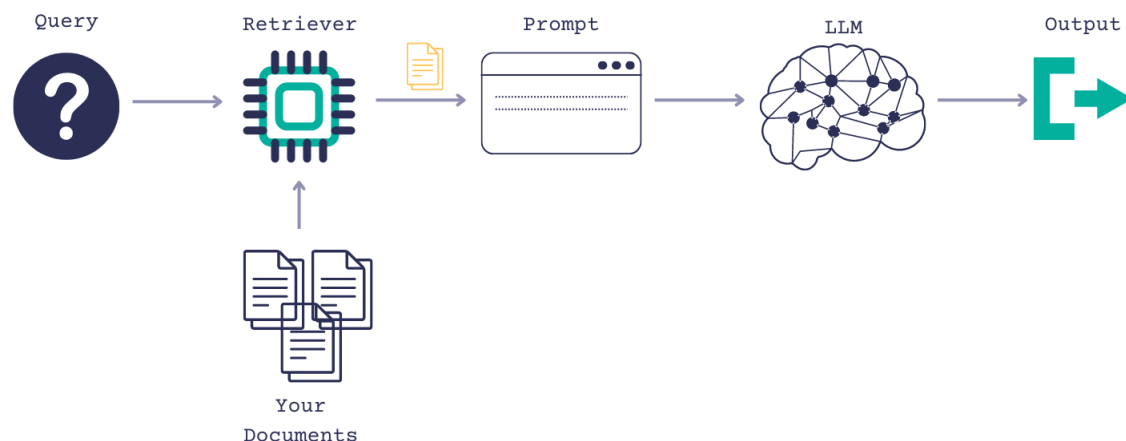


# From Training to Model Deployment: Harnessing Intel oneAPI's Potential

Hi reader, I am Ramashish Gupta, a 4th year undergrad from IIT Kharagpur presenting my intel oneAPI LLM Challenge solution. Generative Question-Answer (Q&A) models, powered by advanced deep learning techniques, have revolutionized the landscape of natural language understanding and information retrieval. These models generate human-like text responses to questions and find application across diverse domains.



The whole workflow can be understood by the above diagram, where **Query** i.e. the question, **Documents** i.e. the context and **LLM** is the model that we are going to train.

Below, we outline various use cases and requirements for these generative Q&A models, highlighting their role in semantic search applications and beyond:

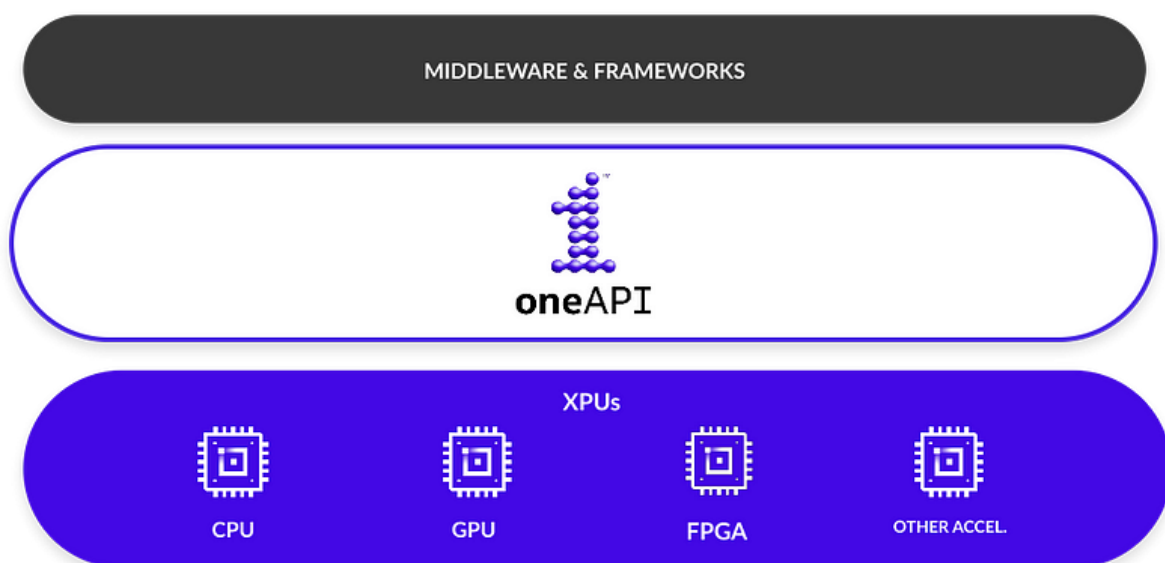
- > **Customer Support Chatbots:** Businesses leverage generative Q&A models to create sophisticated chatbots capable of engaging with customers in a human-like manner. These chatbots understand and respond to a wide array of customer queries.
- > **Educational Tools:** In educational settings, generative Q&A models serve as virtual tutors. Students pose questions related to their coursework, and the models provide detailed explanations and solutions, enhancing the learning experience.
- > **Semantic Search:** Generative Q&A models are instrumental in semantic search applications, going beyond traditional keyword-based search engines. They comprehend the meaning behind user queries and retrieve information based on semantic relevance.
- > **Domain-Specific Expertise:** By fine-tuning generative Q&A models with domain-specific data, they can serve as experts in various fields. For example, a healthcare Q&A model can offer medical advice, and a legal Q&A model can assist with legal inquiries.

Generative Q&A models continue to evolve, offering versatile and valuable solutions across a spectrum of applications where understanding and generating human-like text responses to questions is paramount.

Training generative Q&A models is an immensely complex task, primarily due to the demanding infrastructure requirements, necessitating powerful compute resources and substantial datasets.

## **Introducing Intel oneAPI**

oneAPI is an open, standards-based programming model that frees developers to use a single code base across multiple architectures—CPU, GPU, FPGA, and other accelerators. The result is quicker computation without vendor lock-in.



The Intel oneAPI toolkit is a comprehensive suite of high-performance tools designed for creating Data Parallel C++ applications and oneAPI library-based applications. These toolkits cater to specific domains, with the Intel oneAPI Base Toolkit serving as the foundation for all others.

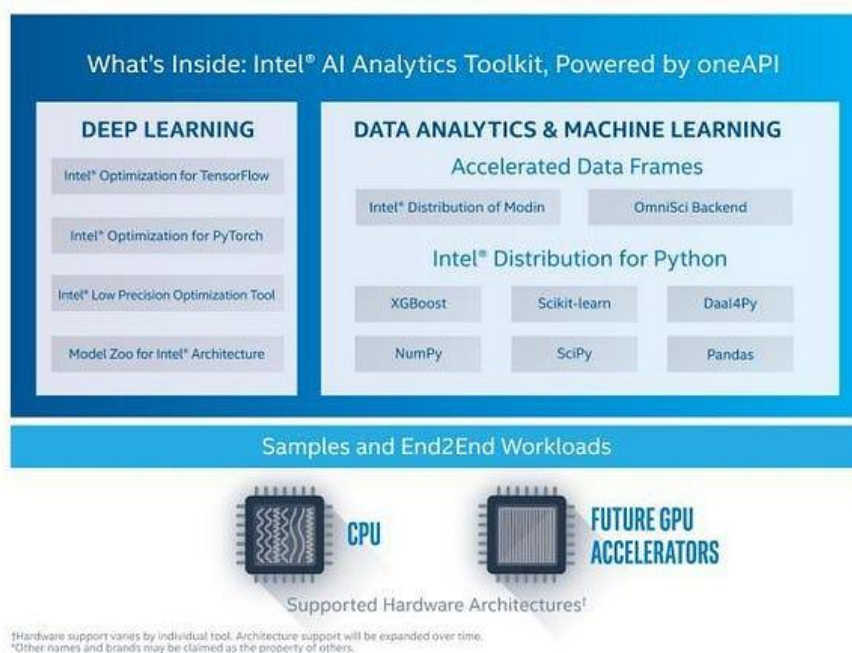
You have a choice of seven meticulously curated toolkits each of these toolkits caters to a unique user base and offers top-tier features. In this blog, we'll delve into the Intel AI Analytics Toolkit, exploring how to train a generative Question Answering model and subsequently use it to create a web application.

## Intel® AI Analytics Toolkit

The AI Kit equips data scientists, AI developers, and researchers with familiar Python tools and frameworks to expedite end-to-end data science and analytics on Intel® architecture. Leveraging oneAPI libraries for low-level compute enhancements, it optimizes performance across preprocessing, machine learning, and enables efficient model development through enhanced interoperability.

With this toolkit, you can

- > Achieve high-performance deep learning training on Intel® XPU<sup>1</sup>s, seamlessly integrating fast inference into your AI workflow. Utilize Intel®-optimized deep learning frameworks for TensorFlow\* and PyTorch\*, including pretrained models and low-precision tools.
- > Experience effortless acceleration for data preprocessing and machine learning workflows using compute-intensive Python packages like Modin\*, scikit-learn\*, and XGBoost.
- > Benefit from direct access to Intel's analytics and AI optimizations, ensuring smooth and harmonious software functionality.



## The Dataset

The dataset we're working with is designed for training a question-answering model. Each sample consists of a substantial context and a question related to that context, for which the model must generate an answer.

Here's an example from the dataset:

**CONTEXT:** Malawi (, or ; or [maláwi]), officially the Republic of Malawi, is a landlocked country in southeast Africa that was formerly known as Nyasaland. It is bordered by Zambia to the northwest, Tanzania to the northeast, and Mozambique on the east, south, and west. Malawi is over with an estimated population of 16,777,547 (July 2013 est.). Its capital is Lilongwe, which is also Malawi's largest city; the second largest is Blantyre, the third is Mzuzu, and the fourth largest is its old capital Zomba. The name Malawi comes from the Maravi, an old name ...

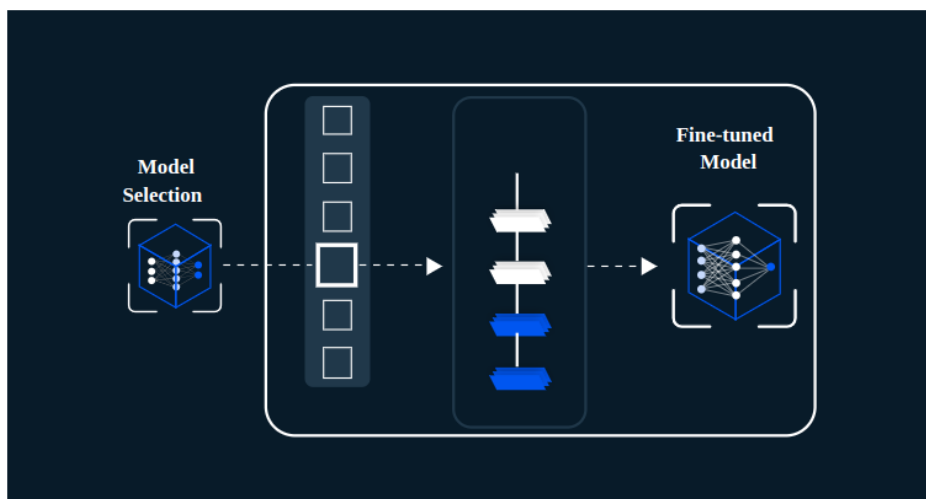
**QUESTION:** Is it a large country?

**ANSWER:** No

This dataset is not your typical extractive question-answering dataset, where the answer can be found verbatim within the context. In fact, upon analysis, it was discovered that 35% of the answers were not present in the context. Hence, traditional encoder models that locate start and end indices of answer text won't suffice. Instead, a generative question-answering model, employing an encoder-decoder architecture, is required for this unique dataset. Also training a separate model for yes-no questions and true-false questions will overcomplicate.

## Fine Tuning the model on the given dataset

In our journey to fine-tune the [model](#), we zeroed in on the T5 architecture, a formidable encoder-decoder model. This model was meticulously fine-tuned using the [SquadV2](#) dataset. We delved into several other encoder-decoder models, including BART and GPT-2. Through rigorous experimentation, we discovered that the T5 model consistently outperformed the others on our validation dataset.



To elevate the training process to its zenith, I harnessed the capabilities of the Intel Developer Cloud. Within this ecosystem, I seamlessly integrated two powerful oneAPI services, which synergized to unlock the full potential of Intel hardware, ensuring the most efficient training environment possible.

To run the training first ssh login into your intel developer cloud instance, and run the following commands:

```
sycl-ls
srun --pty
bash source /opt/intel/oneapi/setvars.sh
git clone https://github.com/ramashisx/oneAPI_hackathon_submission
cd ./oneAPI_hackathon_submission/scripts python train.py
```

### [PyTorch\\* Optimizations from Intel](#)

Intel stands as a significant contributor to PyTorch\*, consistently delivering essential optimizations that enhance the performance of PyTorch within Intel architectures. The AI Kit offers the most current binary version of PyTorch, rigorously tested for compatibility with the entire kit. Additionally, it incorporates the Intel® Extension for PyTorch\*, introducing the latest Intel optimizations and user-friendly features to further elevate your PyTorch experience.

With a few lines of code, you can use Intel Extension for PyTorch to take advantage of the most up-to-date Intel software and hardware optimizations for PyTorch, automatically mix different precision data types to reduce the model size and computational workload for inference and add your own performance customizations using APIs.

### [Intel® Extension for PyTorch\\*](#)

The Intel® Extension for PyTorch\* elevates PyTorch\* by infusing it with the latest feature enhancements and optimizations, unlocking superior performance on Intel hardware. These optimizations harness the capabilities of AVX-512 Vector Neural Network Instructions and Intel® Advanced Matrix Extensions on Intel CPUs, as well as Intel Xe Matrix Extensions AI engines on Intel discrete GPUs. Additionally, with the PyTorch\* xpu device, Intel® Extension for PyTorch\* facilitates effortless GPU acceleration for Intel discrete GPUs, seamlessly integrating them with PyTorch\* for enhanced performance.

Now you are good to go, as you can see in the train.py file using ``cat train.py`` this training pipeline utilizes both **Pytorch Optimization for Intel** and **Intel Extension for Pytorch** to optimize the model and use the most out of your Intel hardware while training.

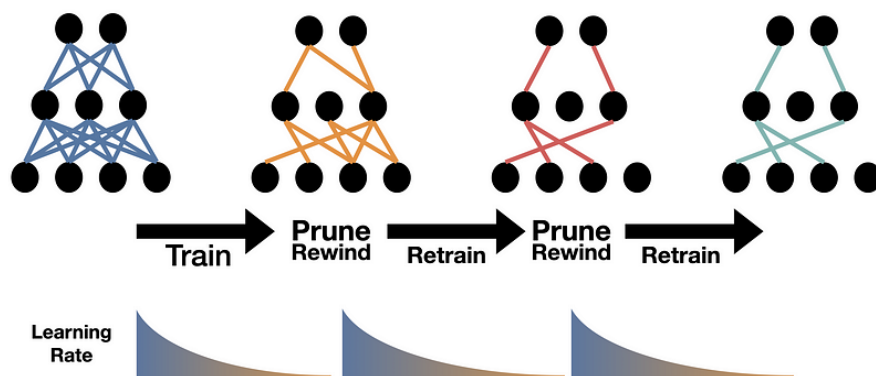
```
Activities Terminal Oct 20 13:29 lord@Linux -
ui36098@idc-beta-batch-pvc-node-86:~$ python train.py
Intel(R) Extension for SciKit-Learn enabled (https://github.com/intel/scikit-learn-intelx)
/home/ui36098/.local/lib/python3.9/site-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Python extension: "If you don't plan on using image functionality from 'torchvision.io', you can ignore this warning. Otherwise, there might be something wrong with your environment. Did you have 'libjpeg' or 'libpng' installed before building 'torchvision' from source?"
  warn(
The 'xla_device' argument has been deprecated in v4.4.0 of Transformers. It is ignored and you can safely remove it from your 'config.json' file.
The 'xla_device' argument has been deprecated in v4.4.0 of Transformers. It is ignored and you can safely remove it from your 'config.json' file.
You are using the default legacy behaviour of the <class 'transformers.models.t5.tokenization_t5.T5Tokenizer'>. This is expected, and simply means that the 'legacy' (previous) behavior will be used so nothing changes for you. If you want to use the new behaviour, set 'legacy=False'. This should only be set if you understand what it means, and thoroughly read the reason why this was added as explained in https://github.com/huggingface/transformers/pull/24065
The 'xla_device' argument has been deprecated in v4.4.0 of Transformers. It is ignored and you can safely remove it from your 'config.json' file.
The 'xla_device' argument has been deprecated in v4.4.0 of Transformers. It is ignored and you can safely remove it from your 'config.json' file.
Map: 100% | 60880/60880 [01:07:00:00, 898.01 examples/s]
Map: 100% | 6011/6011 [00:07:00:00, 848.96 examples/s]
trainable model parameters: 222903552
all model parameters: 222903552
percentage of trainable model parameters: 100.00%
/home/ui36098/.local/lib/python3.9/site-packages/intel_extension_for_pytorch/frontend.py:611: UserWarning: Conv BatchNorm folding failed during the optimize process.
  warnings.warn(
/home/ui36098/.local/lib/python3.9/site-packages/intel_extension_for_pytorch/frontend.py:618: UserWarning: Linear BatchNorm folding failed during the optimize process.
  warnings.warn(
/home/ui36098/.local/lib/python3.9/site-packages/torch/cuda/amp/grad_scaler.py:129: UserWarning: torch.cuda.amp.GradScaler is enabled, but CUDA is not available. Disabling.
  warnings.warn("torch.cuda.amp.GradScaler is enabled, but CUDA is not available. Disabling.")
[0x] | 0/15000 [00:00:07, 71it/s]
You're using a T5TokenizerFast tokenizer. Please note that with a fast tokenizer, using the '__call__' method is faster than using a method to encode the text followed by a call to the 'pad' method to get a padded encoding.
[loss: 1.3424, 'learning_rate': 4.833333333333334e-05, 'epoch': 0.07]
[loss: 0.9958, 'learning_rate': 4.066666666666667e-05, 'epoch': 0.13]
[loss: 0.913, 'learning_rate': 4.5e-05, 'epoch': 0.2]
[loss: 0.9013, 'learning_rate': 4.333333333333334e-05, 'epoch': 0.27]
[loss: 0.8899, 'learning_rate': 4.166666666666667e-05, 'epoch': 0.33]
[loss: 0.8329, 'learning_rate': 4e-05, 'epoch': 0.4]
[loss: 0.8635, 'learning_rate': 3.833333333333334e-05, 'epoch': 0.47]
[loss: 0.8499, 'learning_rate': 3.666666666666666e-05, 'epoch': 0.53]
[loss: 0.8509, 'learning_rate': 3.5e-05, 'epoch': 0.6]
[loss: 0.8312, 'learning_rate': 3.333333333333333e-05, 'epoch': 0.67]
[eval_loss: 0.818685822486877, 'eval_exact_match': 0.3434, 'eval_f1_score': 0.3434, 'eval_gen_len': 4.917712902737861, 'eval_runtime': 1413.9965, 'eval_samples_per_second': 4.675, 'eval_steps_per_second': 0.585, 'epoch': 0.67]
[loss: 0.8260, 'learning_rate': 3.166666666666666e-05, 'epoch': 0.73]
[loss: 0.8413, 'learning_rate': 3e-05, 'epoch': 0.8]
[loss: 0.84, 'learning_rate': 2.833333333333333e-05, 'epoch': 0.87]
[loss: 0.8084, 'learning_rate': 2.666666666666667e-05, 'epoch': 0.93]
[loss: 0.8029, 'learning_rate': 2.5e-05, 'epoch': 1.0]
[loss: 0.8, 'learning_rate': 2.333333333333333e-05, 'epoch': 1.07]
[loss: 0.8, 'learning_rate': 2.166666666666667e-05, 'epoch': 1.13]
[loss: 0.8, 'learning_rate': 2e-05, 'epoch': 1.2]
[loss: 0.8, 'learning_rate': 1.833333333333333e-05, 'epoch': 1.27]
[loss: 0.8, 'learning_rate': 1.666666666666667e-05, 'epoch': 1.33]
57% | 10000/15000 [3:16:15<18:11, 4.58it/s]
58% | 43/827 [02:34<47:31, 3.64s/it]
Jobs: 0 | 0:run* "idc-beta-batch-head-n" 00:59 20-Oct-23
```

## Results Time

The model I trained managed to attain a leaderboard score of **0.376**, ultimately securing my **first-place position in Phase 1**. However, it's crucial to highlight that the model's actual capabilities surpassed this score. **The evaluation script, which considered exact matches in a case-sensitive manner, adversely affected the accuracy metric.** Upon carefully exploring the data I also found that the casing in answer didn't follow a definite pattern so the model never learnt how to follow the exact casing.

## Quantization and Model Pruning

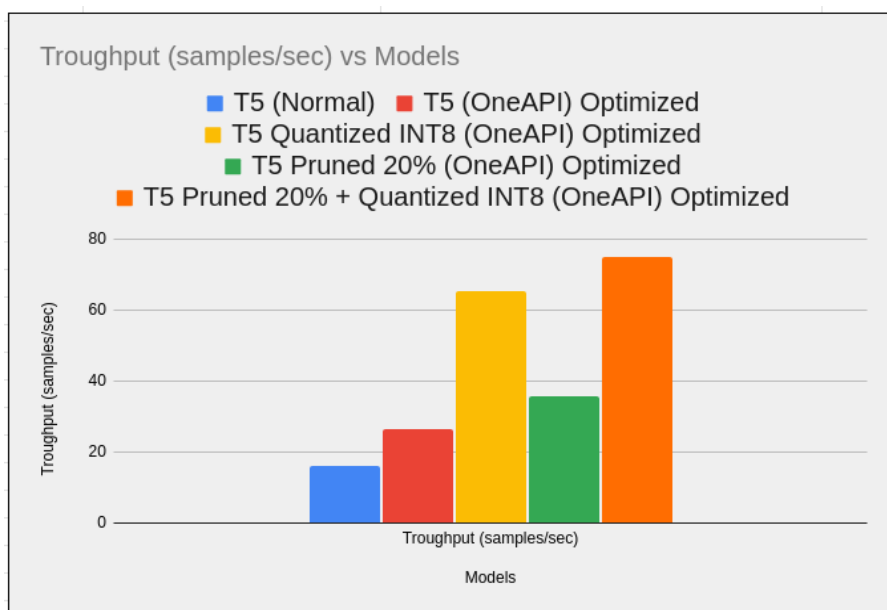
In my journey to optimize machine learning models, I harnessed two vital techniques: quantization and pruning. Quantization, the process of reducing numerical precision, allows the model to run faster and more efficiently, while pruning involves eliminating less critical parameters, boosting efficiency without compromising performance. These methods became my toolkit for achieving a delicate balance between model accuracy and operational speed. My journey began with pruning, where I trimmed away extraneous model elements. Subsequently, I delved into quantization to reduce computational overhead. This combined approach refined my model, making it well-suited for applications that prioritize efficiency without substantial accuracy loss. It's this dual strategy that empowered my model to shine in efficiency-focused scenarios.



Once your training process is successfully completed, it's time to take the next steps in model optimization. Begin by navigating to your trained model directory and execute the following commands.

```
bash run_pruning.sh
```

When prompted, provide the path to your trained model. This step utilizes the Intel Neural Compressor to prune your model, and you have the flexibility to fine-tune parameters by adjusting values in the `run_pruning.sh` file.



After pruning, navigate to your pruned model, and now, let's venture into quantization.

```
bash run_quantization.sh
```

After running the command provide the path to your pruned model when prompted. Once again, this process leverages the Intel Neural Compressor to quantize your model, optimizing it for efficiency.

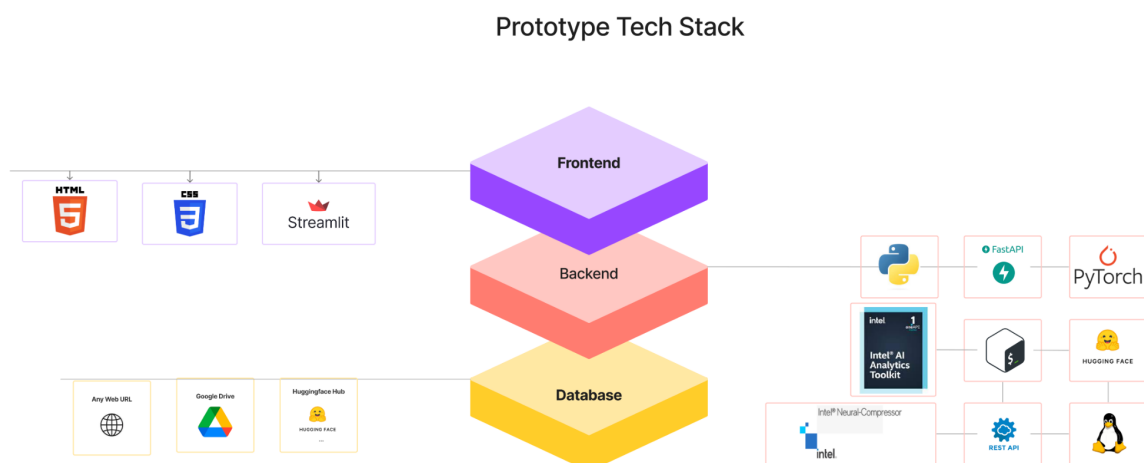
Finally, navigate to your final quantized model. While it may not be the most accurate model, it offers the highest throughput, making it ideal for various applications.

## Intel® Neural Compressor

The Intel® Neural Compressor is a versatile tool designed for model compression, effectively reducing the size of models and boosting the speed of deep learning inference for deployment on CPUs or GPUs. This open-source Python\* library streamlines well-known model compression techniques like quantization, pruning, and knowledge distillation, making them accessible across a range of deep learning frameworks.

This multifaceted library equips you with a range of powerful tools. It enables the acceleration of model convergence through automated accuracy-driven strategies for quantized models, simplifies the optimization of large models by efficiently pruning less vital parameters, empowers the refinement of smaller models for deployment via knowledge distillation from larger ones, and provides a user-friendly, one-click approach to initiate model compression, streamlining the entire process for your convenience.

## Deploying this model as a WebApp





## Web App Structure

This web application utilizes FastAPI for the backend and Streamlit for the frontend. Here's how it all comes together:

**Backend (FastAPI):** FastAPI serves as the backbone of the web application. It handles HTTP requests, processes data, and communicates with the machine learning model. It also automatically generates an interactive API documentation, which is invaluable for developers.

**Frontend (Streamlit):** The user interface of the web app is built using Streamlit. It simplifies the creation of interactive, data-driven applications. Streamlit allows you to incorporate widgets and components, such as sliders, buttons, and charts, to make the user experience intuitive and engaging.

**Model Deployment and Optimization:** The machine learning model deployed in this web app has been optimized for Intel hardware using the Intel Optimization for PyTorch through the Intel AI Analytics Toolkit. However, the code has been structured in a way that it can be used on various hardware platforms, ensuring that it remains versatile and accessible to a wide range of users.

**Modularity:** To reuse this web app for a different model, you can modify the model's configuration and ensure that you update the MD5 checksum in the provided bash script. This modularity makes the web app adaptable for various machine learning models and use cases.

Now, let's proceed with turning this well-trained and optimized model into an actual product. Navigate to the "webapp" folder within the repository, and run the following command.

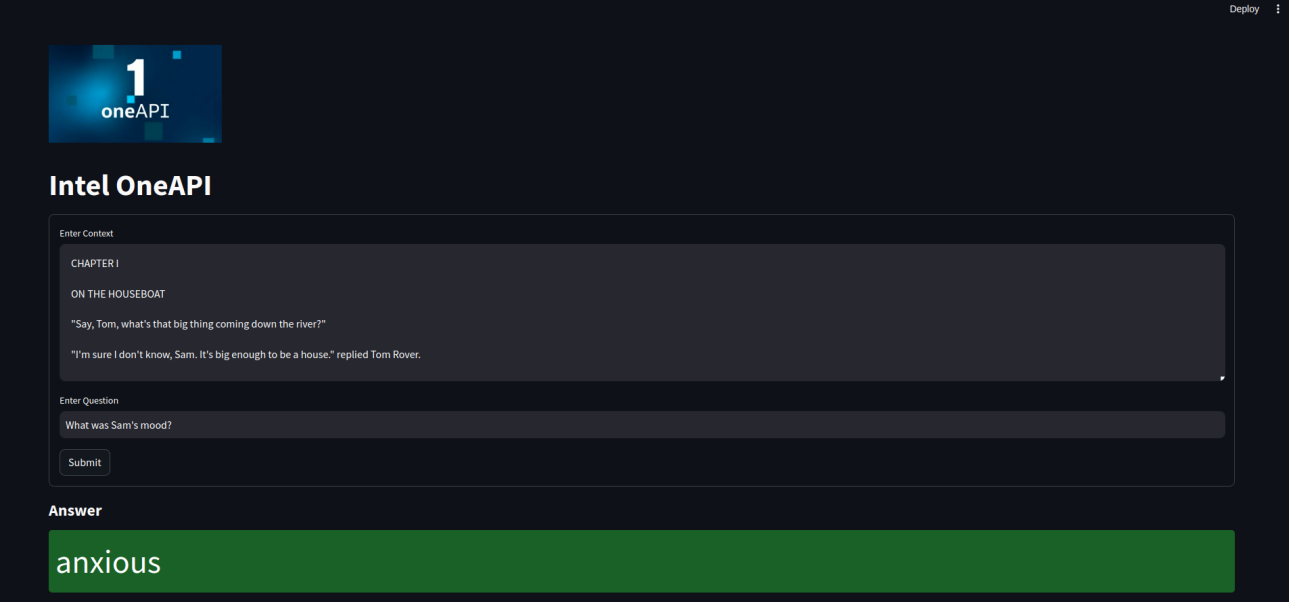
```
pip install -r requirements.txt
bash run.sh
```

If you are running on a remote instance make sure you port forward two ports by running the following in a new terminal

```
ssh myidc
ssh myidc -L 4444:10.10.10.X:4444
ssh myidc -L 4445:10.10.10.X:4445
```

Replace `10.10.10.X` with the Public IP of your remote machine, connect to `localhost:4444` for backend `localhost:4444/docs` for api documentation and connect to `localhost:4445` for UI.

This script will launch both the backend and frontend components of the model. Additionally, it will download and verify the model for use. If you intend to adapt this web application for a different model, remember to update the MD5 checksum within the bash script accordingly.



The screenshot shows a web application interface for Intel OneAPI. At the top left is a logo with a large '1' and the text 'oneAPI'. Below it, the text 'Intel OneAPI' is displayed. The main interface is divided into two sections: 'Enter Context' and 'Enter Question'. The 'Enter Context' section contains a text area with the following text: 'CHAPTER I', 'ON THE HOUSEBOAT', '"Say, Tom, what's that big thing coming down the river?"', and '"I'm sure I don't know, Sam. It's big enough to be a house." replied Tom Rover.'. The 'Enter Question' section contains a text input field with the text 'What was Sam's mood?' and a 'Submit' button. Below the input fields, the word 'Answer' is displayed, followed by a green box containing the word 'anxious'. In the top right corner, there is a 'Deploy' button and a menu icon.

## Conclusion

In conclusion, Intel's oneAPI toolkit is revolutionizing software development by breaking free from hardware constraints. Its comprehensive toolkits cater to diverse domains, and the Intel AI Analytics Toolkit, as showcased here, accelerates deep learning and streamlines Python-based data science.

Intel's collaboration with PyTorch via the Intel Extension enhances performance, while the Intel Neural Compressor enables model compression. The deployment of a web app showcases its adaptability.

Intel's oneAPI is a game-changer, offering flexibility, efficiency, and innovation for developers, transcending hardware limitations to propel us into the future of computing.