**BERT for Question Answer**

The code will be written in Python and will use the PyTorch library and the Hugging Face Transformers library to fine-tune the BERT model for Question Answering tasks. Here are the steps that will be taken to write this code:

* **Import the required libraries:** The code starts with importing the required libraries, including PyTorch, the BERTForQuestionAnswering model, the BertTokenizer, and the DataLoader and Dataset classes from PyTorch.
* **Define the path to the custom dataset:** A path to the custom dataset is defined to load the data.
* **Dataset Used:** Squad Dataset will be used. Link ⇒ [Stanford Question Answering Dataset | Kaggle](https://www.kaggle.com/datasets/stanfordu/stanford-question-answering-dataset)
* **Load the BERT model and tokenizer:** The pre-trained BERT model and tokenizer are loaded using the from\_pretrained() function from the transformers library from HuggingFace.
* **Define the batch size and maximum sequence length:** The batch size and maximum sequence length are defined for the BERT model.
* **Define a custom dataset class for the QA task:** A custom dataset class is defined to read and preprocess the data for the QA task. This class reads the data from the custom dataset, tokenizes the question and context using the tokenizer, and creates input\_ids, token\_type\_ids, attention\_mask, start\_position, and end\_position tensors for each example.
* **Create the custom dataset:** An instance of the custom dataset is created using the data\_path, tokenizer, and max\_seq\_length.
* **Define a function to pad the batch inputs:** A function is defined to pad the batch inputs to the maximum sequence length and concatenate them to form a batch.
* **Create a data loader for the custom dataset:** A data loader is created using the DataLoader class from PyTorch, batch\_size.
* **Define the optimizer and learning rate:** An AdamW optimizer is defined with a learning rate of 3e-5.
* **Train the model:** The fine-tuning loop is defined for a fixed number of epochs. The model is trained on each batch of data from the data loader, and the loss is calculated using the start\_positions and end\_positions tensors. The loss is then back propagated, and the optimizer is updated.
* **Save the fine-tuned model:** Finally, the fine-tuned BERT model is saved using the save\_pretrained() function from the transformers library.
* **Predict using saved model:** At last a prediction will be done using the saved model.

**BioBert for Question Answer**

The code can be implemented with the following steps:

* **Install the necessary libraries:** transformers and datasets.
* **Import the necessary libraries**: torch, AutoTokenizer, AutoModelForQuestionAnswering, Trainer, TrainingArguments, and load\_dataset from datasets.
* **Loading pretrained Model:** Load the BioBERT model and tokenizer using the AutoTokenizer and AutoModelForQuestionAnswering classes from the transformers library.
* **Loading the dataset:** Load the PubMed dataset using the load\_dataset function from the datasets library with the training split. Link:- [pubmed · Datasets at Hugging Face](https://huggingface.co/datasets/pubmed)
* **Tokenizing the inputs:** Define a preprocess\_function that tokenizes the inputs and encodes them as inputs to the model, and maps the start and end positions of the answer to the tokenized inputs.

Preprocess the dataset by applying the preprocess\_function to each example in the dataset, removing the columns 'question', 'context', and 'answers'.

* **Training Arguments:** Define the training arguments using the TrainingArguments class from the transformers library. These arguments specify the output directory, the evaluation strategy, the number of save models, the learning rate, the batch size, the number of epochs, the weight decay, and the logging options.
* **Trainer Class:** Define the trainer using the Trainer class from the transformers library, passing in the model, training arguments, preprocessed dataset, data collator function, and tokenizer. Trainer class is equivalent to model.fit().

Train the model using the train method of the trainer object.

* **Validation Dataset:** Load the validation dataset using the load\_dataset function from the datasets library with the validation split.

Preprocess the validation dataset by applying the preprocess\_function to each example in the dataset, removing the columns 'question', 'context', and 'answers'.

* Evaluate the model using the evaluate method of the trainer object, passing in the preprocessed validation dataset.
* Print the evaluation results

**T5 - Transformer for Question Answer**

* The T5 transformer is a slightly different transformer than the previous 2 transformers.
* Where the previous transformers needed the start and end tokens as the target, here we don’t need starting or ending tokens.
* We just need the text of the answer as the target.
* Hence, here we need only three things, 1)Context, 2) Question 3) Answer.
* T5 is the simplest and more powerful of all the three transformers as it has both encoder and decoder and is also known as a multi purpose transformer.

**Dataset we will use here will be the PubMed dataset as we used in the Biobert Transformer.** [**pubmed · Datasets at Hugging Face**](https://huggingface.co/datasets/pubmed)

* **Import the libraries:** We start by importing the necessary libraries like T5 tokenizer, T5 conditional generation, Dataloader,dataset.
* **Load the dataset:** We load the dataset. In our case, we will use the PubMed dataset.
* **Class QuestionGenerationDataset:** This is a class which performs the heavy lifting of the entire task. Here the loaded dataset is sent. The entities are separated here, viz: Context, Question and Answer. These entities are converted into tokens. Their input ids and attention masks are extracted. The format in which input will be fed to the model is defined here. The final output of this class is tokenized inputs and tokenized targets.
* **Class T5FineTuner:** This is the main class where the entire model is defined. Also the model input pipeline is defined in this class.

Training steps and Validation steps are also defined here.

Optimizer is defined in this class and data loaders are prepared in this class.

* **Model Training:** Finally the model is run using all the entities which we have defined above.

The loss is taken from the ‘model’ class of pytorch and loss is propagated backwards.

The model is run for desired epochs and when the training is done the model is saved.

* **Inference:** Here the saved model is loaded and the inference is done and the output is printed for inferences.