is', 24), ('a', 23), ('it', 22), ('and', 22), ('yes', 17), ('program', 17)]
Index of 'the' in answers: 26
Word at index 100 in answers: at
```

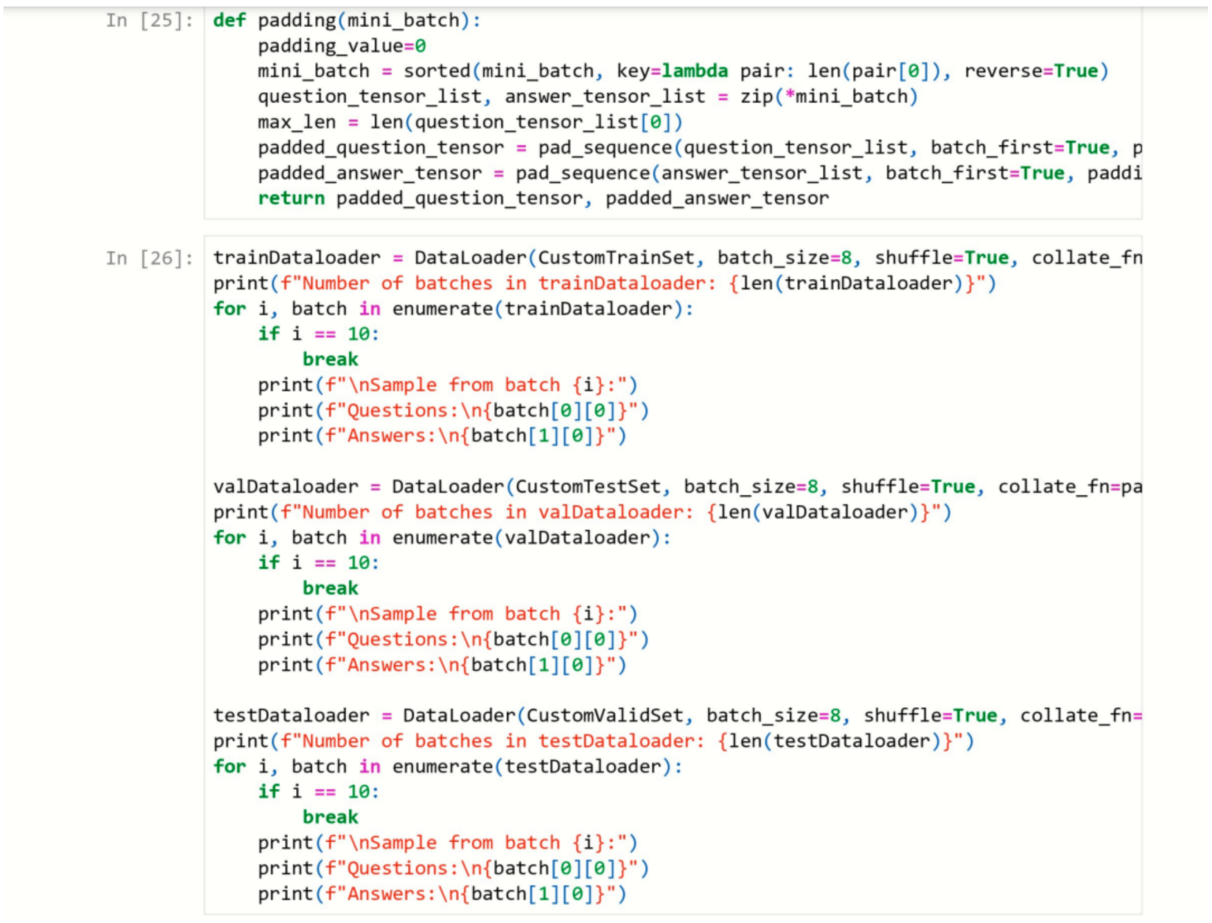
```
In [9]: # creating custom dataset
```

```
In [19]: class OwnDataset(Dataset):
             def __init__(self, conversation, quest_tokenizer, ans_tokenizer):
                 self.conversation = conversation
                 self.end_token = 1

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

             def __getitem__(self, index):
                 quest_and_answer = self.conversation.iloc[index]
                 quest_indexes = [quest_tokenizer.W2I[token] for token in quest_and_answer["
                 quest_indexes.append(self.end_token)
                 ans_indexes = [ans_tokenizer.W2I[token] for token in quest_and_answer["answ
                 ans_indexes.append(self.end_token)
                 return torch.tensor(quest_indexes), torch.tensor(ans_indexes)
```

```
In [20]: trainSet, testSet = train_test_split(df, test_size=0.20)

         validSet, testSet = train_test_split(testSet, test_size=0.50)
```

```
In [21]: CustomTrainSet = OwnDataset(trainSet, quest_tokenizer, ans_tokenizer)
         CustomTestSet = OwnDataset(testSet, quest_tokenizer, ans_tokenizer)
         CustomValidSet = OwnDataset(validSet, quest_tokenizer, ans_tokenizer)
```

```
In [22]: print(f"Number of samples in training set: {len(CustomTrainSet)}")
         print(f"Number of samples in test set: {len(CustomTestSet)}")
         print(f"Number of samples in validation set: {len(CustomValidSet)}")

         Number of samples in training set: 104
         Number of samples in test set: 13
         Number of samples in validation set: 13
```



```
In [25]: def padding(mini_batch):
             padding_value=0
             mini_batch = sorted(mini_batch, key=lambda pair: len(pair[0]), reverse=True)
             question_tensor_list, answer_tensor_list = zip(*mini_batch)
             max_len = len(question_tensor_list[0])
             padded_question_tensor = pad_sequence(question_tensor_list, batch_first=True, p
             padded_answer_tensor = pad_sequence(answer_tensor_list, batch_first=True, paddi
             return padded_question_tensor, padded_answer_tensor

In [26]: trainDataloader = DataLoader(CustomTrainSet, batch_size=8, shuffle=True, collate_fn
         print(f"Number of batches in trainDataloader: {len(trainDataloader)}")
         for i, batch in enumerate(trainDataloader):
             if i == 10:
                 break
             print(f"\nSample from batch {i}:")
             print(f"Questions:\n{batch[0][0]}")
             print(f"Answers:\n{batch[1][0]}")

         valDataloader = DataLoader(CustomTestSet, batch_size=8, shuffle=True, collate_fn=pa
         print(f"Number of batches in valDataloader: {len(valDataloader)}")
         for i, batch in enumerate(valDataloader):
             if i == 10:
                 break
             print(f"\nSample from batch {i}:")
             print(f"Questions:\n{batch[0][0]}")
             print(f"Answers:\n{batch[1][0]}")

         testDataloader = DataLoader(CustomValidSet, batch_size=8, shuffle=True, collate_fn=
         print(f"Number of batches in testDataloader: {len(testDataloader)}")
         for i, batch in enumerate(testDataloader):
             if i == 10:
                 break
             print(f"\nSample from batch {i}:")
             print(f"Questions:\n{batch[0][0]}")
             print(f"Answers:\n{batch[1][0]}")
```

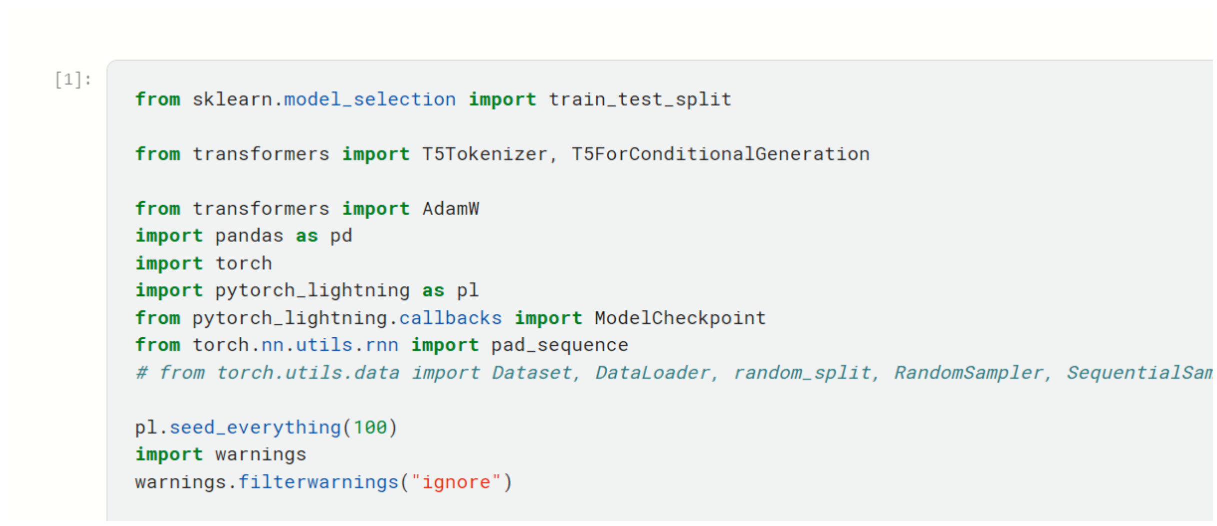
b. Fine Tuning

The process of fine-tuning involves freezing some layers of the pre-trained model, while allowing others to be trained. The frozen layers retain the knowledge learned during the pre-training stage, while the trainable layers are adjusted to the new task by updating their weights.
Whenever the task at hand and the task the pre-trained models was previously trained on are comparable, fine-tuning is especially helpful. In order to obtain favourable results on the new job, fine-tuning can reduce the amount of data necessary by utilising the knowledge acquired by the pre-trained model. Natural language processing (NLP) tasks including text categorization, sentiment assessment, and language production frequently use it.

Some important steps of Fine-tuning:

1. **Select a pre-trained model**:Choose a pre-trained model that is suitable for the task you want to perform. This could be a model that was trained on a similar dataset or a similar task.

2. **Prepare the data**: Prepare a dataset that is specific to the task you want to perform. This dataset should be split into training, validation, and test sets. The data should also be preprocessed to match the input format of the pre-trained model.
3. **Load the pre-trained model**: Load the pre-trained model into memory, and freeze some of its layers. The number of layers to freeze depends on the size of the new dataset and the similarity between the pre-trained model and the new task.
4. **Add new layers**: Add new layers to the pre-trained model that are specific to the new task. These layers are usually added to the end of the pre-trained model and are randomly initialised.
5. **Train the model**: Train the model on the new dataset using a suitable optimizer and loss function. The learning rate and number of epochs should be chosen based on the size of the dataset and the complexity of the model.
6. **Validate the model**: Validate the performance of the model on the validation set. This step is important to avoid overfitting and to select the best model.
7. **Test the model**: Test the final model on the test set to evaluate its performance on unseen data.



```
[1]:
from sklearn.model_selection import train_test_split

from transformers import T5Tokenizer, T5ForConditionalGeneration

from transformers import AdamW
import pandas as pd
import torch
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint
from torch.nn.utils.rnn import pad_sequence
# from torch.utils.data import Dataset, DataLoader, random_split, RandomSampler, SequentialSam

pl.seed_everything(100)
import warnings
warnings.filterwarnings("ignore")
```

Here **PyTorch-lightning** is used: PyTorch Lightning is a lightweight interface for PyTorch that simplifies the process of training deep learning models. It provides pre-built components and features for common tasks, making the code more modular and reusable. PyTorch Lightning also provides various features such as automatic checkpointing, distributed training, and multi-GPU training. It follows a strict design pattern and provides hooks and callbacks for customization. It is compatible with various hardware platforms such as CPUs and GPUs.

```
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
INPUT_MAX_LEN = 128 #input length
OUTPUT_MAX_LEN = 128 # output length
TRAIN_BATCH_SIZE = 8 # batch size of training
VAL_BATCH_SIZE = 2 # batch size for validation
EPOCHS = 5 # number of epoch
```

The task is to create a conversational model that can generate natural and engaging responses to a given input text. The model should be able to understand the context of the conversation and generate appropriate responses that are relevant to the topic and flow of the conversation. Additionally, the model should be able to handle open-ended conversations, where the topic can change dynamically, and maintain coherence throughout the conversation.

T5 tokenizer is the tokenizer used in the T5 model, which converts the raw text input into a sequence of tokens that can be processed by the T5 model. The T5 tokenizer is based on the Byte-Pair Encoding (BPE) algorithm, which is a popular technique for subword tokenization. It also includes a special "extra_ids" token to handle long inputs and outputs.
The T5 tokenizer is designed to handle a wide range of natural language processing tasks, and it can handle inputs and outputs of different lengths and types. Additionally, the T5 tokenizer can generate task-specific tokens to help the model understand the type of task it is performing. This makes the T5 tokenizer very flexible and powerful, and it has been used in many state-of-the-art natural language processing models.

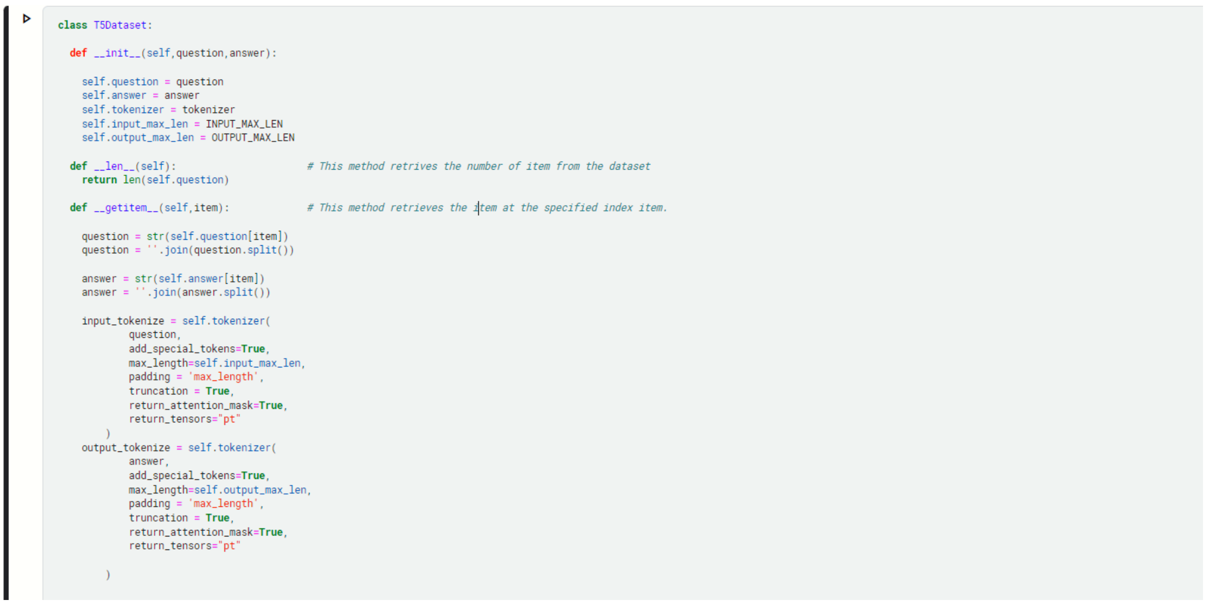
```
MODEL_NAME = "t5-base"
tokenizer = T5Tokenizer.from_pretrained(MODEL_NAME, model_max_length=512)
```

Example of how T5 Tokenizer actually work.

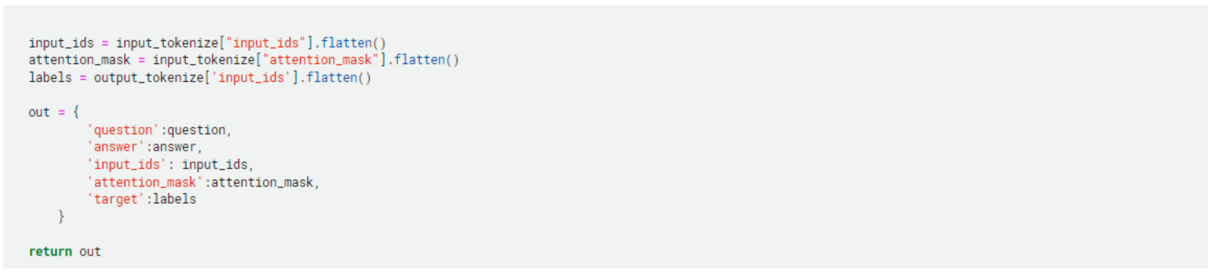
```
text = "Hello, how are you today?"    # assume the text that is to be tokenized

input_tokenize = tokenizer(
             text,
            add_special_tokens=True,        #Add Special tokens like [CLS] and [SEP]
            max_length=128,
            padding = 'max_length',         #for padding to max_length for equal sequence lengt
            truncation = True,              #truncate the text if it is greater than max_lengt
            return_attention_mask=True,     #will return attention mask
            return_tensors="pt"             #return tensor formate
        )
```

Data Loader

PyTorch Lightning is a popular library that provides a lightweight wrapper around PyTorch to help with training deep learning models. It also provides a DataModule class that makes it easy to load and preprocess data for training and inference.
To use a PyTorch Lightning DataModule with a T5-based model, you can define a T5DataModule class that inherits from pl.LightningDataModule. Here's an example implementation:



```
class T5Dataset:

  def __init__(self,question,answer):

    self.question = question
    self.answer = answer
    self.tokenizer = tokenizer
    self.input_max_len = INPUT_MAX_LEN
    self.output_max_len = OUTPUT_MAX_LEN

  def __len__(self):                      # This method retrives the number of item from the dataset
    return len(self.question)

  def __getitem__(self,item):             # This method retrieves the item at the specified index item.

    question = str(self.question[item])
    question = ''.join(question.split())

    answer = str(self.answer[item])
    answer = ''.join(answer.split())

    input_tokenize = self.tokenizer(
            question,
            add_special_tokens=True,
            max_length=self.input_max_len,
            padding = 'max_length',
            truncation = True,
            return_attention_mask=True,
            return_tensors="pt"
        )
    output_tokenize = self.tokenizer(
            answer,
            add_special_tokens=True,
            max_length=self.output_max_len,
            padding = 'max_length',
            truncation = True,
            return_attention_mask=True,
            return_tensors="pt"

        )
```



```
input_ids = input_tokenize["input_ids"].flatten()
attention_mask = input_tokenize["attention_mask"].flatten()
labels = output_tokenize['input_ids'].flatten()

out = {
        'question':question,
        'answer':answer,
        'input_ids': input_ids,
        'attention_mask':attention_mask,
        'target':labels
    }

return out
```

```python
class T5DataLoad(pl.LightningDataModule):

    def __init__(self,df_train,df_test):
        super().__init__()
        self.df_train = df_train
        self.df_test = df_test
        self.tokenizer = tokenizer
        self.input_max_len = INPUT_MAX_LEN
        self.out_max_len = OUTPUT_MAX_LEN

    def setup(self, stage=None):

        self.train_data = T5Dataset(
            question = self.df_train.question.values,
            answer = self.df_train.answer.values
        )

        self.valid_data = T5Dataset(
            question = self.df_test.question.values,
            answer = self.df_test.answer.values
        )
    def train_dataloader(self):
        return torch.utils.data.DataLoader(
         self.train_data,
         batch_size= TRAIN_BATCH_SIZE,
         shuffle=True,
         num_workers=2
        )
    def val_dataloader(self):
        return torch.utils.data.DataLoader(
        self.valid_data,
        batch_size= VAL_BATCH_SIZE,
        num_workers = 2
        )
```

Building T5 Model

To build a T5-based model with PyTorch Lightning, you first need to define a LightningModule class. This class represents the model architecture and contains all the necessary methods for training, validation, and inference.

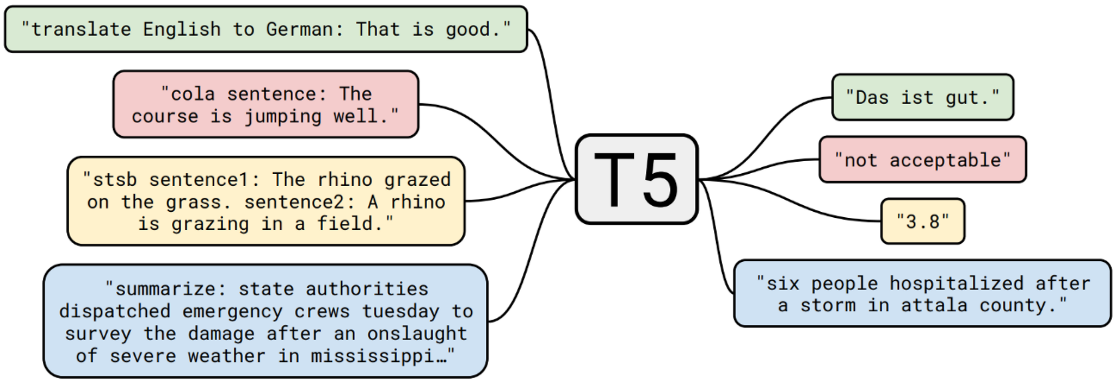
Within the init method of the LightningModule class, you can define the model architecture using the T5 model from the Transformers library. You can also define any other hyperparameters for the model, such as the learning rate, number of epochs, and batch size.

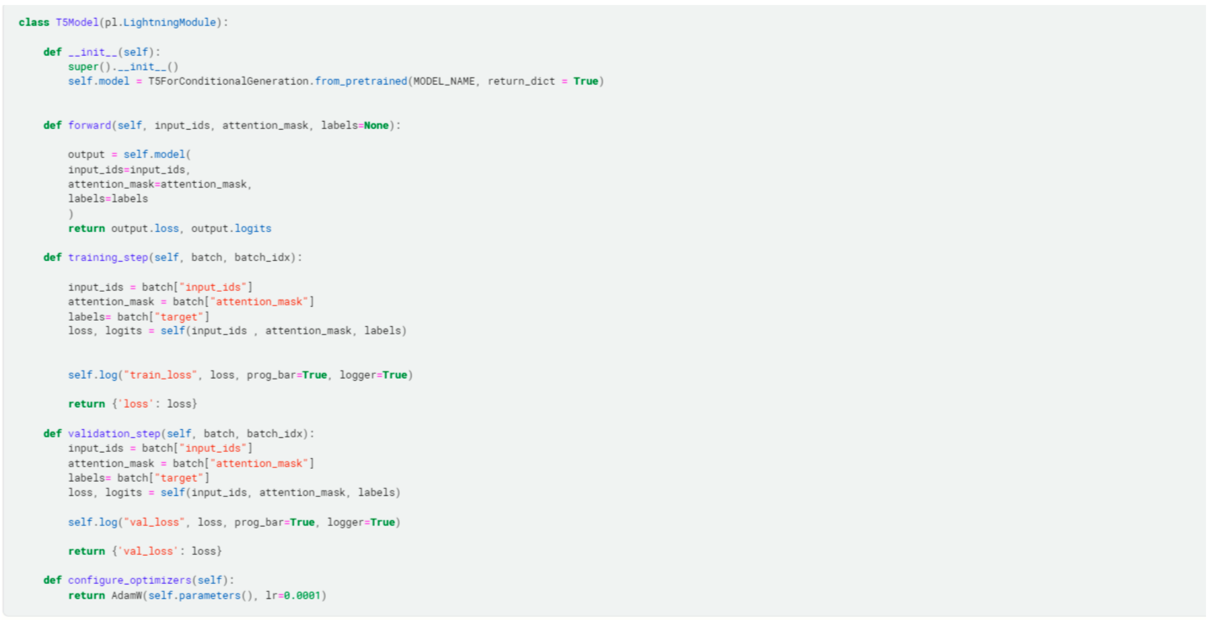
The forward method of the LightningModule class defines the forward pass of the model. In the case of a T5-based model, this involves passing the input sequence through the T5 encoder and decoder to generate the output sequence.

The training_step method of the LightningModule class defines what happens during each training step. This typically involves passing the input sequence through the model, computing the loss, and updating the model weights based on the gradient of the loss.

The validation_step method defines what happens during each validation step. This typically involves passing the input sequence through the model, computing the validation loss, and logging any relevant metrics.

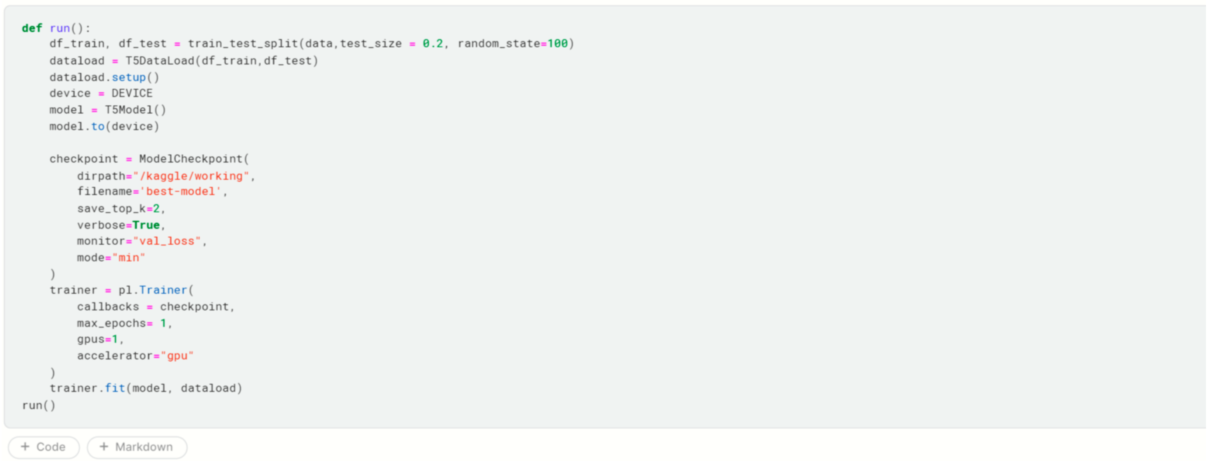
The configure_optimizers method defines the optimizer used to update the model weights during training. This method typically returns a PyTorch optimizer object, such as Adam or SGD.


"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

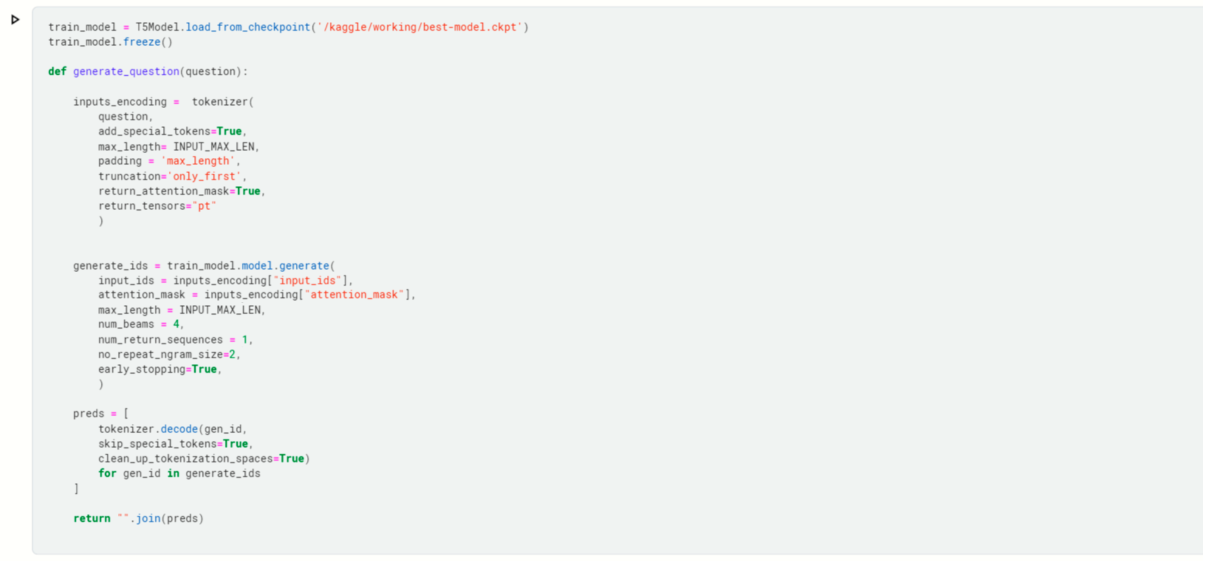


```
class T5Model(pl.LightningModule):

    def __init__(self):
        super().__init__()
        self.model = T5ForConditionalGeneration.from_pretrained(MODEL_NAME, return_dict = True)


    def forward(self, input_ids, attention_mask, labels=None):

        output = self.model(
        input_ids=input_ids,
        attention_mask=attention_mask,
        labels=labels
        )
        return output.loss, output.logits

    def training_step(self, batch, batch_idx):

        input_ids = batch["input_ids"]
        attention_mask = batch["attention_mask"]
        labels= batch["target"]
        loss, logits = self(input_ids , attention_mask, labels)


        self.log("train_loss", loss, prog_bar=True, logger=True)

        return {'loss': loss}

    def validation_step(self, batch, batch_idx):
        input_ids = batch["input_ids"]
        attention_mask = batch["attention_mask"]
        labels= batch["target"]
        loss, logits = self(input_ids, attention_mask, labels)

        self.log("val_loss", loss, prog_bar=True, logger=True)

        return {'val_loss': loss}

    def configure_optimizers(self):
        return AdamW(self.parameters(), lr=0.0001)
```

Training Step



```
def run():
    df_train, df_test = train_test_split(data,test_size = 0.2, random_state=100)
    dataload = T5DataLoad(df_train,df_test)
    dataload.setup()
    device = DEVICE
    model = T5Model()
    model.to(device)

    checkpoint = ModelCheckpoint(
        dirpath="/kaggle/working",
        filename='best-model',
        save_top_k=2,
        verbose=True,
        monitor="val_loss",
        mode="min"
    )
    trainer = pl.Trainer(
        callbacks = checkpoint,
        max_epochs= 1,
        gpus=1,
        accelerator="gpu"
    )
    trainer.fit(model, dataload)
run()
```



```python
train_model = T5Model.load_from_checkpoint('/kaggle/working/best-model.ckpt')
train_model.freeze()

def generate_question(question):

    inputs_encoding =  tokenizer(
        question,
        add_special_tokens=True,
        max_length= INPUT_MAX_LEN,
        padding = 'max_length',
        truncation='only_first',
        return_attention_mask=True,
        return_tensors="pt"
        )


    generate_ids = train_model.model.generate(
        input_ids = inputs_encoding["input_ids"],
        attention_mask = inputs_encoding["attention_mask"],
        max_length = INPUT_MAX_LEN,
        num_beams = 4,
        num_return_sequences = 1,
        no_repeat_ngram_size=2,
        early_stopping=True,
        )

    preds = [
        tokenizer.decode(gen_id,
        skip_special_tokens=True,
        clean_up_tokenization_spaces=True)
        for gen_id in generate_ids
    ]

    return "".join(preds)
```

IV. Hugging Face Model

a. Hugging Face Model- t5 Base

The model picked for fine tuning is the t5 base hugging model.
Model Link : https://huggingface.co/t5-base
Model details:

The developers of the Text-To-Text Transfer Transformer (T5) wrote:

With T5, we propose reframing all NLP tasks into a unified text-to-text-format where the input and output are always text strings, in contrast to BERT-style models that can only output either a class label or a span of the input. Our text-to-text framework allows us to use the same model, loss function, and hyperparameters on any NLP task.

T5-Base is the checkpoint with 220 million parameters.

b. Developed by: Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. See associated paper and GitHub repo
c. Model type: Language model
d. Language(s) (NLP): English, French, Romanian, German
e. Licence: Apache 2.0
f. Related Models: All T5 Checkpoints
g. Resources for more information:

i. Research paper
ii. Google's T5 Blog Post
iii. GitHub Repo
iv. Hugging Face T5 Docs

Hyperparameter Tuning procedure

- Tuning hyper-parameters remarkably improves the performance of the T5 model which results in a better chatbot conservation result.
- Manual tuning: In this, one uses the accuracy curve / other curves to select the best hyper-parameter to train T5.
- Automatic tuning: a more powerful hyper-parameter tuning technique through search, thus the corresponding code for the T5 base automatic tuning is provided in the pytorch notebook for T5 base tuning.
- On the test dataset
  - Test dataset is regarded as new data to a T5 base
  - A model cannot see class labels of test data. Thus, it is impossible to tune parameters on the test dataset
  - In other words, hyperparameter tuning is performed on the training set.
- On the training dataset
  - A hyperparameter must be tuned on training data set
  - However, this is bad practice because it leads to overfitting in some cases

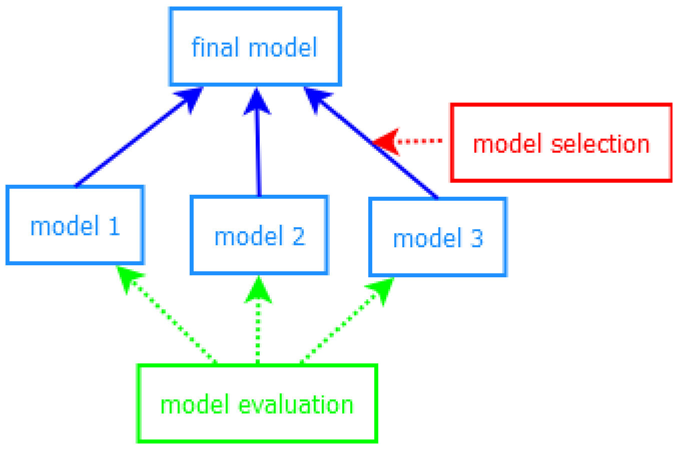
h. Model Selection and Hyperparameter Tuning

Typically, a machine learning algorithm has parameters. Potentially more than one hundred parameters are included in certain deep learning models. Selecting a model involves choosing the one with the most advantageous hyperparameters. Both human and automated methods are available for this. Model evaluation and hyperparameter adjustment are discussed during this lecture.

- Explain concepts of model selection, overfitting, hyperparameter tuning
- Introduce two methods used in model selection: holdout method and k-fold cross-validation
- Present several examples of manual and automatic tuning of hyperparameters

i. What is Model Selection?

- Model Selection: Whether to choose the best modelling from a pool of probable candidates depending on how well they performed using the training data.
  - select a model from different types of models, e.g. from three classifiers: logistic regression, decision tree and KNN
  - select a model from the same model with different model hyperparameters, e.g., from the decision tree model with different tree depths
- Model evaluation: how to evaluate the performance of a candidate models using certain metrics, e.g., accuracy



j. Considerations in Model Selection

- In a business project, model selection is not only a technical problem
- There are many considerations such as
  - business requirements
  - cost
  - performance
  - state-of-the-art
  - maintenance
- It is difficult or impossible to find the best model meeting all criteria. Usually, turn to find a satisfactory model
- The best model in terms of technical performance may not be the best in business
  - Netflix prize In 2009, 'Netflix held the Netflix Prize open competition for the best algorithm to predict user ratings for films. The grand prize was $1,000,000 and was won by BellKor's Pragmatic Chaos team. This is the dataset that was used in that competition'. But this algorithm was never used by Netflix

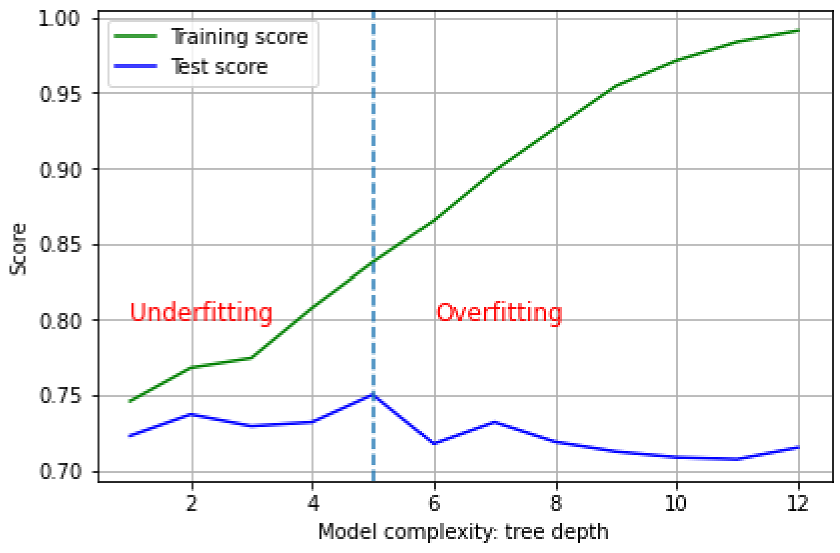
k. Dataset for Tuning Hyperparameter

- On the test dataset
  - Test dataset is regarded as new data to a model
  - A model cannot see class labels of test data. In Kaggle competitions, class labels of test data are set unseen to participants

- Thus, it is impossible to tune parameters on the test dataset
- In other words, hyperparameter tuning in our previous labs is wrong

- On the training dataset
  - A hyperparameter must be tuned on training data set
  - However, this is bad practice because it leads to overfitting

I. Overfitting and Underfitting



- Overfitting
  - a model performs very well on the training data
  - but not on the test data
  - e.g. overfit when the tree depth = 10
- Underfitting
  - a model fits the training data not well
  - and does not the fit test data very well
  - e.g. underfit when the tree depth = 1

V. Conclusions

Conversational agents, or chatbots, converse with people in a variety of ways. Chatbots are gaining traction in a variety of fields, from administrative assistants to ticketing services to physical therapists. In fact, using a chatbot instead of people might be highly reasonably priced. It can be difficult to create a chatbot that is as effective as a human being though.