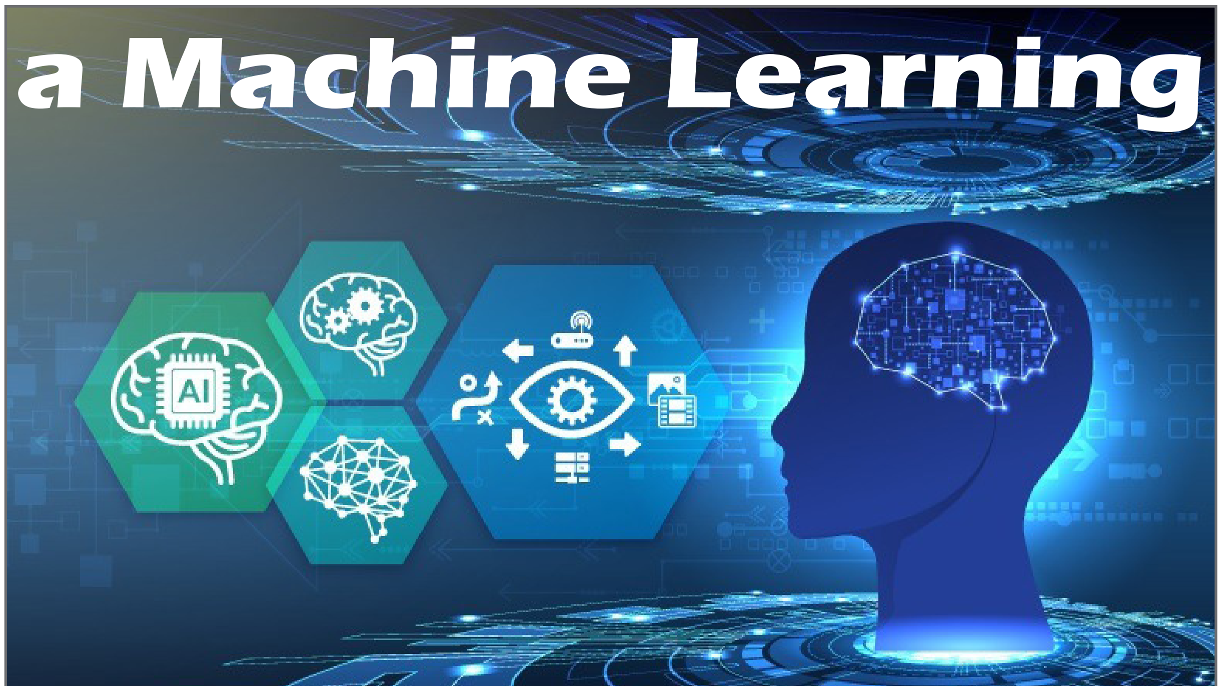
How to Build, Train, Test, and Deploy



Building a machine learning (ML) model is exciting. Arriving, after tweaks and changes and sleepless nights, at something you feel you can use in the real world is likely a relief. Unfortunately, that proof of concept you've just created marks the beginning of your work, not the end. After establishing the baseline model, it's time to build a more thorough version, test, deploy, and maintain that model. The work of an ML team is never finished, it seems. Here are some considerations as your team turns its proof of concept into the real deal: a model that is the engine of a real-world application.

If we think about an object-detection model for pedestrians, we can walk through the various steps required to turn the concept into production. Before we get into the nitty-gritty details, teams will have to agree upon a few basics.

• What kind of programming language are you using? There are always new languages, like Julia, arriving on the scene. Many people work in R or Python since they have the largest support for ML frameworks.

• Then there is the mass of existing frameworks from which to choose. Some, like TensorFlow, have a large user base and existing tome of documentation. Others might offer something more closely aligned with your vision but with less available support.

• Will your team run this model on your own machine with GPUs, some embedded target hardware, or will you run it entirely in the cloud?

• What algorithm will the model require? In this example, the team might opt to use the "You Only Look Once (YOLO)" real-time object detection algorithm, as it exists for your team's explicit purpose.

These aren't the only questions to ask yourself as you get started, but answering them is imperative to the success of your endeavor. More so than establishing specific answers to those questions, understanding how to arrive at an answer will be important. The way that you approach building, testing, and deploying your model will change along with your confidence in how well you're solving the business problem in question. With a solid process in place, your team can start acquiring data, building, and training, and finally validating your model. Remember, structure your process for rapidly iterating rather than having a perfect end-to-end solution all at the start.

Data Acquisition and Gathering

For the hypothetical object-detection model, first, decide on the input and output. The input is a given image. The output is an image with bounding boxes-boxes around pedestrians with X/Y coordinates and height and width parameters. To arrive at that basic input/output arrangement, the model will need annotated images for eventual training.

Yet, before training the model with the annotated data, you'll need to consider where that data will reside. Where is your data warehouse and how is that data arriving at the warehouse and moving to the model? Will those inputs be available when you finally begin serving your model in the real world? In our example, if we use HD images for training and the images we get while serving are in lower resolution camera, the model is likely to fail. Are you using the same code base for the preprocessing phase of training and serving your model? Again, a litany of questions about your process will arise at this juncture, and it's important to understand how you'll answer them.

If you have your data storage and movement architecture arranged, it's time to acquire the specific data you need. In this case, we're looking for, say, a few thousand images of streets with pedestrians. Once you acquire the data necessary to build the model, you'll have to, via randomization, separate them into three different sets. A good rule of thumb is to keep 80 percent for your training set, and 10 percent each in your validation and test set. It's important here to not look at the test set yet.

The goal is not to overfit the model to any one set of data. If your model works in training but doesn't work in validation, it won't work in production. If it works in training and validation but not testing, you've probably overfitted the model to the training and validation data. Next, we'll examine how to best approach training, testing, and validating.

Building, Training, Testing, Validating

If you overfit the model to the training images, it may mean that the model only identifies pedestrians under very specific circumstances. Maybe it detects pedestrians in photos taken outside but isn't able to detect them in photos taken from behind windows. If you only have training data from rainy days, the model might not detect pedestrians in sunny-day photos from your validation or test set. For this purpose, it's important to establish a ground truth for your data based on human experience that ensures good coverage of all the important cases in each of the datasets.

Establishing a ground truth requires asking humans to annotate some of your data set. Because a human will know-regardless of whether or from where a photo was taken-what a pedestrian looks like, a group of annotators will be able to create bounding boxes more accurately than an algorithm alone can. By incorporating a ground truth into your training data, you're likely to avoid the overfitting problems that arise. Creating a ground truth with a panel of skilled human annotators will also give your model

a level of accuracy to achieve: The goal is to reach human-level accuracy, not 100 percent accuracy.

After splitting your data and achieving a human-created ground truth, it's time to begin training the model with that annotated data. As you send training data through the model, you'll have to determine if the incremental improvements you make are worth the investment. If more time training the model leads to a one percent increase in accuracy that will impact only about a thousand requests later, it probably isn't worth the extra time. If more time spent training leads to a one percent increase for one million users or improving coverage of your edge-cases, then it's likely worth even the small improvement.

As you hone the model through training, occasionally leverage the test set of data as a benchmark for whether the model will work in production. As you work from test data–which can tell you if your model is trained appropriately–to validation data–which helps ensure you haven't overfitted your model–you'll likely end up with a few more iterative changes to your model. Eventually, it will be time to launch your model in production.

Serving the Model in Production

You've tested and validated the model, and are ready to send it into the real world for consumption. Tools, such as Seldon, an ML deployment platform offer out-of-the-box serving capabilities. However, these types of frameworks lack business logic, which means your team will have to build rules to specify who can use the model or when to use this model rather than another model your organization may employ.

A model in production is a model that needs constant management. Let's say our object-detection model requires weekly updates. To perform regular updates, your team will need an infrastructure in place that supports that process. For example, the infrastructure you select should minimize loading and moving data for analysis, as your model in production will require continuous improvement. This constant improvement is only possible when your team logs the model activity to look for signals of failure and incorporates analytics to ensure your model can meet the demand of requests it's experiencing.

One major issue to monitor and analyze for is concept drift or shift. Drift happens when your data changes somewhere along the pipeline from when you trained your model to it being served live in production. To monitor for this, your team should track a signal that alerts you to model failure. As you monitor your model in production, you will end up with what amounts to a few different models, as you tweak and change your initial algorithm. As you arrive at each successful variation of a model, it will become your champion model, which you can use as a basis for subsequent models. Your champion model has gone through testing, validation, and monitoring in production, so it represents the most effective version of your ML journey.

Finally, during your build-to-serve process, keep in mind the final location of your model. If our pedestrian-detection model will be served in an embedded car camera, then building a model that only works on HD images in high compute-available software environments will be useless. Ensure you have the correct processing capacity available and that your architecture and framework are appropriate for the end goal.

Iterate and Collaborate

Building and deploying a machine learning model is an iterative process. Taking a proof of concept all the way through to a deployed product is painstaking and involves many stops and starts along the way. Have patience and lean on your team to deliver a truly collaborative effort. It's the only way you'll achieve the goals you set out to accomplish by building an ML model in the first place.