The Ultimate Guide

for Computer Vision Deployment on NVIDIA® Jetson™

AUTHORS

Kate Yurkova
Deep Learning Research Engineer, Deci

Avi Lumelsky
Software Architect, Deci

Shai Rozenberg
Customer Facing Deep Learning Group Manager, Deci

Nave Assaf
Inference Team Lead, Deci

Lucy Kadets
Product Manager, Deci

Ran Rubin
DevOps Engineer, Deci

EXECUTIVE SUMMARY

Deploying deep learning models at the edge can be daunting. Each choice of device comes with its own pros and cons. Ultimately, you'll need to build a model that can support the level of complexity your application requires, while taking into account the hardware's specifications and available compute power.

In recent years we've been witnessing the introduction of more and more powerful devices with small form-factors designed for low-power operation. Among such devices is the NVIDIA® Jetson™ family. The NVIDIA Jetson family is designed to accelerate deep learning models for a variety of applications including robotics, drones, IoT devices, autonomous cars, and more.

What makes NVIDIA Jetson ideal for edge computing is the fact that it's a small but powerful computer that can run multiple neural networks in parallel. This is perfect for tasks such as image recognition, especially classification, object detection, and segmentation, as well as speech processing, speech recognition, and many others. That said, there are some challenges due to the restrictions of its constrained environment.

If you're working with, or considering working with Jetson, this guide is the perfect place to gain expertise.

The material gathered here is written from a deep learning inference perspective. Our goal is to guide you on your journey through Jetson, starting from the initial decision to use it as your deployment hardware, ending with those nuances so crucial to deployment, and including tips and tricks for all the stages in between.

For each recommendation provided in this guide, you'll learn how to do it yourself, where to find the code, what tools you need, and where to go for even more detail.

You'll also learn about the Deci Platform, which will save you loads of time and frustration. We explain how you can take advantage of neural architecture search to guarantee the best results possible for your application and hardware – all while maintaining accuracy.

Go ahead, read on, and let us know if you've got any ideas or questions.

TABLE OF CONTENTS

WHAT YOU NEED TO KNOW ABOUT JETSON HARDWARE

The NVIDIA Jetson family is a great choice when it comes to finding the optimal edge hardware in terms of computing power, price, and physical weight. The platform is presented as a family of boards, such as Nano, TX2, Xavier NX, AGX Xavier, and Orin. You can buy a Jetson Nano for as little as $60, its weight is a mere 250 g (~9 ounces), and it can run on as little as 5 watts.

As for the supported software stack, an onboard NVIDIA GPU makes deployment easier by opening access to a CUDA backend, something that many machine learning practitioners are already familiar with. It also includes the TensorRT graph compiler, built to speed up your inference.

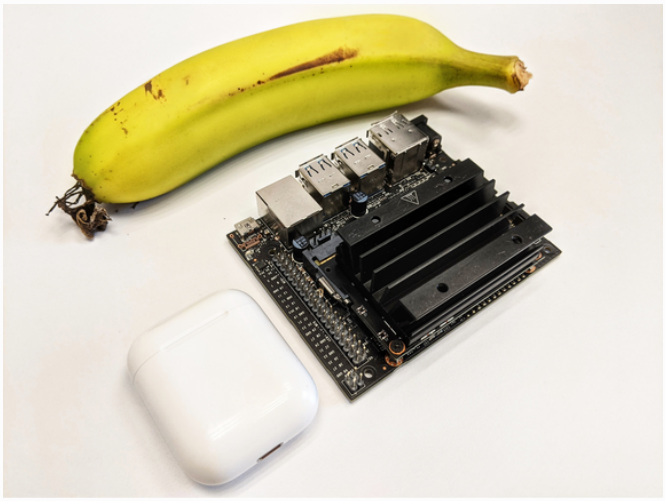


Figure 1: A Jetson Nano with a size reference (Image source: Deci)

The Board Family

The Jetson platform has high-performance edge computers named modules, or boards. There are a few of them available on the market with different capacities. If you're thinking that Jetson might be approximately what you need, then it's time to choose a specific board. Table 1 below presents the major configurations available.

For each one, you can see its GPU memory, GPU architecture, and the quantization levels supported for the architecture. The power budgets are also included, and the range shows all the options available within an individual device. For more information on these power options please refer to "Maximizing Compute Power with the Right Configuration" on page 11.

Table 1 should help you understand which option is right for you, or at least help you rule out a few configurations. If you already have, or are planning on using, a light-weight model, then consider the options in the upper rows. Likewise, if your application requires a larger model or a few concurrent workloads, then opt for a more powerful device.

Table 1: Jetson Hardware Configurations

Board	Memory	GPU Architecture	Supported Quantization Levels	Power
Nano	4 GB	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores	FP32, FP16	5W | 10W
TX2 NX	4 GB	NVIDIA Pascal™ architecture GPU with 256 NVIDIA CUDA® cores	FP32, FP16	7.5W | 15W
TX2	8 GB	256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA® cores	FP32, FP16	7.5W | 15W
TX2 NX	4 GB	NVIDIA Pascal™ architecture GPU with 256 NVIDIA CUDA® cores	FP32, FP16	7.5W | 15W
Xavier NX	8 GB / 16 GB	384-core NVIDIA Volta™ GPU with 48 Tensor Cores	FP32, FP16, INT8	10W | 15W | 20W
AGX Xavier	32 GB / 64 GB	512-core Volta™ GPU with 64 Tensor Cores	FP32, FP16, INT8	10W | 15W | 30W
AGX Xavier Industrial	32 GB	512-core Volta™ GPU with 64 Tensor Cores	FP32, FP16, INT8	20W | 40W
AGX Orin	32 GB / 64 GB	NVIDIA Ampere, 1792 / 2048 NVIDIA CUDA® cores and 56 / 64 Tensor Cores	FP32, FP16, INT8	15W | 40W | 60W

NVIDIA Jetson Developer Kit or Plain Jetson Module?

One more thing you need to consider before making a purchase is the contents of the box you get and your future work setup. This means understanding whether you should be working on the NVIDIA Jetson developer kit or getting the plain Jetson module. While the latter can be used for production, it needs some additional work like finding a carrier board to mount it on, and flashing the NVIDIA JetPack SDK. Here's the difference between the two:

Each Jetson developer kit includes a 'non-production specification' Jetson module attached to a reference carrier board. The carrier board has the ports needed for day-to-day work, such as an HDMI port to connect to a display, Ethernet port to connect to the Internet, USB ports for a mouse and a keyboard, etc.

You can use the developer kit together with the JetPack SDK to develop and test software for your use case in a pre-production environment.

Once you're ready to move to production, you'll have to switch to a production-grade Jetson module and carrier board. This comes without pre-installed software. You'll need to attach it to a carrier board and flash it with the JetPack SDK.

Read more about this here.

Major Board Components

Since we are already talking about the different device configurations, it's worth taking a peek at the internals of the Jetson module. For the most part, it won't impact your configuration choice but it will offer background for our later discussion on "Maximizing Compute Power with the Right Configuration" (see page 11).

All Jetson boards feature processors that belong to the Tegra series, which integrates the following components into one chip:

- ARM architecture CPU
- GPU
- Northbridge and southbridge, known as a chipset in modern motherboards
- Deep Learning Accelerator (DLA), available only for the Xavier module and Orin module

GETTING STARTED WITH INSTALLATION AND SETUP

The way you install and set up your device depends on your needs. If you are using a Jetson module in production, you most likely have unique hardware on which you'll mount your module; therefore, installing the JetPack is also unique to your hardware.

As described on the NVIDIA site, the JetPack SDK includes "the Jetson Linux Driver Package with bootloader, Linux kernel, Ubuntu desktop environment, and a complete set of libraries for acceleration of GPU computing, multimedia, graphics, and computer vision. It also includes samples, documentation, and developer tools for both host computer and developer kit, and supports higher level SDKs such as DeepStream for streaming video analytics, Isaac for robotics, and Riva for conversational AI."

Figure 2: Plugged in Jetson with peripheral setup (Image source: Deci)

If you are using the Jetson module for experimentation and development, you can find many suppliers who manufacture a suitable carrier board that can facilitate your Jetson module, for example, Seeed, AverMedia, and of course, NVIDIA's Developer Kit.

With so many options, we're going to focus on the NVIDIA Developer Kit. Fortunately, there are NVIDIA quickstart guides for the Jetson Nano Developer Kit, Jetson Xavier NX Developer Kit, and Jetson Orin Developer Kit. Their tutorials will walk you through setting up your developer kit using a MicroSD card.

If you have a Jetson Xavier NX developer kit, you can also mount an SSD card and run from it. You can follow this tutorial by JetsonHacks.

What Makes Deep Learning at the Edge Challenging?

Deep learning can be challenging under the best of circumstances, but deep learning on edge devices carries extra complexity because of the constrained environment. Edge hardware devices simply don't have the flexibility you get in cloud machines when it comes to OS, drivers, compute resources, memory, testing, and tuning. Failing to adapt to this environment can lead to delays in deployment. Hence, it's important to be aware of the specific requirements as well as potential challenges that can arise.

Let's familiarize ourselves with Jetson's specifications and requirements.

Jetson's ARM Architecture

The ARM architecture used in Jetson requires the software to be compiled specifically for ARM. This means you can't install or connect to your favorite development tools with a single command. If you want to install any open-source software, you have the option to either build the ARM version yourself from the source code or rely on the supplier to release the build.

Most software providers share their installation guidelines for Linux distributions, Windows, and MacOS. However, they don't usually share installation guidelines for Jetson with its ARM architecture. The same problem occurs when installing many other tools, which means that Jetson developers end up searching through forums for tutorials on how to get it done. This applies to:

- Code editors and IDEs like PyCharm, VSCode, and so on. When you do manage to install it and open a window, it consumes loads of resources and ends up restricting the device's performance.
- Deep learning libraries - PyTorch, TensorFlow, and others.

How can you work more smoothly in this environment? First of all, unless necessary, don't try to override libraries that come with a JetPack. Instead, use as much as you can from what is offered inside.

If you're not using the latest version, you'll probably benefit from an upgrade, since each version comes with a handful of improvements. You can check for the latest release on the official JetPack SDK page. And, in case you forgot which version you installed on your device, use the following command:

```
$ jetson_release
```

Python Libraries

If you're a Python developer, then you are certainly used to the usual "pip install $library" command. But on Jetson, many libraries can't be installed this way; you have to download and install a wheel instead.

Python 3.6 is the only version of Python available in the latest JetPack 4.6, even though it reached its end-of-life in December 2021. You can build a newer Python version on your own, but it's not guaranteed to work.

As for the basic Python libraries, doing a regular pip install won't hurt, so try that first. If that fails, you can probably find an appropriate wheel in the Jetson Zoo or in the Jetson Nano Wheels page on GitHub (if your module is Nano).

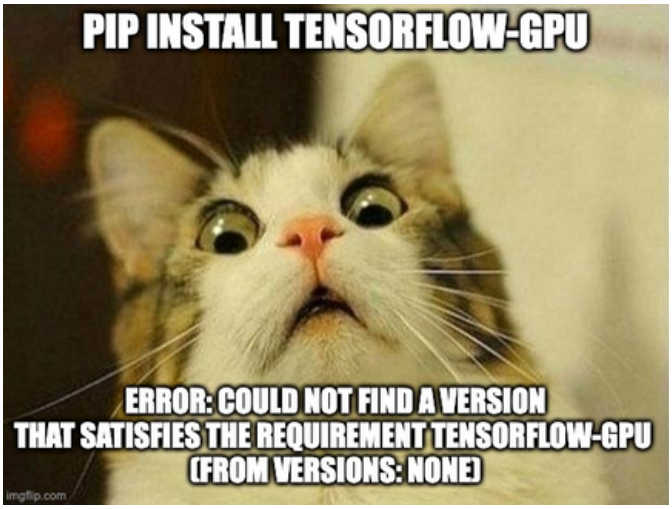


Figure 3: One does not simply install TensorFlow from pip on a Jetson, (Image source: Deci)

Updating Libraries

Updating major NVIDIA libraries like CUDA to a newer version, requires installing a whole new JetPack that will define the versions of the following and more:

- CUDA Driver
- CUDA toolkit
- TensorRT
- Python

In practice, a solid working environment can be hard to create, often making an already complex project even more complicated. Pre-built containers can be a simple solution. Check out the NVIDIA containers catalog and filter the catalog using the tag "Jetson" to see what's available for your use case.

TensorRT™ Checkpoints

TensorRT checkpoints can only be loaded on the same environment as the one in which they were compiled. Because TensorRT is also coupled with the JetPack, make sure you're optimizing the model's runtime for the JetPack version you intend to use in production. You can see more information in the NVIDIA developer's forum for Jetson.

Writing and Executing Code

When it comes to writing and executing code, if you are comfortable with command line text editors like vim or nano, you shouldn't have any problem using them on Jetson. But if you prefer advanced IDEs, this will require a bit more effort. Here are a few options:

- **Look for installation guides on forums and install an editor directly to the device.** This option will allow you to work as if it's just another computer. With a display, a mouse, and a keyboard plugged in, you can edit and execute your code directly. Keep in mind that an IDE will share the compute power with your application during tests.

  Let's be honest, can you really develop software without Googling a few issues? It's likely that at some point you'll want to open an Internet browser and it's going to become yet another draining workload. Then, you may find yourself jumping in between computers, developing on a Jetson, and Googling elsewhere, which, as you can imagine, isn't the most convenient experience.

- **Enable remote access to your Jetson (e.g. through SSH) and use it from your computer.** Here, all the editing is done in your regular environment: your laptop or PC will handle an IDE and a browser. The trick is to set up a deployment configuration and a remote interpreter/compiler in the code editor.

  This will allow you to continuously update the code on Jetson (to deploy it) and execute it directly on the device using the Jetson environment. As it happens, this feature is available in the Professional Edition of PyCharm or in VS Code.

MAXIMIZING COMPUTE POWER WITH THE RIGHT CONFIGURATION

Remember the internal module's parts listed in "Major Board Components" on page 6? These and other board components don't perform at the same level all the time.

Take a car for example; it would be wasteful to leave the engine running unless you are going to drive it. The situation with the Jetson board's components is similar - they don't work at full-power all the time. Instead, they support ranges of frequencies and states that are scaled dynamically and automatically by the system.

This is an essential process for power management, thermal management, and electrical management. In addition, not all CPU cores are always online and visible to the OS. If you have a fan installed on your module, it also operates in a mode of which you aren't necessarily aware. The mode is either "quiet" or "cool" and remains fixed until you change it.

Your user experience and your application's speed will be substantially impacted by the way frequencies are scaled, which cores are online, and other configurations.

Although there are many system governors that can handle everything in an optimal and steady way, you'll want to take charge so you can get your app up and running as fast as possible. It's the same basic reason why sports cars use a manual gear. If you understand these details and the different Jetson performance modes, you can unlock the full potential of your device.

When used correctly, these settings can boost your deep learning model's inference speed by as much as 8X compared to your current runs.

In this section, you'll learn how to correctly tune your Jetson hardware with minimal effort. You'll see a few examples that demonstrate the impact of the frequency scaling on the performance it will yield. We'll also highlight some caveats to keep in mind when applying new settings.

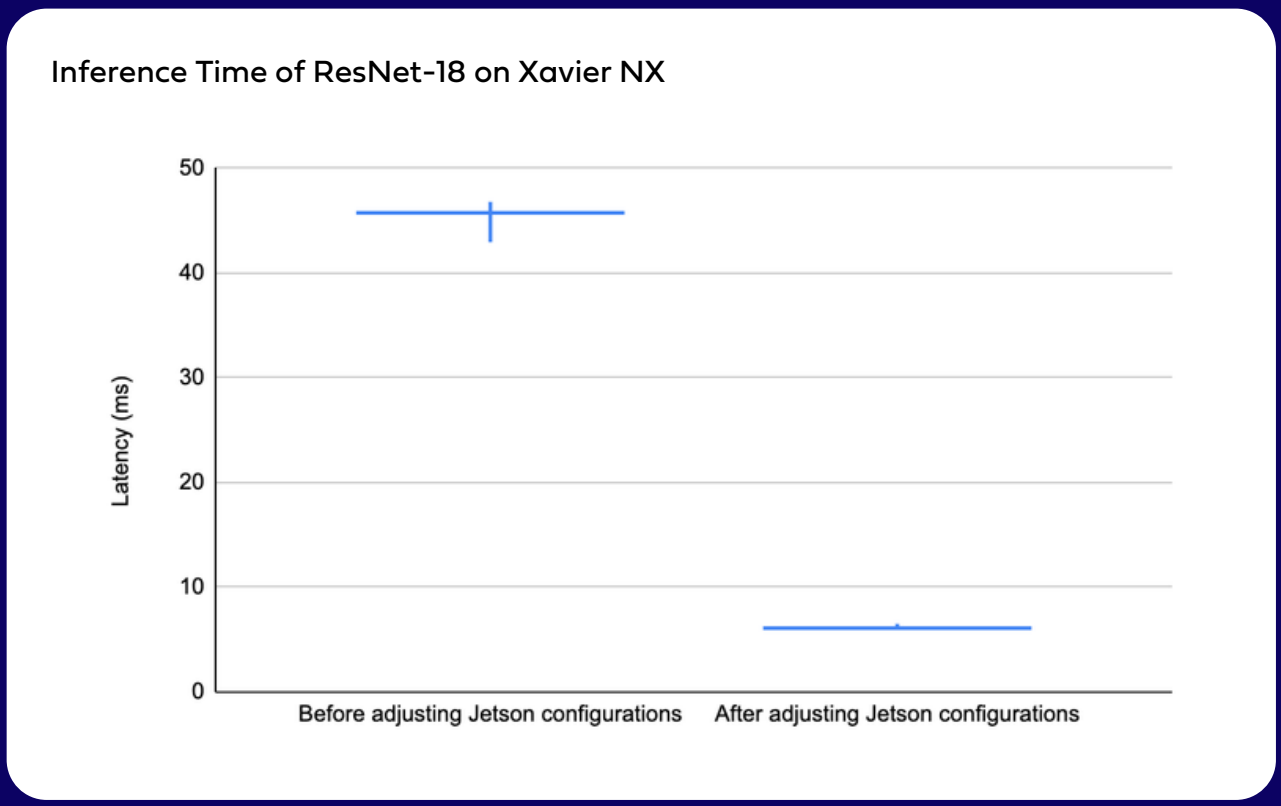


Figure 4: Changes in inference time of ResNet-18 on Xavier NX - before and after adjusting configurations: Resolution: 224, Batch size: 1. Series of predictions done with a 10-second interval in two different states of Jetson Xavier NX. The candlestick chart depicts min and max latencies with a vertical range and an average latency with a horizontal line.

Tuning for Performance

If this is your first time working with Jetson, you might be reluctant to jump in and adjust frequencies on your own. The good news is that you don't have to. Each Jetson device comes with a few optimized power budgets (e.g., for 5W, 10W, etc.) as well as a command line tool called nvpmodel, which you can use to set the performance and energy usage characteristics. This tool offers pre-defined power modes for each energy budget and is very easy to use.

You can select a mode or even create a custom one, depending on the expected workload of your application, the desired power consumption, energy source, etc.

To maximize performance using power and energy settings, begin by running:

```
$ sudo /usr/sbin/nvpmodel -q
```

to check the current settings on your device. You'll see the names of the selected power mode, fan mode, and so on. For example, here is a default output on Jetson Xavier NX:

```
NV Fan Mode:quiet
NV Power Mode: MODE_10W_2CORE
3
```

MODE_10W_2CORE doesn't tell us everything. To learn what it incorporates either add **`--verbose`** to the command above or look for your Jetson series in the Clock Frequency and Power Management Documentation. All the modes are defined in /etc/nvpmodel.conf. Once you understand what your options are, try setting a different configuration from what you currently have using the following:

```
$ sudo /usr/sbin/nvpmodel -m $ID
```

where ID is an index of the mode you want to select. The maximum one is usually defined with the ID 0. You can change the fan mode as well using:

```
$ sudo /usr/sbin/nvpmodel -d $FAN_MODE
```

where FAN_MODE is either cool or quiet.

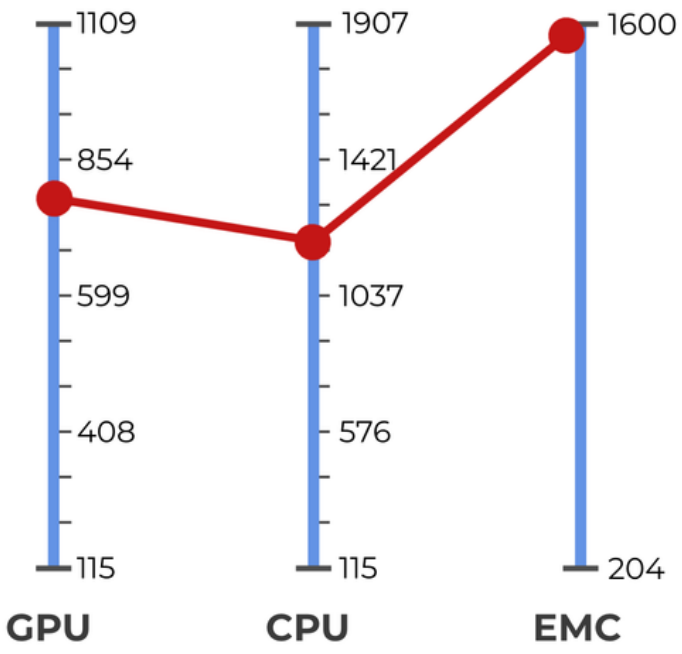


Figure 5: Xavier NX - the supported and selected maximum frequencies for GPU, CPU, and EMC. The main takeaway is this: no matter which frequencies a device supports, you can limit the maximums to lower values

The Jetson Board Support Package provides the /usr/bin/jetson_clocks.sh script, which sets the CPU, GPU, and EMC clocks to their maximum values. You can also use the **`--fan`** flag to make the fan work at maximum speed. Run it with sudo as follows:

```
$ sudo /usr/bin/jetson_clocks --fan
```

Running this script is different from simply choosing specific settings with nvpmodel. Each power mode set with **`nvpmodel`** defines a range of frequencies, and the current ones will be raised or lowered by the system within these ranges, depending on the incoming workload.

You can check these values by adding the **`--show`** flag to the command above. The big difference is that jetson_clocks.sh overrides this behavior and ensures that your application uses maximum frequencies for the given mode at all times.

Remember, when you opt for the highest performance for your runs, you make your device work harder. This will lead to increased power consumption and will raise the temperature of your device.

To prevent throttling, keep an eye on the degrees in the output of the **`tegrastats`** command line utility. The throttling temperatures vary from module to module and are available in the Power Management documentation under the Thermal Specifications section. Here is a sample of the tegrastats output:

```
 $ tegrastats
 RAM 738/3964MB (lfb 4x2MB) SWAP 253/4096MB (cached 7MB) CPU
[60%@921,52%@921,off,off] EMC_FREQ 0% GR3D_FREQ 0% PLL@38C CPU@41C
PMIC@100C GPU@39C AO@45.5C thermal@40C
```

In addition to the official utilities discussed here, there are also many unofficial tools (e.g. jetson_stats) that allow you to monitor the device state directly from Python.

Why is this important? Let's take a look at the latency that can be achieved on the Jetson Xavier NX in two power modes using different architectures compiled for TensorRT in FP32. If you are not familiar with TensorRT or quantization levels, don't worry, it's orthogonal to the experiments below. You'll learn more about them later on in "Choosing a Model for Jetson" (See page 18).

We're going to compare the default (MODE_10W_2CORE) and the maximum performance (MODE_15W_2CORE) modes. Both graphs are based on a resolution of 224 and a batch size of 1, and show how the latency changed after each inference. The x axis contains the latency of the x-th consequent inference.

ResNet-18

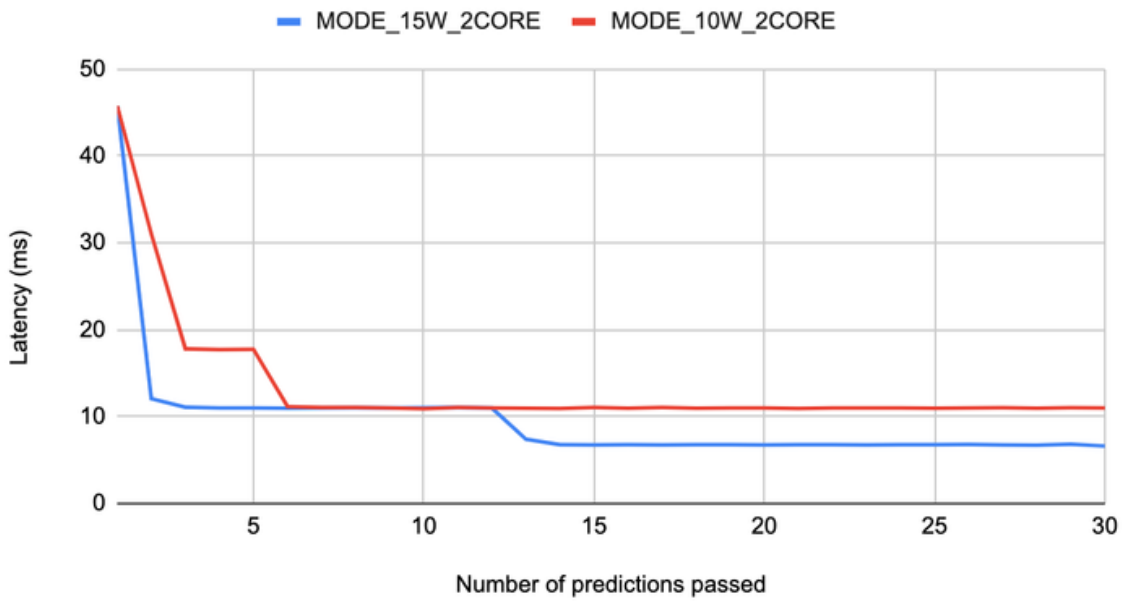


Figure 6: Latency for ResNet-18 compiled for TensorRT on Jetson Xavier NX in 2 power modes. The default is shown in red.

DenseNet-121

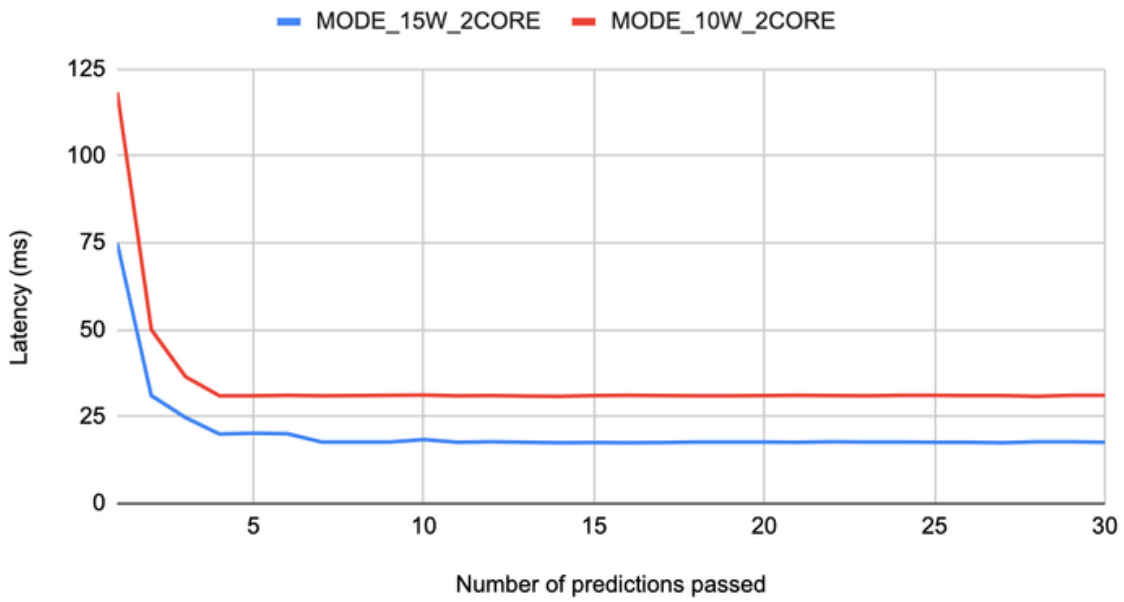


Figure 7: Latency for DenseNet-121 compiled for TensorRT on Jetson Xavier NX in 2 power modes. The default is shown in red.

The important insights from these graphs are:

1. It takes a few back-to-back iterations to warm up the GPU before the latency settles on a certain number. Your app will start slow and remain slow if you don't utilize the GPU consistently. Consistency here refers to some back-to-back workload that would prevent a GPU from cooling down in the interim, and this is oftentimes defined by your use case.

2. Even after the warmup is complete, we can see that the default mode is significantly slower than the maximum one.

To address the first issue, you can reduce the latency variance using jetson_clocks. Take a look at Figures 8 and 9: we set up the benchmarking exactly as we did for Figure 6 and 7 but first ran **`jetson_clocks`** .

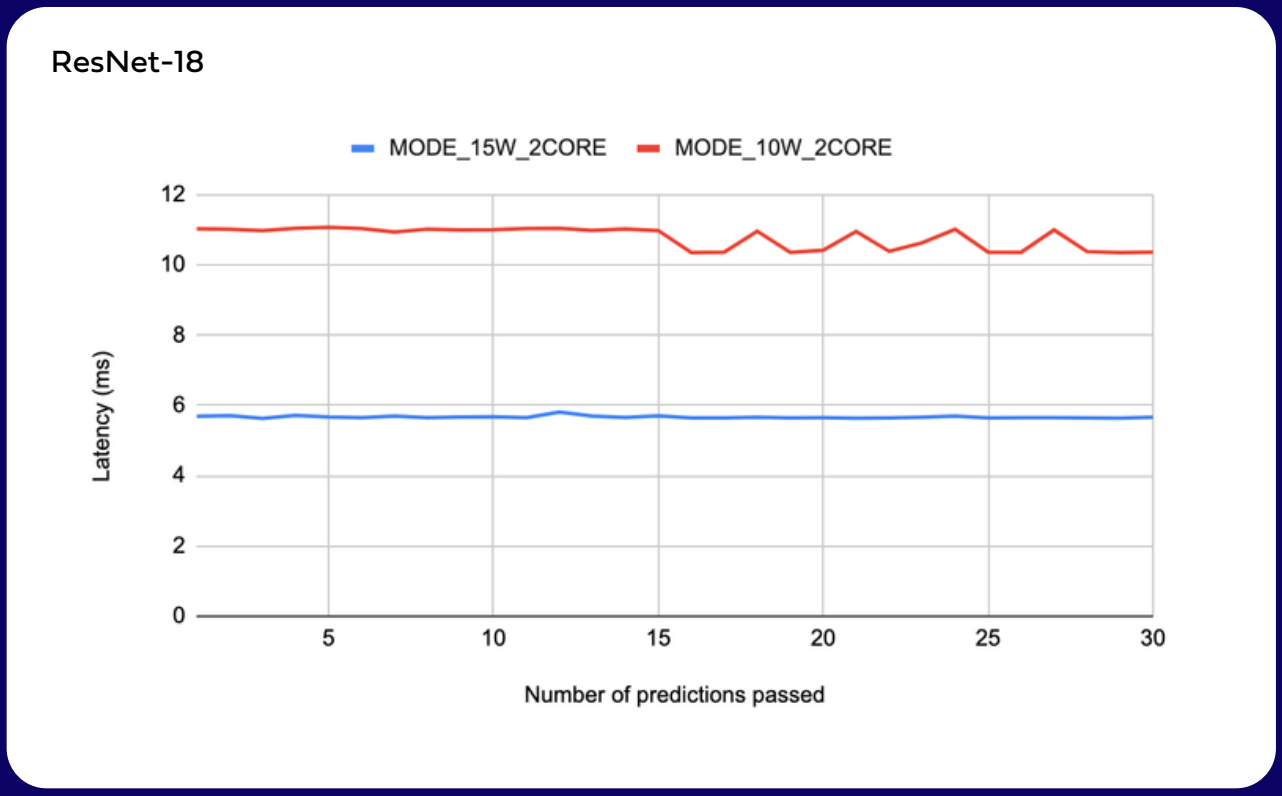


Figure 8: ResNet-18 latency after tuning clock and power parameters. The default is shown in red.

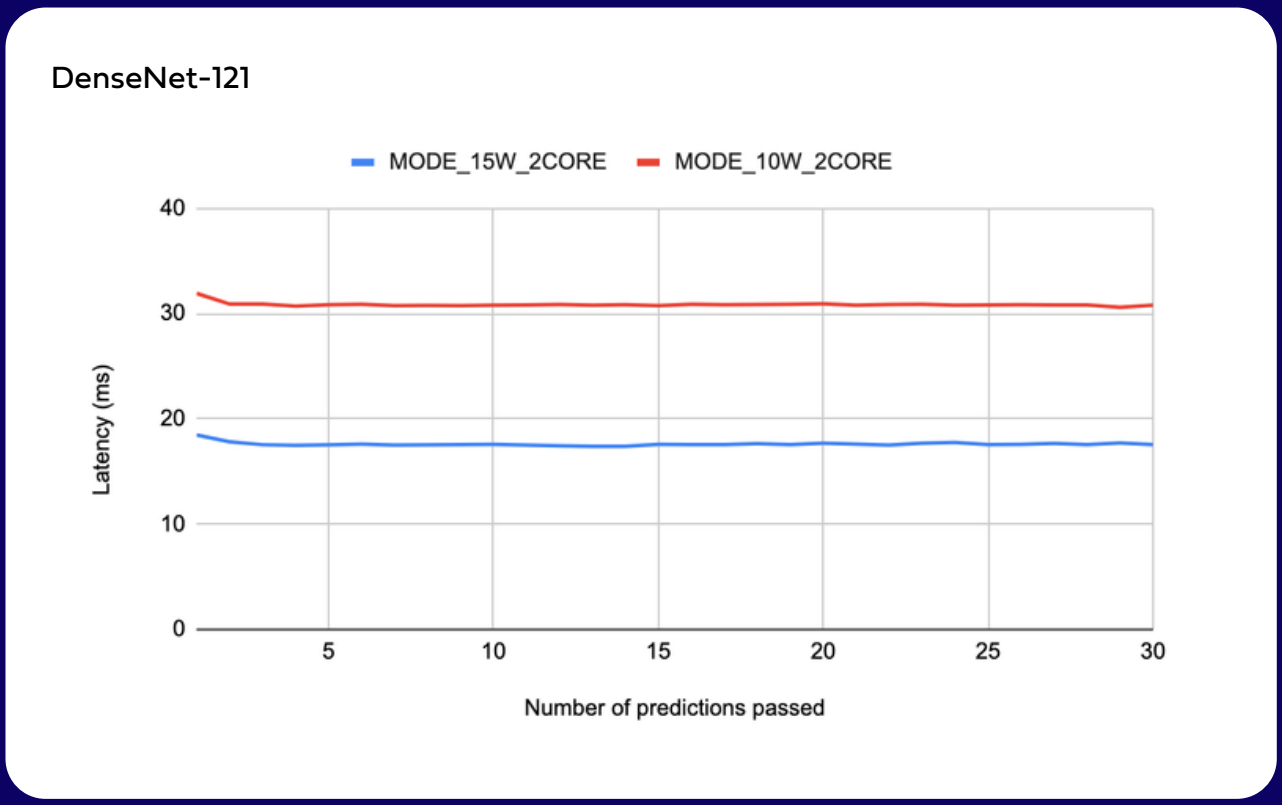


Figure 9: DenseNet-121 latency after tuning clock and power parameters. The default is shown in red.

To recap, in this section you became familiar with the Jetson architecture and learned which settings can dramatically impact its performance. You also learned how to change and monitor your module's state using the following Jetson utilities:

- nvpmodel
- tegrastats
- jetson_clocks

It's definitely worth playing with these settings and customizing them to get the best performance out of your Jetson hardware.

CHOOSING A MODEL FOR YOUR JETSON DEVICE

With the hardware and development setup ready, you can finally dive into what is arguably the most interesting task: selecting a model.

If your company is developing AI-infused applications, you likely have a model in mind, perhaps an off-the-shelf solution for your use case or a proprietary design. You should ask yourself before using this model if you checked its latency and throughput, or compared it with other similar architectures. Even if you have done so, this step must be done on the target inference hardware, in this case, your Jetson.

In deep learning, a form of AI that is modeled after the network architecture of the human brain, there's no such thing as "one-model-fits-all." Different use cases, deployment constraints, and even Jetson modules, all call for different architectures.

Of course, you may also arrive at this stage without a model in mind. For those of you who are just starting, we've got you covered. Model selection is discussed in detail in this section.

No matter where you are in your project, once the hardware arrives, the benchmarking stage begins. There is no need to train a model prior to checking its runtime, so this opens up the opportunity to test many options early on. Remember, to get the most accurate measurements, benchmarking must be performed with a sufficient understanding of the planned deployment configuration and as close to production conditions as possible.

In addition to setting a power mode, you need to select which TensortRT you want to use in advance and decide on the JetPack version accordingly. On Jetson, TensorRT is responsible for the two cornerstones of runtime optimization: compilation and quantization. Let's make sure these concepts are clear, so you have a good base for understanding the rest of this ebook.

Compilation

Model compilation is a technique that provides acceleration almost for free and with no impact on accuracy. Compilers work by evaluating and fixing the computation graph of a given model. The graph turns out to be incredibly useful, because it allows the compiler to generate optimized inference code.

Not only is it adjusted for your hardware, but it also fuses together some sequences of layers. Memory allocation and computation can be done more efficiently when you know for certain which operation comes next.

Both the Jetson platform and TensorRT are NVIDIA products, so it's no surprise that they work well together. Of all the options for migrating your deep learning framework to TensorRT, the easiest one is through ONNX, an open format built to represent machine learning models. In practice, you need to look for a way to export your model as an ONNX file, then convert this file into a TRT engine, for example using the NVIDIA trtexec tool.

As you can imagine, the abundance of layers can complicate the process. While most layers in your deep learning network are probably plain old convolutions, fully connected layers, or ReLU activations, you might also encounter newer ones, like activation functions from very recent academic publications.

In short, choose your layers wisely. The newer the layer, the more likely that it's not yet supported in either ONNX or the TRT itself. The good news is that you can also implement it yourself by decomposing a layer into its basic operations.

For example, in PyTorch instead of using **`torch.nn.SiLU`** you can interchangeably use the following class:

```
class SiLU(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)
```

You can also write your own plugin in C++ and CUDA, but this will require advanced engineering and algorithmic knowledge, and is easier said than done. Check out these docs to get a feel for this process.

The bottom line is that unless you understand why you need those particular layers in your network, you are better off replacing them with something more conventional. What's more, if you follow our advice and attempt the compilation prior to training, you'll discover potential conversion problems early on, before the time-consuming stage of training is done.

Quantization

Quantization is an optimization technique that defines and applies different bit-widths individually to each hidden layer of the neural network. Using quantization will produce a network that is smaller in terms of a checkpoint weight and faster when it comes to computation. On an edge device, this translates directly into energy efficiency of the inference.

While the default model precision for Jetson is FP32, the existing quantization levels are FP16 (also known as half-precision) and INT8. The available levels of quantization vary from Jetson to Jetson, as you might have noticed in "The Board Family" section (see page 4).

What does this really mean? A single weight (one value) in your network takes 32 bits to be stored. The same weight in the FP16 representation will take half as much memory (16 bits). Going further, an INT8 value occupies 8 bits and uses integer computation during inference instead of floating-point math.

But nothing is truly free: be aware that quantization can cause information loss and distort network representation. The lower your level of quantization, the bigger the distortion. In practice, you can convert your weights into FP16 without visible degradation of accuracy, but INT8 is much trickier.

If you're lucky, you'll experience a minor drop in accuracy; otherwise, the drop can be quite dramatic, making your weights unusable. Most of the time, quantization also requires a dataset calibration to be performed. So make sure to test all the metrics after your model undergoes quantization.

This behavior can be explained by two factors. First of all, the weights become quite different from what they converged to during training: different enough for error to accumulate throughout the layers. Second, different activation functions "survive" quantization differently.

For example, Alexander Finkelstein et al. in "A Quantization-Friendly Separable Convolution for MobileNets" investigated the reason for the performance degradation of MobileNet when it is quantized to 8-bits and claimed that the problem stems from the use of ReLU6. Although ReLU6 is known to reduce the quantization range, which should be a good thing, they explain that in MobileNet it clips the signal in the early layers, distorting the signal distribution and making it less quantization friendly.

Now that you've learned what TensoRT is and what quantization gives you, you can compile your model with the precision you want. As mentioned above, it can be done in two steps: PyTorch model to ONNX, ONNX to TRT engine. So, with a TRT engine at hand, let's proceed to benchmarking.

Know Your Application Flow

It's easy to forget that inference is not the only thing that takes time. Aside from inference, an application is composed of operations such as pre/post-processing, data loading, data copies, etc. These are all repetitive and take up time in addition to inference. So, when you benchmark, remember to test your applications end-to-end.

It may be that you have the most control over the model, but measuring the whole pipeline can help you discover where bottlenecks are hiding. Profiling the following metrics will help ensure an accurate benchmark time for real-world application flows:

- Read from disk / network
- Pre-processing
- Copy CPU → GPU
- Inference
- Copy GPU → CPU
- Post-processing
- Write results to disk / network
- Repeat

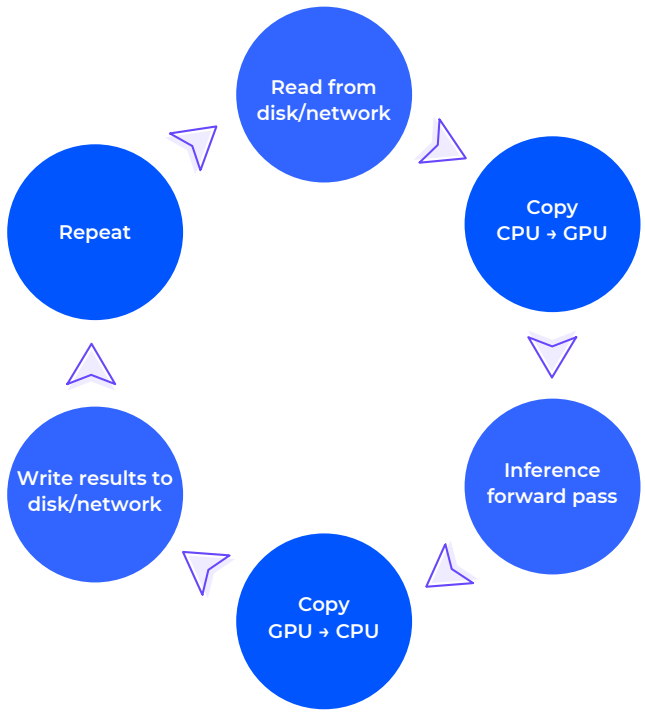


Figure 10: Application flow. It's also a good idea to use jtop to monitor the real-time status of your lifecycle efficiency during the application pipeline. It displays CPU, RAM, GPU status, frequency, and more with everything updated live.

Let's Benchmark

Benchmarking is harder than it sounds. It's full of subtle nuances that even experienced programmers forget about, and can lead to inaccurate latency measurements. Knowing this, we developed Infery, a free Python runtime engine that makes it simple to run inference and benchmark optimized models. It involves just three lines of code and supports major hardware types and frameworks, including:

- NVIDIA TensorRT
- TensorFlow
- Keras
- ONNX Runtime
- TorchScript

Considering the above, if you choose to use Infery for your TRT engines on Jetson, there's a bonus: an API that will help you benchmark in your future projects. The constant evolution of deep learning frameworks and libraries means developers often get bogged down trying to learn and integrate new code to do the same work on a new device.

What's more, error handling is not always user-friendly, leaving you in a situation where you don't really understand what you did wrong or how to fix your code and move on. Infery takes care of all this. It's all about being developer-friendly. This is accomplished thanks to its unified API, which is a combination of industry best practices, simplified to the max.

If you are not convinced yet, here are three more reasons to use it:

1. **You can benchmark parts of the application flow separately**

   Together with latency and throughput, the measurements you get from Infery include the timing of a copy from CPU to GPU and back.

2. **Infery manages dependencies for you**

   Installing several deep learning libraries or runtimes together often results in a struggle with broken environments. For example, to install the Torch, ONNX, and TensorFlow libraries together, you'd have to hunt for the correct version of a mutual dependency, like NumPy. This will enable them to coexist in the same environment, without breaking or affecting other installed packages.

   On Jetson this is an even bigger deal because you need to manually download the correct wheels. With Infery, it's just the opposite. Infery's versions are designed and tested to be installed successfully under multiple environments and edge cases. Whether it's an existing environment or a new one, Infery will make sure the installation is successful.

3. **It works seamlessly with the Deci platform**

   Infery is a stand-alone library, but when coupled with the Deci platform, you can optimize and deploy your models in a matter of minutes to boost inference performance on your preferred hardware, while maintaining the same accuracy. Once you optimize your model on the Deci platform, only three simple copy-pastes separate you from running local inference for your optimized model. Read on to learn more about the Deci platform.

For instructions on how to install Infery on your Jetson please refer to Infery's documentation page or the Deci Inference Examples repo on GitHub.

How to Use Infery in 3 Simple Steps

Here's the flow for working with Infery and what makes it so easy:

1 Load the model

First and foremost, the TRT engine that you created should stay in its environment.

If you created an engine on one Jetson model (e.g., Nano) and in one JetPack + TensorRT version, but then moved it elsewhere, it won't load because model compilation leverages the GPU architecture and takes versions of the CUDA-related libraries into account. Please keep this in mind. Even if you manage to load your checkpoint after the mentioned changes, it might be a good idea to recreate it.

Infery checks and validates the compatibility of the model. If there are errors, it helps you understand exactly what you have to do next to progress.

```
In [1]: import numpy as np
In [2]: import infery
Jetson stats: {
    "APE": 150,
    "CPU1": 28,
    "CPU2": 96,
    "CPU3": "OFF",
    "CPU4": "OFF",
    "CPU5": "OFF",
    "CPU6": "OFF",
    "EMC": 3543340,
    "GPU": 3,
    "MTS BG": 1,
    "MTS FG": 1,
    "NVDEC": "OFF",
    "NVENC": "OFF",
    "NVJPG": "OFF",
    "RAM": 3543340,
    "SWAP": 0,
    "Temp AO": 37.5,
    "Temp AUX": 37.5,
    "Temp CPU": 41.0,
```

```
    "Temp GPU": 38.5,
    "Temp thermal": 38.85,
    "fan": 100.0,
    "jetson_clocks": "ON",
    "nvp model": "MODE_15W_2CORE",
    "power avg": 5580,
    "power cur": 6511,
    "time": "2022-05-15 20:17:01.579317",
    "uptime": "3 days, 7:39:11.480000"
}
__init__ -INFO- Infery was successfully imported with 2 CPUS and 1 GPUS.
In [3]: model = infery.load('yoloxS_640_fp16.engine',
framework_type='trt')

infery_manager -INFO- Loading model yoloxS_640_fp16.engine to the GPU
```

As you can see, Infery also summarizes Jetson's state on import, which is extremely important to keep in mind during benchmarking.

2 Run inference

It's worth noting that Infery also lets you easily run inference in Python. It's called Infery for a reason, right? ☺

For example, say you have a YOLOX-s object detection model with an image that is already loaded in memory, and you want to run inference on this image. Run a few lines of code and you'll get the predicted bounding boxes. There's no need for any extra configuration file. Just remember to pre-process the image according to what your model expects, before calling **`model.predict()`**.

Just to clarify, we use YOLO here for demonstration purposes, but the task of the model doesn't really matter. Whether it's computer vision, natural language processing, or voice recognition, Infery wraps the model for seamless inference.

```
In [4]: inputs = np.random.random((1, 3, 640, 640)).astype('float16')

In [5]: model.predict(inputs)
Out[5]:
[array([[[-2.75634766e-01, 1.99829102e-01, 8.36910152e+00, ...,
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02],
 [ 7.72265625e+00, 1.99829102e-01, 8.36910152e+00, ...,
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02],
 [ 1.57265625e+01, 1.99829102e-01, 8.36910152e+00, ...,
```

```
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02],
 ...,
 [ 5.46500000e+02, 6.06500000e+02, 3.13676586e+01, ...,
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02],
 [ 5.78500000e+02, 6.06500000e+02, 3.13676586e+01, ...,
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02],
 [ 6.10500000e+02, 6.06500000e+02, 3.13676586e+01, ...,
 1.00097656e-02, 1.00097656e-02, 1.00097656e-02]]],
 dtype=float32)]
```

3 Benchmark the model

At last, back to benchmarking. All you have to do is enter a batch size and input dimensions. You'll see that the measurements change significantly as you modify these factors.

Remember, these should be the values you used during model compilation. For example, if you need to bump up the batch size to a bigger one or increase input resolution, you'll have to create a new engine specifically for this purpose. This is how TensorRT is designed.

```
In [6]: model.benchmark(batch_size=1, input_dims=(3, 640, 640))
base_inferencer -INFO- Benchmarking the model in batch size 1 and
dimensions (3, 640, 640)...
Out[6]:
<ModelBenchmarks: {
    "batch_size": 1,
    "batch_inf_time": "17.01 ms",
    "batch_inf_time_variance": "1.01 ms",
    "memory": "2.48 mb",
    "pre_inference_memory_used": "2.47 mb",
    "post_inference_memory_used": "2.48 mb",
    "total_memory_size": "15.45 mb",
    "throughput": "58.79 fps",
    "sample_inf_time": "17.01 ms",
    "include_io": true,
    "framework_type": "trt",
    "framework_version": "8.0.1.6",
    "inference_hardware": "GPU",
    "date": "20:17:42__05-15-2022",
    "ctime": 1652645862,
    "h_to_d_mean": "0.94 ms",
    "d_to_h_mean": "0.27 ms",
    "h_to_d_variance": "0.22 ms",
    "d_to_h_variance": "0.00 ms"
}>
```

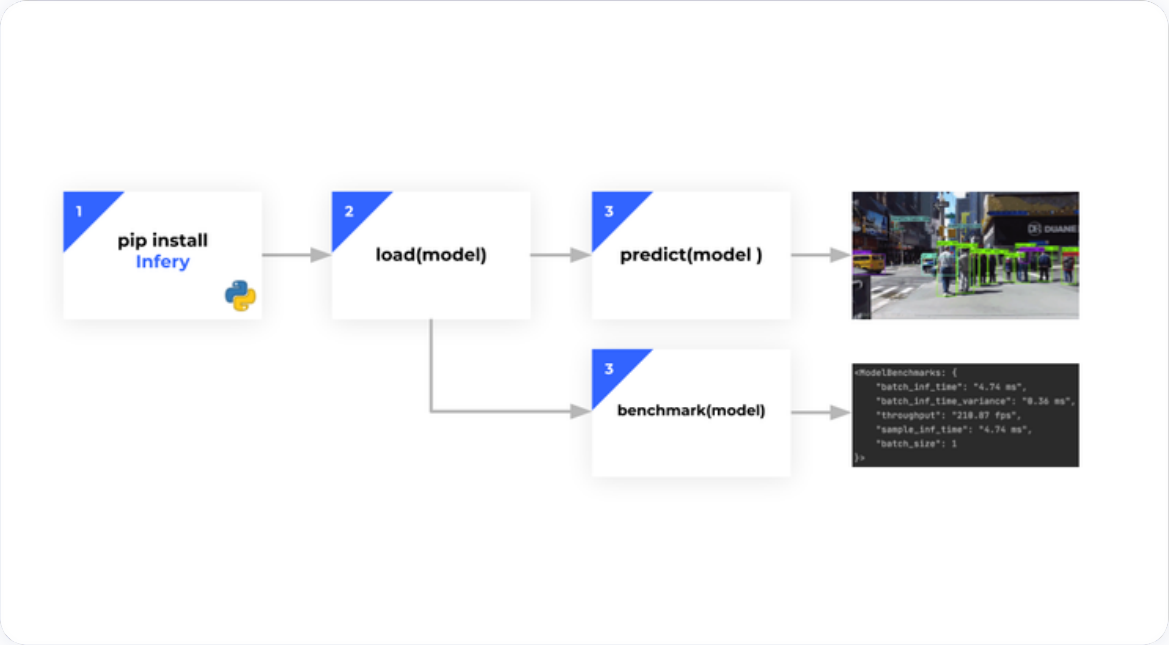


Figure 11: Infery Python runtime engine to run inference locally

Everything seems rather easy, but then, someone still has to install the package, create an engine, benchmark it in a few batch sizes, and more. Don't you wish someone (or something) would do it for you beforehand?

Deci Platform to the Rescue!

The Deci deep learning development platform includes various tools that help tackle the problems described above and to relieve the pains of taking models to production. The platform lets you optimize your deep learning models' accuracy and inference performance for any hardware by leveraging techniques such as graph compilation, quantization, and Neural Architecture Search (NAS), among others.

You can also use the Deci platform to compare your model's performance on different hardware and batch sizes. This can help you decide what is the best hardware for your use case. The platform also makes it easy to experiment with production settings. And, if you still don't have a model at hand, or are not sure what architecture to choose for your task, you can use Deci's NAS engine to build a custom model architecture that is tailored for your use case, hardware, and performance goals.

We know no one likes to read long manuals for simple things, so just go ahead and check out our platform for yourself. Below, for your convenience, you can find more details on the main tools the platform offers:

1. Benchmark your models on various hardware, hassle free with our online hardware fleet
2. Compile and quantize your models for the selected hardware with a few clicks
3. Build fast and accurate hardware aware model architectures with our NAS engine
4. Easily leverage advanced training techniques and SOTA recipes with one line of code
5. Deploy with 3 lines of code using Infery - Deci's Python Inference Runtime Engine

Benchmarking Your Model on Various Hardware

You can upload models for benchmarking and optimization. Your model will be automatically benchmarked on a variety of different hardware and batch sizes, with randomized tensors as inputs. You will then be able to see your model's performance (latency, throughput, model size, and memory footprint) on different hardware, and see which hardware and batch size works best with your model.

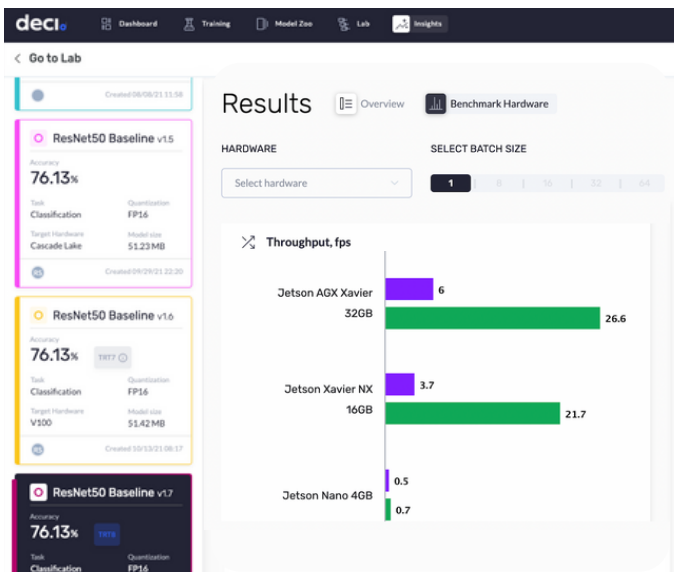


Figure 12: Deci platform - compare your models' performance across various hardware.

Optimizing your Model for Selected Hardware

With the Deci platform you can automatically quantize and compile your model for the production environment. If you're still thinking about what hardware to choose, you can always make multiple optimizations and compare them in the platform to choose the one with the best results.

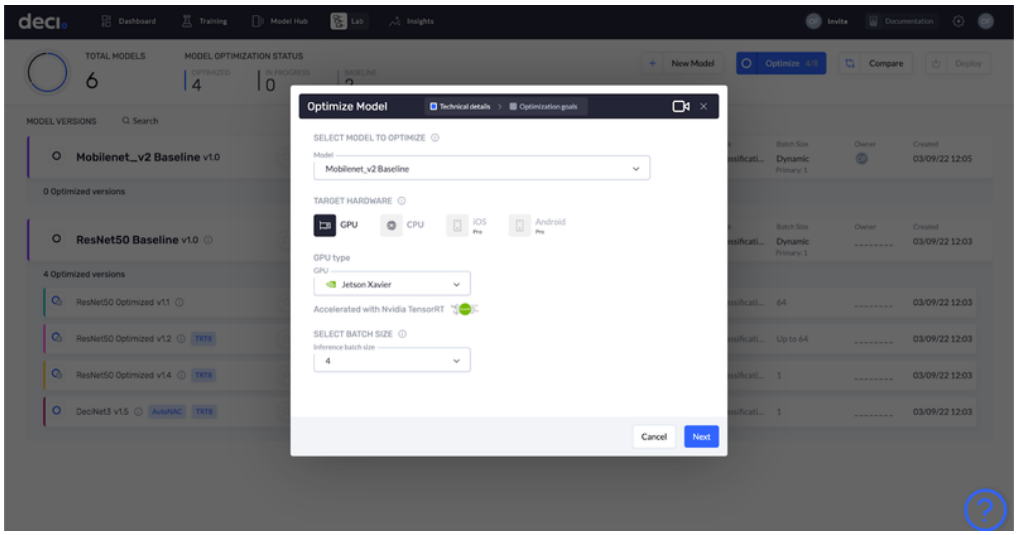


Figure 13: Deci platform - optimize your model to target hardware and batch size

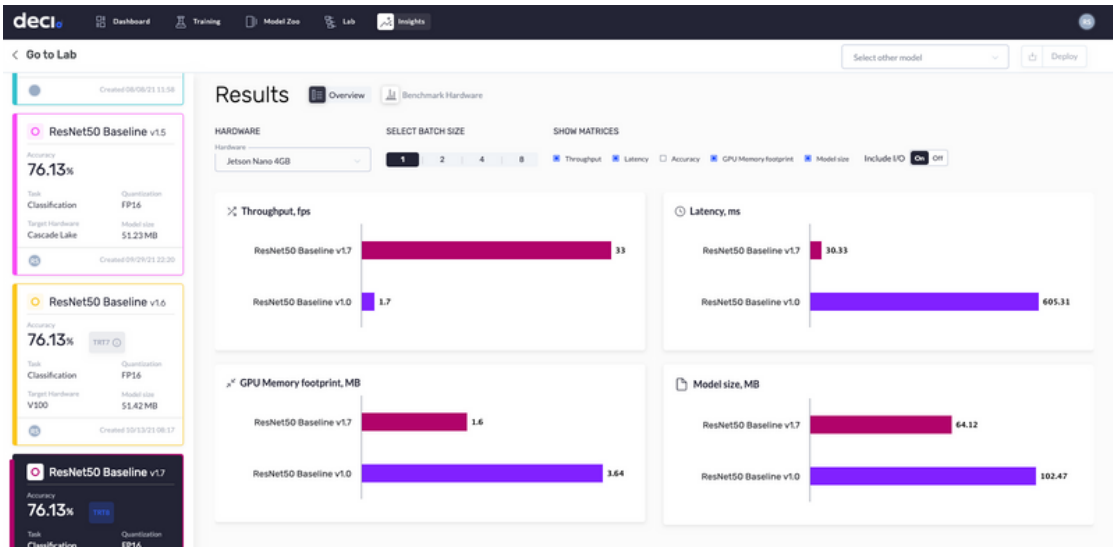


Figure 14: Deci platform - compare your model and its optimized versions

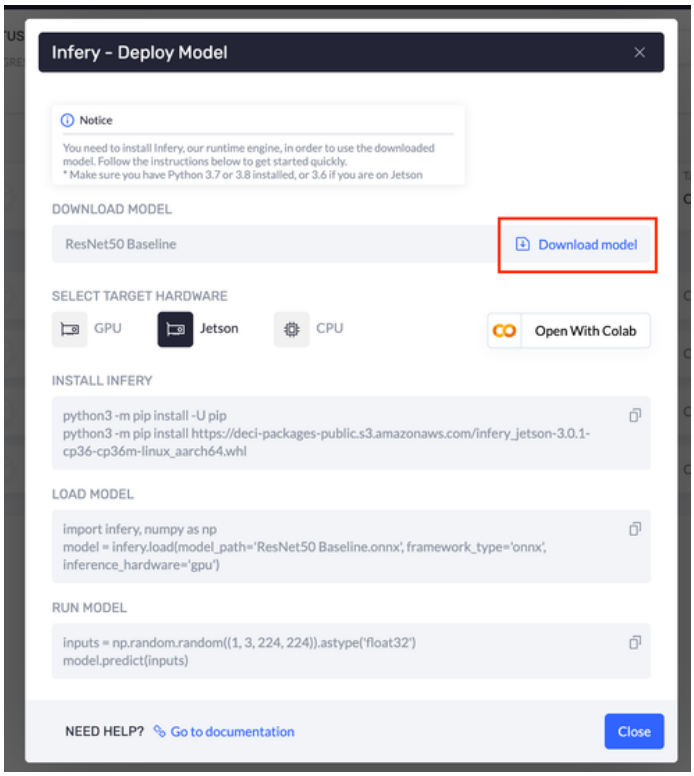


Figure 15: After optimizing your model with the Deci platform, simply download the optimized model file and with 3 copy-pastes you are ready to run inference with Infery

When the Hardware Surprises You

In your regular training environment, your model may have a similar runtime to its competitor architectures, while on Jetson it might become much slower. For example, look at the latencies (in milliseconds) of GoogLeNet V1 and ResNet34 compiled in TensorRT, FP16 compared on two GPUs: T4 and Jetson Nano.

Table 2: GoogLeNet V1 and resNet34 Latencies, ms

	T4	Jetson Nano
GoogLeNet V1	1 ms	13 ms
ResNet34	1 ms	24 ms

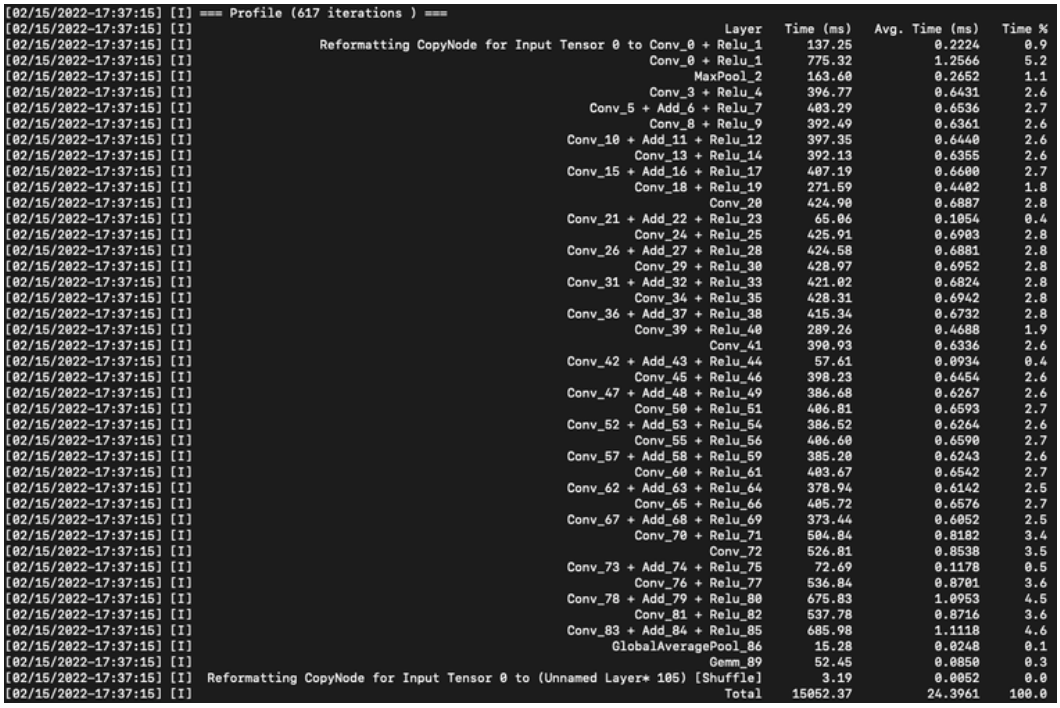
This and other similar situations are not uncommon. They prove that on each hardware, and even more so with each model compiler and quantization level, different layers will benefit from different rates of acceleration. Beyond being a surprise, it is also a challenge to predict the behavior in advance and identify its root cause. Instead of guessing and before digging into possible theoretical explanations, consider the debugging technique discussed in the next section.

Profile Your Network

Go ahead and profile how much time each layer takes. With the trtexec mentioned earlier, you can see the average time it takes to execute each layer's output: the absolute value as well as the percentage of the total.

To get these measurements, use the **`--dumpProfile`** flag. Here is the command:

```
/usr/src/tensorrt/bin/trtexec --
loadEngine=/path/to/you/trt_model.engine --dumpProfile
```



```
[02/15/2022-17:37:15] [I] === Profile (617 iterations ) ===
[02/15/2022-17:37:15] [I]                                                 Layer   Time (ms)   Avg. Time (ms)   Time %
[02/15/2022-17:37:15] [I]               Reformatting CopyNode for Input Tensor 0 to Conv_0 + Relu_1      137.25           0.2224      0.9
[02/15/2022-17:37:15] [I]                                         Conv_0 + Relu_1      775.32           1.2566      5.2
[02/15/2022-17:37:15] [I]                                              MaxPool_2      163.60           0.2652      1.1
[02/15/2022-17:37:15] [I]                                         Conv_3 + Relu_4      396.77           0.6431      2.6
[02/15/2022-17:37:15] [I]                                 Conv_5 + Add_6 + Relu_7      403.29           0.6536      2.7
[02/15/2022-17:37:15] [I]                                         Conv_8 + Relu_9      392.49           0.6361      2.6
[02/15/2022-17:37:15] [I]                                Conv_10 + Add_11 + Relu_12      397.35           0.6440      2.6
[02/15/2022-17:37:15] [I]                                        Conv_13 + Relu_14      392.13           0.6355      2.6
[02/15/2022-17:37:15] [I]                                Conv_15 + Add_16 + Relu_17      407.19           0.6600      2.7
[02/15/2022-17:37:15] [I]                                        Conv_18 + Relu_19      271.59           0.4402      1.8
[02/15/2022-17:37:15] [I]                                              Conv_20      424.90           0.6887      2.8
[02/15/2022-17:37:15] [I]                                Conv_21 + Add_22 + Relu_23       65.06           0.1054      0.4
[02/15/2022-17:37:15] [I]                                        Conv_24 + Relu_25      425.91           0.6903      2.8
[02/15/2022-17:37:15] [I]                                Conv_26 + Add_27 + Relu_28      424.58           0.6881      2.8
[02/15/2022-17:37:15] [I]                                        Conv_29 + Relu_30      428.97           0.6952      2.8
[02/15/2022-17:37:15] [I]                                Conv_31 + Add_32 + Relu_33      421.02           0.6824      2.8
[02/15/2022-17:37:15] [I]                                        Conv_34 + Relu_35      428.31           0.6942      2.8
[02/15/2022-17:37:15] [I]                                Conv_36 + Add_37 + Relu_38      415.34           0.6732      2.8
[02/15/2022-17:37:15] [I]                                        Conv_39 + Relu_40      289.26           0.4688      1.9
[02/15/2022-17:37:15] [I]                                              Conv_41      390.93           0.6336      2.6
[02/15/2022-17:37:15] [I]                                Conv_42 + Add_43 + Relu_44       57.61           0.0934      0.4
[02/15/2022-17:37:15] [I]                                        Conv_45 + Relu_46      398.23           0.6454      2.6
[02/15/2022-17:37:15] [I]                                Conv_47 + Add_48 + Relu_49      386.68           0.6267      2.6
[02/15/2022-17:37:15] [I]                                        Conv_50 + Relu_51      406.81           0.6593      2.7
[02/15/2022-17:37:15] [I]                                Conv_52 + Add_53 + Relu_54      386.52           0.6264      2.6
[02/15/2022-17:37:15] [I]                                        Conv_55 + Relu_56      406.60           0.6590      2.7
[02/15/2022-17:37:15] [I]                                Conv_57 + Add_58 + Relu_59      385.20           0.6243      2.6
[02/15/2022-17:37:15] [I]                                        Conv_60 + Relu_61      403.67           0.6542      2.7
[02/15/2022-17:37:15] [I]                                Conv_62 + Add_63 + Relu_64      378.94           0.6142      2.5
[02/15/2022-17:37:15] [I]                                        Conv_65 + Relu_66      405.72           0.6576      2.7
[02/15/2022-17:37:15] [I]                                Conv_67 + Add_68 + Relu_69      373.44           0.6052      2.5
[02/15/2022-17:37:15] [I]                                        Conv_70 + Relu_71      504.84           0.8182      3.4
[02/15/2022-17:37:15] [I]                                              Conv_72      526.81           0.8538      3.5
[02/15/2022-17:37:15] [I]                                Conv_73 + Add_74 + Relu_75       72.69           0.1178      0.5
[02/15/2022-17:37:15] [I]                                        Conv_76 + Relu_77      536.84           0.8701      3.6
[02/15/2022-17:37:15] [I]                                Conv_78 + Add_79 + Relu_80      675.83           1.0953      4.5
[02/15/2022-17:37:15] [I]                                        Conv_81 + Relu_82      537.78           0.8716      3.6
[02/15/2022-17:37:15] [I]                                Conv_83 + Add_84 + Relu_85      685.98           1.1118      4.6
[02/15/2022-17:37:15] [I]                                   GlobalAveragePool_86       15.28           0.0248      0.1
[02/15/2022-17:37:15] [I]                                              Gemm_89       52.45           0.0850      0.3
[02/15/2022-17:37:15] [I] Reformatting CopyNode for Input Tensor 0 to (Unnamed Layer* 105) [Shuffle]        3.19           0.0052      0.0
[02/15/2022-17:37:15] [I]                                               Total    15052.37          24.3961    100.0
```

Figure 16: Output of this command for ResNet34 on NVIDIA Jetson Nano (the model from the table above)

These numbers are useful for two reasons:

1. They will help you see if there's a layer that takes more time than you'd expect. For example, imagine a layer that takes 30% of the runtime. That's your first candidate for some improvement!

2. Say, layer X takes 2% of the runtime on T4, but 7% on Nano. That could be an indication that this operation is less optimized on a Jetson and it's worth looking into further.

   You can also do this in Infery, which was introduced in "How to Use Infery in 3 Simple Steps" (see page 23). Infery can conveniently highlight the layers that are the biggest bottlenecks. See example below:

```
In [2]: model = infery.load('yoloxS_640_fp16.engine',
framework_type='trt', profiling=True)
infery_manager -INFO- Loading model yoloxS_640_fp16.engine to the GPU
[TensorRT] WARNING: Using an engine plan file across different models
of devices is not recommended and is likely to affect performance or
even cause errors.
trt_engine -WARNING- You have specified "profiling=True" thus runtime
latency will be affected! Please do not use engine this for
benchmarks.
infery_manager -INFO- Successfully loaded yoloxS_640_fp16.engine to
the GPU.
In [9]: model.get_layers_profile_dataframe()
Out[9]:
                Layer Name    ms Percentile
0         Slice_4 0.184128  1.235265
1         Slice_9 0.081024  0.543568
2        Slice_14 0.150752  1.011354
3        Slice_19 0.080256  0.538416
4        Slice_24 0.150656  1.010710
..          ...   ...     ...
190 PWN(Exp_348, Mul_350) 0.008320  0.055817
191       Slice_355 0.106688  0.715741
192       885 copy 0.023584  0.158219
193       893 copy 0.017408  0.116786
194       898 copy 0.123616  0.829306

[195 rows x 3 columns]
In [10]: model.get_bottlenecks(5)
Out[10]:
               Layer Name    ms Percentile
134 Conv_248 || Conv_255 0.762560  5.115809
10        Conv_41 0.594592  3.988957
136       Conv_251 0.415680  2.788685
140       Conv_258 0.412672  2.768505
12        Conv_44 0.376800  2.527849
```

Resort to Neural Architecture Search

Let's assume you chose the best model architecture you could, localized all the bottlenecks, and made some changes to improve the runtime. The model is now ready to be trained. But after all the modifications, how can you guarantee that the new architecture is capable of reaching good performance results as well as your target accuracy metric?

The answer is very specific. A high level of expertise is crucial for success, as well as extensive resources, time, and budgets for iteration after iteration. Luckily, there is another option: neural architecture search (NAS).

The NAS process designs and searches for the best structure of the neural network for your task, performance targets, and inference environment. To be more specific, NAS starts with a candidate neural architecture, and it spans from there, creating a huge search space containing many potential architectures. The algorithm then analyzes those models and assesses the accuracy of the candidates. It uses a machine learning-based algorithm to analyze a given set of architectures and select those that have a high probability of possessing the best accuracy while also achieving the desired latency or throughput or model size.

Doing NAS the Easy Way

As you can imagine, NAS comes with challenges and is far from easy. The search space can be somewhere in the order of 10^{18}. When approached directly, even benchmarking at this kind of scale is a complex problem, let alone training. Now, imagine how much training time it would take: around 4 to 16 GPU days per model multiplied by 10^{18} !!!!

At Deci, we looked into how we can scale the optimization factor of this algorithm. Our proprietary NAS technology, known as Automated Neural Architecture Construction (AutoNAC), modifies the process, and therefore, unlike traditional methods, Deci's AutoNAC engine is a viable solution for AI teams of all sizes.

With NAS-based AutoNAC, everything is done in a single shot that takes around 2.5X the standard training time. The output is a custom model architecture that is guaranteed to meet your accuracy, latency, throughput, and model size targets. Compared to the theoretical cost of training hundreds or thousands of models, this is a major improvement in terms of time, money, and required manual effort. You can read more about AutoNAC in the white paper here.

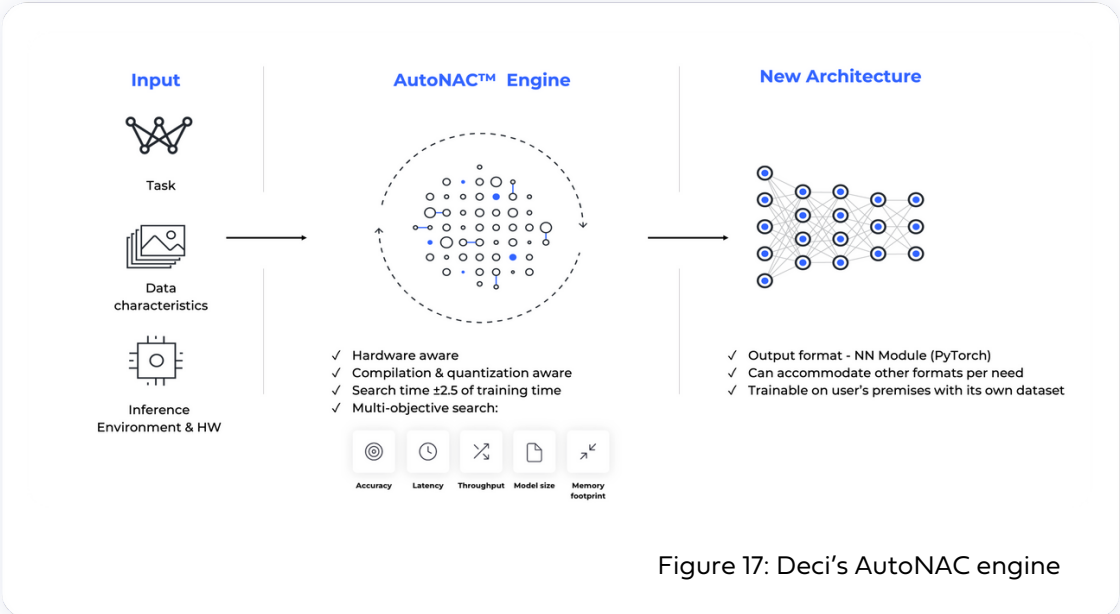


Figure 17: Deci's AutoNAC engine

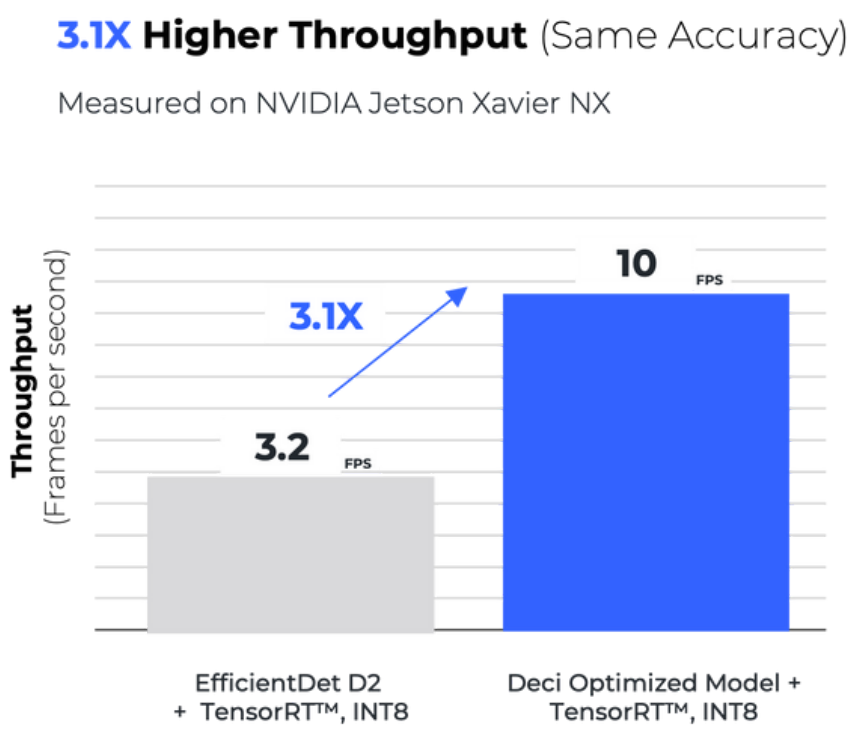


Figure 18: Enabling real-time semantic segmentation for an automotive application. Using Deci's AutoNAC engine, a faster and smaller model was built. Latency was reduced by 2.1X, model size was reduced by 3X, and memory footprint was reduced by 67% – all while maintaining the original accuracy.

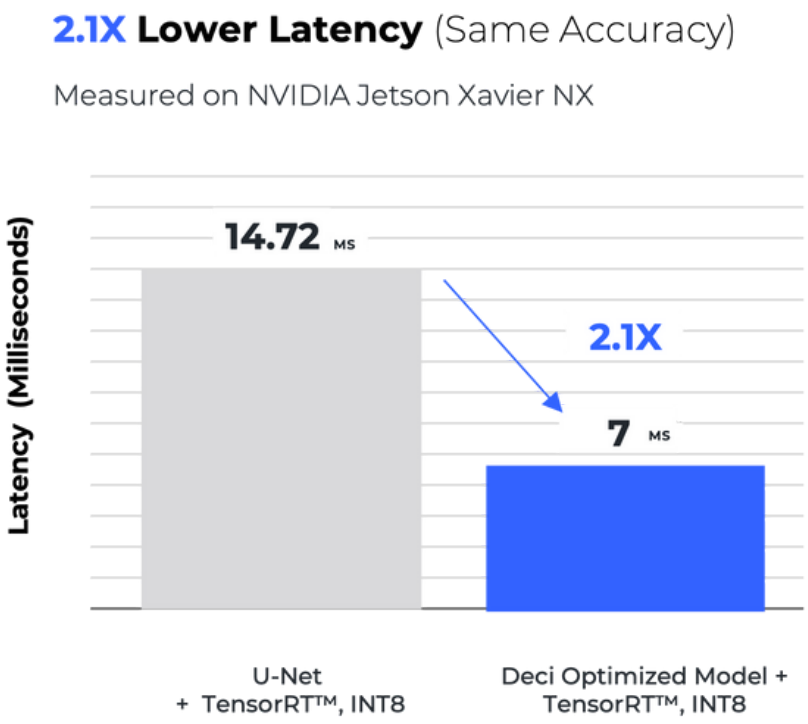


Figure 19: The new model built with Deci's AutoNAC engine, delivered a 3.1X increase in throughput while maintaining the same accuracy.

TRAINING YOUR MODEL FOR JETSON

If we're already talking about neural architecture search and the training involved in this process, we might as well dive deeper into the training itself. How do you train your architecture for Jetson? Well, aside from choosing the architecture, training a particular model for Jetson is the same as for any other hardware. It is performed on any server with strong GPUs, or on a couple of them; ultimately, the weights obtained are deployed into production together with the model.

But wait! Remember the quantization step mentioned earlier and how it can seriously decrease the precision? We noted specifically that using INT8 can hurt your model's accuracy. This is indeed oftentimes true, and training is the stage where you get the chance to mitigate this problem.

Say, for example, you decide that deployment in INT8 is your go-to option in terms of latency. You take your trained model and perform post-training calibration during quantization to INT8, but unfortunately get unsatisfactory results. This is when quantization-aware training might be what you need to restore the accuracy.

Quantization-aware training, or QAT for short, is a technique that introduces additional steps **during** training that prepare your model to be deployed in 8-bit. If deployment in 8-bit is not your plan, QAT is an unnecessary complication; but, otherwise it can be a very effective approach.

The name speaks for itself: training is performed with awareness that the inference will be done in INT8. It results in a much faster model with uncompromised accuracy. However, finding a library that offers a smooth transition to such a training regime can be a headache. Luckily, our team implemented quantization-aware training into SuperGradients - Deci's free, open source library for PyTorch-based computer vision models. To aid you in this stage even more, we collected some general tips and tricks for training any model. All of them are features of SuperGradients.

SuperGradients is a one-stop-shop for training or fine-tuning SOTA pre-trained models for all the most commonly applied computer vision tasks, such as object detection, image classification, and semantic segmentation for videos and images. It's a convenient way to start playing with the listed tasks or with your own model, and most importantly to experiment with the training tricks listed below.

1 Exponential Moving Average – EMA

EMA is a method that increases the stability of a model's convergence and helps it reach a better overall solution by preventing convergence to a local minima. To avoid drastic changes in the model's weights during training, a copy of the current weights is created before updating the model's weights. Then the model's weights are updated to be the weighted average between the current weights and the post-optimization step weights.

2 Weight Averaging

Weight averaging is a post-training method that takes the best model weights across the training and averages them into a single model. By doing so, we overcome the optimization tendency to alternate between adjacent local minimas in the later stages of the training. This trick doesn't affect the training whatsoever, other than keeping a few additional weights on the disk, and can yield a substantial boost in performance and stability.

3 Batch Accumulation

When you use a model 'off the shelf,' it generally comes with a suggested training recipe. The thing is, these models are usually trained on very powerful GPUs, which may mean the recipe is not necessarily appropriate for your target hardware. Reducing the batch size to accommodate your hardware will likely require tuning other parameters as well and you won't always get the same training results.

To overcome this issue, you can perform several consecutive forward steps over the model, accumulate the gradients, and backpropagate them once every few batches. This mechanism is known as batch accumulation.

4 Precise BatchNorm

BatchNorm layers are meant to normalize the data based on the dataset's distribution. Ideally, we would like to estimate the distribution according to the entire dataset. Although this kind of estimation isn't possible, we can use BatchNorm layers to evaluate the statistics of a given mini-batch throughout the training.

The paper Rethinking "Batch" in BatchNorm by Facebook AI Research, showed that these mini-batch based statistics are sub-optimal.

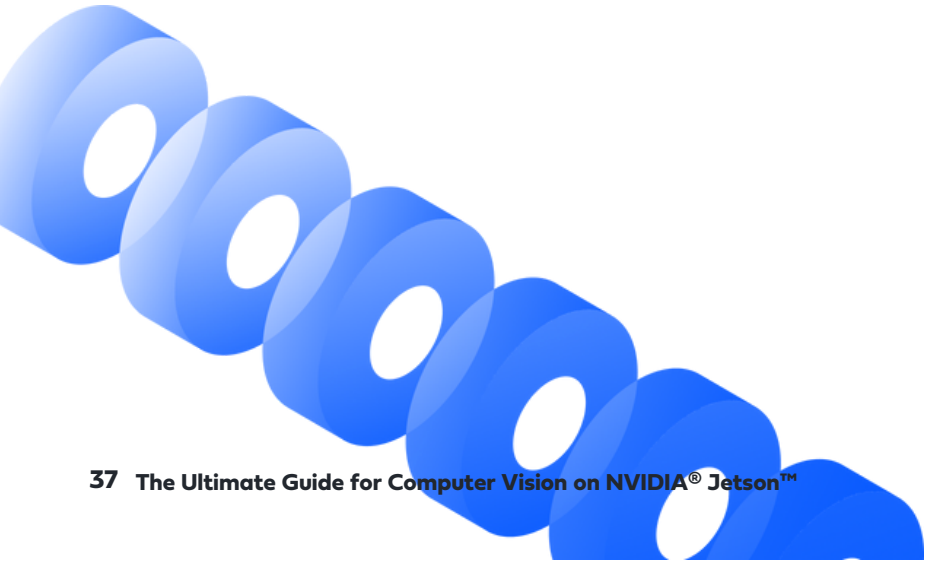
Instead, the data statistics parameters (the mean and standard deviation variables) should be estimated across several mini-batches, while keeping the trainable parameters fixed. This method, known as Precise BatchNorm, helps improve both the stability and performance of a model.

5 Zero-weight Decay on BatchNorm and Bias

Any 'go-to' model for various computer vision tasks is likely to have batchnorm layers and biases added to their linear or convolution layers. When you're training this kind of model, it works better if you set the optimizer's weight decay to zero for the batchnorm and bias weights.

The weight decay is a regularization parameter that prevents the model weights from 'exploding.' Zeroing the weight decay for these parameters is usually done by default in various projects and frameworks, but it's still worth checking since we noticed that it's not yet the default behavior for PyTorch.

Weight decay essentially pulls the weights towards zero. While this is beneficial for convolutional and linear layer weights, batchnorm layer parameters are meant to scale (the gamma parameter) and shift (the beta parameter) the normalized input of the layer. As such, forcing these values to a lower value would affect the distribution and lead to inferior results, so it's helpful to enable zero-weight decay for these parameters.



BEST PRACTICES FOR DEPLOYING YOUR MODEL ON JETSON

By now, you should have completed the steps above and reached deployment. Congratulations! It's time to complete the very last stage before your application is seen by the world.

This includes addressing potential challenges such as power distribution and deployment parameters. Let's look at these issues one by one.

Distributing and Balancing Compute Power

Fully utilizing the power of a single Jetson module is tricky enough and it was already discussed in "Maximizing Compute Power with the Right Configuration" on page 11. However, what if we're not just talking about one module but a few of them? After all, Jetson modules are powerful, but you may need more compute power than just one module.

In some cases, you'll want to connect to as many IoT sensors as possible to reduce the hardware costs. In other cases, like autonomous vehicles, you'll want to maximize the number of model instances that you can fit on a single Jetson device and run them in real-time (in terms of memory).

The amount of IoT devices connected directly affects the IO of the Jetson device. This means the concurrency sweet spot must be taken into account when searching for the optimal number of processes and threads.

Finding the Right Deployment Parameters

There are no firm rules when it comes to deployment parameters such as batch size, image resolution (if you operate on images), model size, or the number of simultaneous inputs. Your application will usually apply some constraints on the choices you have. What's more, the constraints of the Jetson itself make it difficult to tune to the right deployment parameters that will optimize compute power and memory space.

Based on our experience with AI model optimization, we've gathered the three best practices to address these issues.

Use the Right Batch Size

Because Jetson has limited memory space, you'll need a batch size that is small but not so small that it adversely impacts throughput. For example, if your application involves object detection, the ideal batch size should allow you to reach 30 FPS end-to-end to perform in real-time. When working with a GPU, we strive to use a batch size higher than 1 but lower than 64.

Deci has automated solutions for this long and frustrating process, but generally, we recommend keeping the batch size small if the output is large. Sometimes it's better to use 4 replications of batch size 2 than work with a single replication in a batch size of 8. If you haven't done it already, consider using the Deci platform to determine how your model performs with different batch sizes on Jetson devices.

Use Concurrent Code and Multiple Processes

One way to ramp up your code optimization is by using concurrent code and allowing multiple processes to run at the same time. This lets the application carry out analytics on more inputs at once, instead of just one.

- Don't be constrained by a Python-like mindset that runs line-by-line, and use an asynchronous approach whenever possible. Python is an interpreted language and we often use it to implement inference, but we rarely use it to implement a multi-process solution that uses multiple CPU cores at the same time.

- If you want better throughput, increase parallelism. Your main process can spawn multiple processes, where each one loads the model to the GPU and runs inference independently.

- The model isn't the only part that does the heavy-lifting though. Take the time to find the optimal number of processes for loading data, inference, post-processing, and client threads. If you use Python (and quite likely you do), you can leverage the Python Multiprocessing module to enable parallel / asynchronous code execution.

To summarize this part, the number of threads and processes makes a huge difference. The optimal configuration or combination of these is usually achieved by trial and error. They have to live in harmony and work in concert so they don't disrupt one another. Make sure to invest enough time in this step since it's crucial for optimal deployment on Jetson.

Use a Swap File for Production

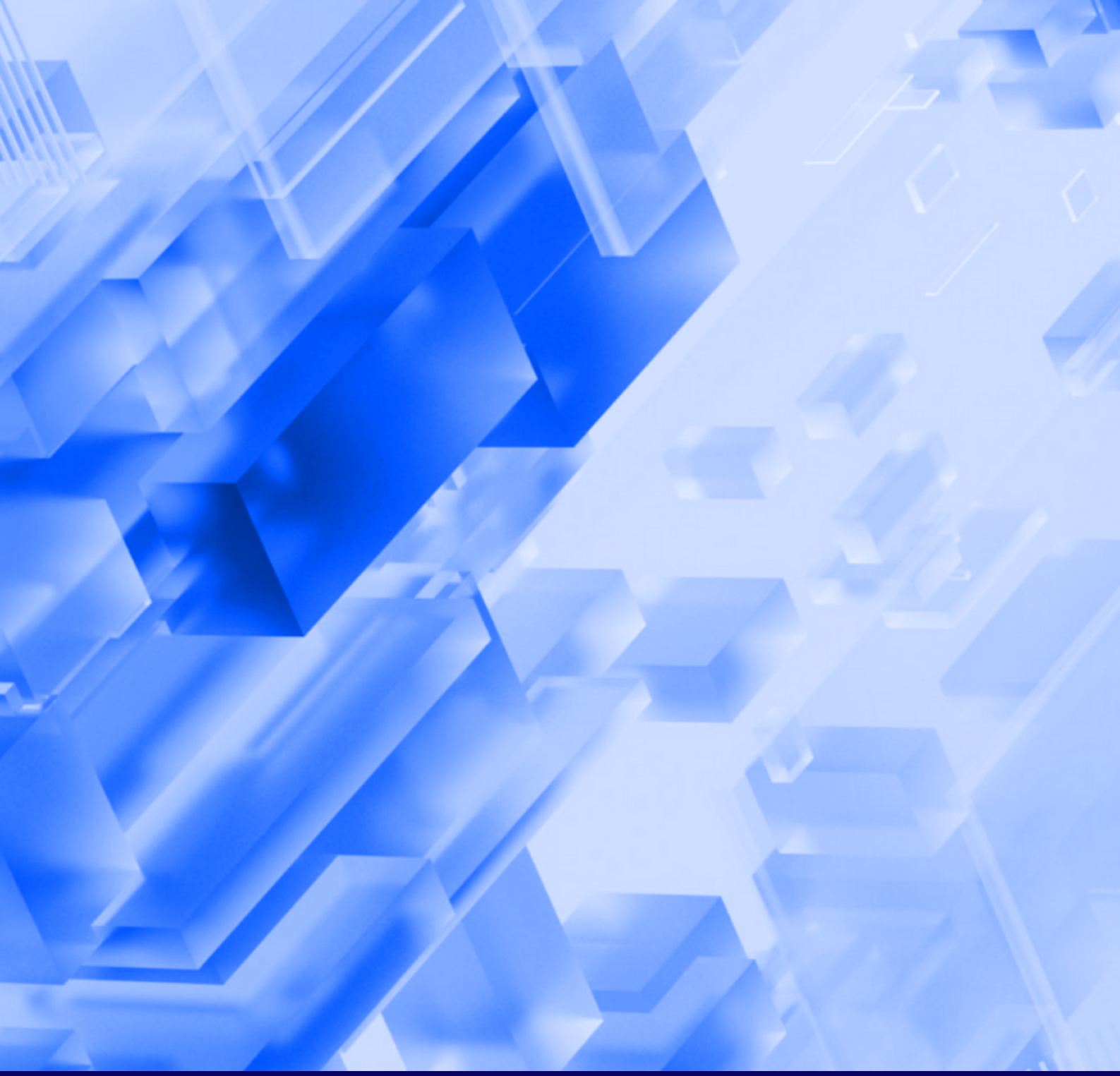
Although this may slightly compromise your ability to get an accurate reading for speed, it serves to increase the device's fault tolerance.

SUMMARY

NVIDIA Jetson is one of today's most popular families of low-power edge hardware. It's designed to accelerate deep learning models on edge hardware, whether for robotics, drones, IoT devices, or autonomous cars.
We know that hardware plays a major role in performance, but you also need to tune, optimize, or update your model so it does its best work on your hardware.

In short, deploying deep learning AI models on Jetson can be tricky. But once you have the right tools and best practices in your arsenal, it becomes easier. We looked into ways you can optimize for better performance, find the parameters for production that are most cost effective, adjust your compute and power settings, and do your testing in a way that identifies which aspects of your environment are problematic.

And, of course, you can always request a free trial and use the Deci platform to automate your model optimization for Jetson, build custom hardware aware model architectures to reach unparalleled accuracy and speed and use the Infery Python runtime engine to seamlessly deploy your optimized models to production.



ABOUT DECI

Deci enables deep learning to live up to its true potential by using AI to build better AI. With Deci's platform, AI developers can easily build, optimize, and deploy highly accurate and efficient models to any environment including cloud, edge, and mobile. Leading enterprises are using Deci to boost their deep learning models' performance, shorten development cycles, enable new use cases on edge devices, and reduce computing costs.

For more information, visit deci.ai

deci.