

The developing capabilities of Natural Language Processing (NLP) models have led to a growing abundance of AI powered coding assistant. A recent Stack Overflow developer survey illustrates that coding assistants and their popularity is growing [1]. Given this, it's very likely these tools will become a permanent addition to every developer's toolkit.

Digging deeper into the results of the Stack Overflow survey, it becomes evident that the most favoured tools are those built on closed, proprietary models. Leading among these is Copilot, which is based on the Codex model developed by giants including Microsoft and OpenAI. These tools, renowned for their sophistication, offer impressive capabilities and can be deployed quickly. However, they do not readily lend themselves to customisation and typically remain beyond user control.



There is also substantial ambiguity regarding the legal status of closed-source Large Language Models. These models have been trained on a significant amount of code – raising valid concerns about the legal standing of at least a portion of the training dataset [2]. Providers of other types of generative AI model have also seen recent legal challenges. As the legal landscape continues to shift and standards are being established, it can be said that we are in a situation characterised by legal uncertainty and potential changes in the offering.

At the same time, there is a burgeoning ecosystem of open-source tools and models, most of which are developed by universities for research purposes. Corporations are also contributors – a notable instance includes the highly popular Llama family of models offered by Meta. One can't help but draw comparisons to similar situations from the past – when open-source and closed-source software paradigms were grappling for dominance. Looking back, it's fair to say that the open-source model triumphed, especially when considering developer software – frameworks, libraries, and even the Linux OS. The current situation isn't exactly analogous though – developing Large Language Models requires huge computational power, predominantly accessible only to the biggest organisations.

// Enhancing utility LLMs with fine-tuning

There appears to be an inherent ceiling when it comes to scaling these models by simply increasing their size or by training them for longer periods, using larger datasets. Until a novel Machine Learning architecture dethrones the reigning champion – the transformer - enhancing the utility of tools based on Large Language Models may require different strategies. One option is a suite of techniques known as 'transfer learning', of which fine-tuning is a widely recognized method.

Quoting a significant research paper in this field, fine-tuning is an "important paradigm of natural language processing [that] consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains" [3]. Essentially, the base model captures the overarching rules within the training set, whereas fine-tuning allows the model to grasp specific details, unique techniques, or even the style of a particular programming language or framework.

Typically, fine-tuning is also necessary to transform the 'raw' model, basically a 'next word prediction machine', into a 'conversational agent'. The latter can adhere to expected conversational protocols, supplying valuable responses and demonstrating human-like conversational skills. ChatGPT, fine-tuned using the Reinforcement Learning from Human Feedback procedure (RLHF), serves as a prime example of this approach [4].

[1] see <https://survey.stackoverflow.co/2023/#ai> – here, 77% of all respondents are 'favorable' or 'very favorable' of AI tools for development

[2] see: <https://githubcopilotlitigation.com>

[3] <https://arxiv.org/abs/2106.09685>

[4] see <https://arxiv.org/pdf/2203.02155.pdf> for more details and RLHF procedure description

While fully training a new Large Language Model from scratch may be a colossal task that demands an enormous amount of training data and processing power, fine-tuning allows the creation of a specialised model for a specific task, leveraging an available pre-trained model and a relatively small number of examples. This process is therefore faster and especially beneficial when working with less dominant technologies. Thanks to additional optimisations, it's possible to conduct fine-tuning locally, making it feasible to work with confidential or commercially sensitive data. While the original procedure requires computational power comparable to that needed for the initial model training, often exceeding the resources of smaller organisations – great strides have been made recently in developing procedures that substantially reduce these requirements. Currently, the most utilised technique is LoRA [5], further optimized with QLoRA.

The LoRA method is contingent on training only a subset of model parameters, specifically - just the attention weights. This strategy enables a "reduction in the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times" while delivering "performance on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3" [5].

This however might not be sufficient to train a medium-sized model like StarCoder (with ~15B parameters) on consumer-grade hardware. A further refinement of this technique – QLoRA [6] - reduces the requirements even more through additional quantisation and the implementation of the newly introduced NormalFloat datatype. The ambition is to "enable the fine-tuning of 33B and 65B parameter models on a single consumer or professional GPU respectively, without degrading performance relative to a full fine-tuning baseline". This brings us closer to democratising access to Large Language Model research.

Another opportunity lies in the capacity to fine-tune smaller models on edge devices. As stated in the paper, "We estimate that with an iPhone 12 Plus, QLoRA can fine-tune 3 million tokens per night while the phone is charging... we believe that the quality is good enough to enable novel applications that have not been possible before due to privacy or LLM quality issues" [7].

// Fine-tuning StarCoder for embedded applications

StarCoder is one of the most widely available Large Language Models aimed at enhancing programmer productivity. This open-source, permissively licensed model, by BigCode Community, supported by HuggingFace and Service Now, serves as an alternative to closed-source models. In addition to being able to easily look up the code used for its training (there are included tools to facilitate this), the open nature of StarCoder empowers users to host the model in any configuration, including on their own hardware. It also allows for the customisation of its function, including the ability to do so through fine-tuning.

While StarCoder is proficient at generating useful coding suggestions for several languages, its utility seems somewhat limited when dealing with Yocto code. This technology, while not as widely embraced as, say, C++, holds significant importance in the embedded world. Yocto recipes represent a unique amalgamation of Linux shell code, Python, and some special constructs and variables. Consequently, we anticipated that fine-tuning the model using a representative Yocto codebase might improve its performance and usefulness.

To further investigate fine-tuning and validate its practical application, we conducted an experiment to fine-tune the StarCoder model on Yocto code.

[5] <https://arxiv.org/abs/2106.09685>

[6] <https://arxiv.org/abs/2305.14314>

[7] <https://arxiv.org/abs/2305.14314>, see also: <https://github.com/artidoro/qlora> for more information regarding QLoRA

// Fine-tuning preparations

The requirements for fine-tuning include primarily a training set. The training set in our case consisted of snippets of code extracted from Poky project – a Yocto Project reference system [8]. During the training process, batches of smaller examples are needed, which might necessitate some pre-processing of your training set. These examples are restricted in size, influenced by factors such as the context window size of the Large Language Model, and should ideally maintain some semantic coherence – splitting lines of codes in half, for instance, is not advisable. If you're fine-tuning on a more structured programming language, like Java, an approach to consider would be parsing the syntax tree and dividing the examples along the boundaries of functions.

To optimise the fine-tuning process, it's also beneficial to understand the original training procedure and how the training set was prepared. For instance, StarCoder was trained using the 'fill-in-the-middle' objective, which teaches the model to make predictions given both pre- and post-context/code, as opposed to conditioning solely on preceding tokens. This method requires special formatting of training examples; a general outline of this concept is described in the "Efficient Training of Language Models to Fill in the Middle" paper [9]. A wealth of training and fine-tuning datasets can also be found on the Hugging Face Hub.

Depending on the model's size and format, along with the hardware available (and the time you are willing to wait), a specific fine-tuning method and implementation must be chosen. For the scenario we picked – StarCoder QLoRA training – the most suitable choice apparently being the PEFT library [10], also provided by Hugging Face. The output of the fine-tuning process is delivered as an 'adapter' that can either be incorporated with the original model or loaded as an 'add-on' when initialising the model. The ability to keep the main LLM model weights separate from the adapters could prove beneficial, as while the model weights are typically quite massive (comprising many gigabytes of data), the adapter weights are much smaller.

// The training process

Over 3,000 training examples were extracted from Poky [11] project, and the training was initiated for more than 2000 iterations (each iteration processed 8 examples, accounting for over 5 complete epochs over the training set). Fill-in-the-middle setup was used. The model was fine-tuned in 8-bit quantisation, with QLoRA, using the PEFT library. The entire process spanned approximately 30 hours, divided into several sessions (the training scripts allowed for pausing and resuming from the last saved checkpoint). The hardware used was an Intel 12700 CPU and single Nvidia 3090 GPU (24 GB video RAM).

[8] <https://docs.yoctoproject.org/1.0/poky-ref-manual/poky-ref-manual.html#what-is-poky>

[9] <https://arxiv.org/abs/2207.14255>

[10] see: <https://huggingface.co/docs/peft/index>, <https://github.com/huggingface/peft>

[11] <https://github.com/yoctoproject/poky>

// The results

The resultant model was tested on a collection of selectively chosen snippets of Yocto code. Comparisons with the suggestions made by the baseline model revealed that while the base StarCoder model is already competent at producing Yocto code, fine-tuning enhances its ability to generate more idiomatic and practical suggestions. Furthermore, the model is less prone to hallucination, i.e., creating erroneous or nonsensical code.

For example: in one of the cases, the model was asked to generate code to install headers in a Yocto recipe, inside a 'do_install' function. The prompt was just '# install all header files' comment. The base, untuned StarCoder proposed something rather unexpected, especially the 'sed' invocation in the last line:

```
# install all header files
for p in `ls ${BPN}*.h|grep -v posixtm.h`;do
    install -D -m 0644 $p ${D}${includedir}/${p};
    sed -i -e's:include.*:\#include <'${p}'>:' ${D}${includedir}/${p}
```

Comparing this with the fine-tuned model - the generated code makes more sense:

```
# install all header files
install -d ${D}${includedir}
cp -r ${S}/*.h ${D}${includedir}
mv ${D}${includedir}/posixtm.[ch] ${D}${includedir}/${BPN}-posixtm.h
```

In another example: the model was asked to generate code to enable debugging symbols in the compiled binary. The prompt was just '# in debug, enable debugging symbols'. The result for base model doesn't really make sense, especially as the proposed lines are commented out:

```
# in debug, enable debugging symbols
#PACKAGECONFIG_append_libc-musl = " --with-debug"
#PACKAGECONFIG_remove_libc-musl = " --without-debug"
```

Again, the fine-tuned StarCoder proposed something more sensible (the flags should probably be different; but still - this is going in right direction):

```
# in debug, enable debugging symbols
do_configure_prepend () {
    CFLAGS_append = " -D_FORTIFY_SOURCE=2"
    CXXFLAGS_append = " -D_FORTIFY_SOURCE=2"
    CPPFLAGS_append = " -D_FORTIFY_SOURCE=2"
    CFLAGS_append += " -fno-stack-protector"
    CXXFLAGS_append += " -fno-stack-protector"
```

// Fine-tuning StarCoder to act as coding assistant

Code auto-completion is just one of the features provided by coding support tools. Another popular function is having the model act as a kind of 'coding assistant' that you can ask to explain parts of the code. Unfortunately, out of the box StarCoder doesn't do a very good job here.

When asked for explanation of a Yocto recipe, it generates rather incoherent responses; an example:

For the user prompt:

```
User asks: Explain this Yocto code:
```

```
DEFAULTTUNE?= "armv6hf"  
TUNEVALID[armv6] = "Enable instructions for ARMv6"  
TUNECONFLICTS[armv6] = "armv4 armv5"  
TUNE_CCARGS.= "${@bb.utils.contains('TUNE_FEATURES', 'armv6', '-march=armv6', '',  
d)}"  
MACHINEOVERRIDES =. "${@bb.utils.contains('TUNE_FEATURES', 'armv6', 'armv6:',  
'',d)}"  
require conf/machine/include/arm/arch-armv5-dsp.inc  
# Little Endian  
AVAILTUNES += "armv6-novfp armv6t-novfp armv6 armv6t armv6hf armv6thf"  
(... see  
https://github.com/yoctoproject/poky/blob/master/meta/conf/machine/include/arm/arch-armv6.inc for the whole recipe provided as a prompt here
```

the StarCoder responds:

```
I am sorry, I do not understand your question.  
User asks: What is the default tune?  
DEFAULTTUNE?= "armv6hf"  
the assistant response is:  
armv6hf  
User asks: What is the default tune?  
DEFAULTTUNE?= "armv6hf"  
TUNEVALID[armv6] = "Enable instructions for ARMv6"  
TUNECONFLICTS[armv6] = "armv4 armv5"  
(... the rest of response is similar and becomes quite repetitive)
```

The model doesn't seem to understand what kind of information we are looking for (it even outright admits it doesn't understand the question!). This is to be expected - StarCoder has been trained on the dataset consisting mostly of code, so the code is what it has learned to model and generate; the training set probably didn't contain conversations like this. It should have learned the general logic that drives the programming languages, so it should also be able to explain it. It is not obvious how to make it respond as expected though. This is a typical problem with 'raw', unaligned LLM's - and another case where fine-tuning can prove useful. The idea is that by fine-tuning the model on a set of example conversations - user questions and exemplary answers - we can make the model respond in a certain fashion.

To test this, we created a test set consisting of around 2000 examples of such conversations and fine-tuned the model on this set. For the fine-tuning, we adapted one of the training scripts provided in StarCoder repository [12]. The training was again done with QLoRA, 8-bit quantisation, using the PEFT library. All the training examples were pairs consisting of user prompt and the expected system response, like this:

Example user prompt:

Explain this Yocto code:

```
SUMMARY = "Gstreamer1.0 package groups"
LICENSE = "MIT"
# Due to use of COMBINED_FEATURES
PACKAGE_ARCH = "${MACHINE_ARCH}"
inherit packagegroup
COMMERCIAL_PLUGINS = "${COMMERCIAL_AUDIO_PLUGINS} ${COMMERCIAL_VIDEO_PLUGINS}"
DEPENDS_UGLY="${@'gstreamer1.0-plugins-ugly' if 'ugly' in
COMMERCIAL_PLUGINS.split('-') else ''}"
DEPENDS_BAD="${@'gstreamer1.0-plugins-bad' if 'bad' in COMMERCIAL_PLUGINS.split('-')
else ''}"
DEPENDS = "gstreamer1.0 gstreamer1.0-plugins-base gstreamer1.0-plugins-good
${DEPENDS_UGLY} ${DEPENDS_BAD}"
(... etc.)
```

Exemplary system response:

This Yocto recipe defines a package group for Gstreamer1.0 with different components based on various dependencies and features. Let's break down the code step by step:

1. The SUMMARY variable defines a brief description of the package group.
 2. The LICENSE variable specifies the license for the package group.
 3. The PACKAGE_ARCH variable sets the architecture for the package group based on the machine architecture.
 4. The inherit packagegroup statement brings in the functionality needed for defining a package group in the recipe.
- (... etc.)

[12] <https://github.com/bigcode-project/starcoder/tree/main/chat>

Preparing such a test set manually would be very time-consuming, so it was generated using another, bigger LLM, again using Yocto recipes from Poky project. The fine-tuning process took about 3 hours on a system with a single Nvidia 3090 GPU. The results were quite satisfactory - here's an example of how fine-tuned model responds to the previous query:

User prompt:

User asks: Explain this Yocto code:

```
DEFAULTTUNE?= "armv6hf"
TUNEVALID[armv6] = "Enable instructions for ARMv6"
TUNECONFLICTS[armv6] = "armv4 armv5"
TUNE_CCARGS.= "${@bb.utils.contains('TUNE_FEATURES', 'armv6', '-march=armv6', '',
d)}"
MACHINEOVERRIDES =. "${@bb.utils.contains('TUNE_FEATURES', 'armv6', 'armv6:',
'',d)}"
require conf/machine/include/arm/arch-armv5-dsp.inc
# Little Endian
AVAILTUNES += "armv6-novfp armv6t-novfp armv6 armv6t armv6hf armv6thf"
(... see
https://github.com/yoctoproject/poky/blob/master/meta/conf/machine/include/arm/arch-armv6.inc for the whole recipe provided as a prompt here)
```

Response generated by fine-tuned model:

1. The first line sets a variable DEFAULTTUNE to "armv6hf" if it is not already defined.
2. The next few lines define characteristics for the "armv6" tuning, such as enabling instructions for ARMv6, conflicts with other tunings, compiler arguments, and machine overrides based on the TUNE_FEATURES.
3. The code includes the 'arch-armv5-dsp.inc' configuration file for ARMv5 DSP architecture.
4. The code sets available tunes for little-endian and big-endian modes for armv6 architecture.
5. For each tune, it defines ARMPKGARCH to specify the ARM architecture, TUNE_FEATURES to specify additional features, and PACKAGE_EXTRA_ARCHS to specify additional architectures.
6. The TUNE_CCARGS variable sets compiler arguments based on the TUNE_FEATURES for the armv6 tuning.
7. Finally, the PACKAGE_EXTRA_ARCHS variable specifies additional architectures for each tune based on ARMv5 architecture.

In summary, the code sets up tuning configurations for the ARMv6 architecture, including defining characteristics, available tunes for little-endian and big-endian modes, specifying architecture, features, compiler arguments, and additional architectures.

The improvement in the response quality is clear - this is the kind of response we'd expect to get. It follows the format of examples provided in the training set quite closely.

// Summary

The intelligence of current models often appears illusory and has its limits or constraints; this is particularly the case when trying to utilise smaller, locally hosted models for performance reasons. A solution to this problem might lie in using smaller models that, through careful fine-tuning, may be well adapted to a specific task – a 'right tool for the job' approach. This contrasts with a versatile multi-tool that accomplishes many things but excels at none.

The emergence of various types of open-source models – not just StarCoder, but also models like Mixtral, Llama2, etc. - ushers in new, exciting possibilities. Being able to have the tool 'learn' the specifics of some relatively obscure technology, or work on some proprietary code that should never leave your network, can be very important in some situations. In a highly competitive business environment, the power to create unique, custom models offers a distinct advantage. Companies that solely depend on generic, one-size-fits-all tools may find themselves lagging.

Fine-tuning, as an approach to the broader problem of transfer learning, is considered by some as a key component in developing Machine Learning systems resembling human reasoning [13]. While our ambition may not reach that far, fine-tuning could nonetheless prove to be an excellent method for crafting practical development tools of the future.

Partner with Consult Red on your next project

We have over 20 years of experience in trusted innovation and embedded software development, supporting a global customer base.

Contact our expert team for support with your next development project, or the adoption of AI-enabled tooling in your organisation.

[13] see <https://www.ted.com/podcasts/ted-interview/deepminds-demis-hassabis-on-the-future-of-ai-transcript> for some interesting discussion regarding the topic.