Machine Learning on iMX

Diego Dorta

Version 0.1, Mar 21, 2018

Table of Contents

1. Overview

This document walks through the *Artificial Intelligence*[1] world offering an overview about this technology and its usage on *embedded systems*. Examples are explored and shown to demonstrate how to get started with *Machine Learning*[2] from *scratch* on *i.MX* boards.

1 . [Artificial Intelligence]
2 . [Machine Learning]

1.1. Introduction

Machine Learning (*ML*) make our minds probably deliberate about robots and talking machines, some of us might even related ML to robots bent on destroying the world. However, this technology is more about building intelligent systems with decision making abilities.

Think about *Machine Learning* as a subfield of AI that can analyse large amounts of data and automate analytical model building. This branch of Artificial Intelligence was born from *pattern recognition*[1] and the theory that computer can learn, and have the ability to independently draw knowledge from experience.

Historically, AI programs typically excelled at just on feature, like the *Deep Blue*[2] computer, that could play chess in a high level championship, but that's all it could do. Today, large data centers and huge storage capacities make things, that were believed for years to be distant concepts, possible.

Machine Learning exists solely as software, but for most cases ML requires the use of hardware components to build intelligent machines. Having a basic form, ML combined with embedded systems can reach significant improvements in image and video recognition, and the reason is due a certain level of *smartness* that embedded systems reached over the last years. This also explains the cause it has been gaining space and moment in several types of industrial processes nowadays.

This field boils down on spending time to write many lines of code that eventually solve a problem by applying some type of *intelligent* algorithm. For instance, some smart homes have lighting systems that automatically turn on and off based on whether anyone is present in the room. The system does not amuse, but when you start thinking about it, you realize that the system is actually making decisions on its own.

The much-needed algorithms required for real-time image and video recognition are being

developed and will get us quickly. Not just that, several types of algorithms for different issues the future is reserving for us, and the embedded system world is fully prepared to completely replace the work of a human being.

1 . [Pattern Recognition]
2 . [Chess Computer]

1.2. Branches

Machine Learning and *Deep Learning*[1] are two branches of AI that use the possibilities of *Big Data*[2] (a huge amount of data is well known as *Big Data*) to optimize processes, find new solutions, and gain new insights. Keep in mind that *ML* is a kind of intelligence developed as a result of excellent experiments, this type of intelligence can make machines and devices think and act like human beings. On the other hand, *Deep Learning* is the process that implement *ML*. These fields combined will lead the world into an **exponential growth**.

1 . [Deep Learning]
2 . [Big Data]

1.3. Examples

Differently from *Deep Blue*, the currently programs can solve many problems without needing to be rewritten. Put differently, *ML* is the study of algorithms that learn from examples and experiences instead of relying on **hard-coded rules**. These algorithms can make predictions based on a huge amount of data, just by recognizing aspects and relations. For instance, think about the following situation where is required to write a code to recognizer two types of fruit. How this can be solved? In a first approach, it seems to be easy, but it is almost impossible to solve without *ML*.

1.3.1. Types of Fruit

This fruit example takes an image file as input, analyses it and output the types of fruit. To start developing this code it would be necessary to write lots of manual rules. That works fine for simple images, but as **deeper it dives into the problem, more messy it gets and the written rules starts to break down**. How simple rules can handle black-and-white photos or images with no fruits at all? In fact, for just about any rule it is possible to find an image where it will not work for sure. After spending many hours it can get there, but that's just to tell the differente between two types of fruit. The problem begins on any new problem, where it is necessary to start all over again and write more rules for the new issue.

1.3.2. Learning the Game's Rules

Clearly it requires something better. The *ML* can solve this problem using an algorithm that can figure out the rules itself, by recognizing patterns and draws conclusions from data without the necessary to write them by hand, and its findings can be used further as well. The most commonly used learning method is **image recognition**, other uses examples are: *digital assistants or intelligents bots*, *face recognition*, *speech recognition*, *natural language processing*, *automated translation*, and the trending one *autonomous car*.

1 . [Image Recognition]
2 . [Virtual Assistant]
3 . [Facial Recognition System]
4 . [Artificial Intelligence and Natural Language Processing]
5 . [Machine Translation]
6 . [Autonomous Car]

1.4. 4.0 Industrial

Known as *The Fourth Industrial Revolution* or *4.0 Industry*[1], that in fact is happening right now, this concept consists in a smart factory where production processes are connected, such as machines, interfaces and components communicating each other. Large amounts of data can be collected to optimize the manufacturing process, and AI expected to influence the growth of these companies by using the *ML* field.

The industries are already using the AI in maintenance and support services by capturing the energy consumption of individual machines, analyzing maintenance cycles, and then optimizing them in the following stage. As the amount of data increases, the system becomes better at optimizing itself and making more accurated predictions.

The world should embrace and encourage who is up to *invest*[2] in AI, according to experts, companies that use machine learning processes increase their economic performance. The biggest benefits are expected in the IT and finance sectors, telecommunications, and the manufacturing industry.

1 . [Industry 4.0]
2 . [AI and Economy]

1.5. Quantum computing

We are at the limits of the data processing power of traditional computers and the data just keeps growing. To be continued.

2. Applications

The iMX is an example for applying this technology. Think about the types of fruit that was mentioned before or any type of similar problem. It is possible to create a very simple code using *ML* for solving that problem by training a **classifier**. Imagine the classifier as a function which takes some data as input and assigns a label to it as output.

For instance, using the fruit example we can show a fruit to iMX8 board and can it tell us what type of fruit is it? Or better than that, can we have a picture and the board says what is in it? Yes for all.

More than just fruits, ML also has techniques that can make iMX transform a car into an autonomous one, recognize your face and speech, translate data and make predictions for you. These topics will be approach on the next section.

The technique that approaches this type of issue is called *supervised learning*[1], even with several techniques this is the most appropriate because it can create a classifier by finding patterns in examples, instead of writing rules. It begins with examples of the problem that needs to be solved.

1 . [Supervised Learning]

2.1. Machine Learning layer

To work with *Machine Learning* on *i.MX* boards, it is necessary to have installed *Python language* and its package installer *pip*, along with a few libraries which are basically:

- Numpy - Fundamental package for scientific computing with Python.
- Scipy - Open-source software for mathematics, science, and engineering.

Yocto has recipes for *python*, *pip* and *numpy*. Currently, it does not have a recipe for including *scipy* and its dependencies *libatlas-base-dev* (libraries etc. for ATLAS/BLAS) and *gfortran* (GNU Fortran compiler). For that reason, instead of struggling for awhile to create these recipes, a *Debian file system* is used momentarily. The *meta-machinelearning* layer will be created when these dependencies have recipes on Yocto.

For making it easy to develop codes using *ML*, a *Python* library called *scikit*[1] can help us to get a solution, for example, to the fruit issue.

1 . [Scikit Learng - Simple and efficient tools for data mining and data analysis.]

2.1.1. Creating image

This section assumes you have a basic knowledge on *i.MX* systems. The first step consists on creating a image using Yocto:

1. Create your *core-image-base* using *meta-fsl-arm*:

```
$ MACHINE=imx6qsabresd DISTRO=fslc-framebuffer source setup-environment build
```

2. Add the following line on the *bb.layers* of your project:

```
${BSPDIR}/sources/meta-openembedded/meta-python/
```

3. Add the following line contaning the packages to be installed on the *local.conf*:

```
CORE_IMAGE_EXTRA_INSTALL += " python python-pip python-core python-numpy python-
distutils "
```

4. Start the building:

```
$ bitbake core-image-base
```

This can take several hours depending on your machine and network speed connection. 5. Flash the image located on *tmp/deploy/images* folder into the SD card:

```
gunzip -f core-image-base-imx6qsabresd.sdcard.gz
dd if=core-image-base-imx6qsabresd.sdcard of=/dev/sdX status=progress && sync
```

As mentioned before, if we have the recipes for all the dependencies the normal procedure would end here, however a uglyhack can solve our problem at the moment. The next section will change the *root file system* for a *Debian* one that includes a *package manager* which allows us to install the requires packages for *scikit-learn*.

2.2. Debian File System

NOTE	This section is a temporary one until the Yocto recipes be made it as said before, it also uses SD card fully burned from the previoulys step. The next steps were copied from the post How to create a Debian image to i.MX.

Start by installing the requires packages for creating the file system:

```
$ sudo apt-get install debootstrap qemu-user-static
```

Create a folder to mount your SD card and start the installation:

```
$ mkdir mounting
$ sudo mount /dev/sdXC mounting/
$ sudo debootstrap --foreign --arch=armhf wheezy mounting/
```

On */dev/sdXC* should be your root partition, for instance: */dev/sdb2*.

Enable *QEMU ARM* and *second stage*:

```
$ sudo cp /usr/bin/qemu-arm-static mounting/usr/bin/
$ sudo chroot mounting/ /debootstrap/debootstrap --second-stage
```

Add the following line in the file *mounting/etc/inittab*:

```
T0:23:respawn:/sbin/getty -L ttymxc0 115200 vt100
```

Add the eth0 configuration above in the file *mounting/etc/network/interfaces*:

```
auto eth0
iface eth0 inet dhcp
```

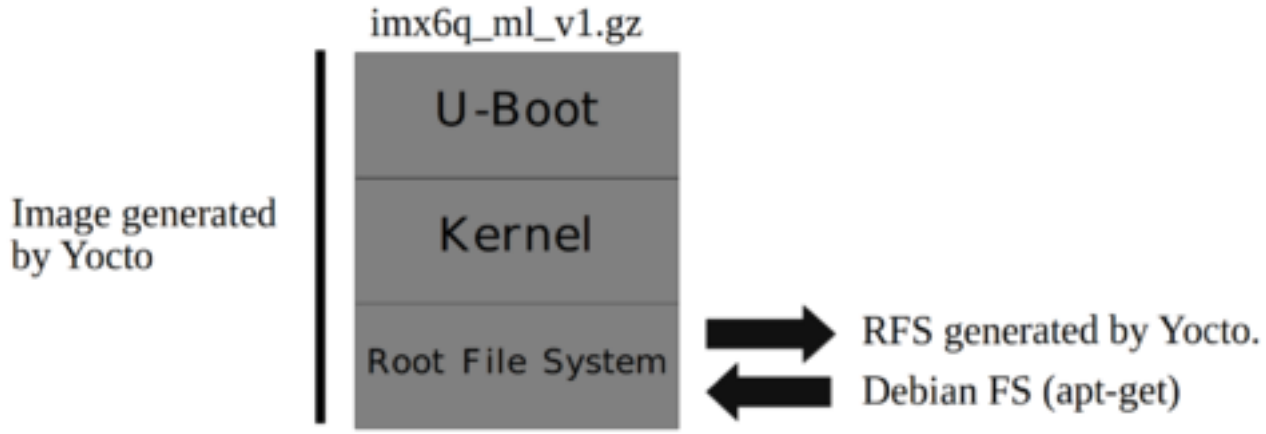
To enable the system to let you enter as root edit the file *mounting/etc/shadow* changing the first line to be like the following (remove the *):

```
"root:*:15880:0:99999:7:::"
```

Unmount the partition and boot the board:

```
$ sudo umount mounting/
```

The following image will show what the above steps have done with the image generated by Yocto.



NOTE	Again, the *root file system* was changed while the recipes for Yocto are being prepared.

2.3. Board

On your board, run the following command for installing the required packages:

```
$ apt-get update
$ apt-get install gfortran libatlas-base-dev python python-pip python-dev
```

Now, install the packages required by *Machine Learning*:

```
$ pip install -U numpy --index-url=https://pypi.python.org/simple/
```

```
root@dorta:~# python
Python 2.7.3 (default, Mar 14 2014, 17:55:54)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import numpy
>>>
```

```
$ pip install -U scipy --index-url=https://pypi.python.org/simple/
```

```
root@dorta:~# python
Python 2.7.3 (default, Mar 14 2014, 17:55:54)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import scipy
>>>
```

Then, install the *scikit-learn* library:

```
$ pip install -U scikit-learn --index-url=https://pypi.python.org/simple/
```

```
root@dorta:~# python
Python 2.7.3 (default, Mar 14 2014, 17:55:54)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import sklearn
>>>
```

Your board is configured for running applications using *Machine Learning* codes.

2.4. Compressing

To compress the image, just run the following command:

```
sudo dd if=/dev/sdX count=800 bs=1M status=progress | gzip > ml.gz
```

To uncompress the image, just run the following command:

```
gzip -dc ml.gz | sudo dd of=/dev/sdX status=progress && sync
```

2.5. Image

These steps take a few hours, if you do not have time you can download on the following link, and burn it into the SD card as follows:

```
gzip -dc ml.gz | sudo dd of=/dev/sdX status=progress && sync
```

1 . [i.MX6Q Sabre SD - Machine Learning Image.]

2.6. Fruits code - Supervised Learning

This step consists to collect training data. For solving this problem, we are going to write a function to classify a piece of fruit. For starters, it will take a description of the fruit as input and predict whether it's an **apple** or an **orange** as output, based on features like its weight and texture.

Imagine you go to the mall and stop your car at the front door. Then, your car goes park by itself. When the car is parked, it keeps monitoring your steps by GPS. When you get to the front door of the mall again, your car goes where you are to pick up you. This examples can be implement using *ML* for training data, such as learning the way, obstacles, where is the front door and etc.

To use the *supervised learning*, let's follow a recipe with a few standard steps:

2.6.1. Collect Training Data

This first step consists on collecting the training data, these are examples of the problem that needs to be solved. To collect this data, imagine an orchard, you look at different **apples** and **oranges**, and write down measurements that describe them in a table. In *ML* these measurements are called *features*.

Table 1. Features as Training Data

Weight	Texture	Label
150g	bumpy	orange
170g	bumpy	orange
140g	smooth	apple
130g	smooth	apple

This example must be simple, so let's just use two types. A good feature makes it easy to discriminate between types of fruits, the features in this case are the weight and texture, and each row in the training data is an example describing a piece of fruit. How much does each fruit **weights** in grams? And about its **texture**? Is it be **bumpy** or **smooth**?

The last column is called the label, that identifies what type of fruit is in each row, and there are just two possibilities - **apple** and **orange**. The entire table is the training data, think of these as all the examples we want the classifier to learn from. The more training data you have, the better a classifier you can create.

Writing the code

Let's write down our training data in code:

```
import sklearn
features = [[140, "smooth"], [130, "smooth"], [150, "bumpy"], [170, "bumpy"]]
labels = ["apple","apple","orange","orange"]
```

Two variables *features* and *labels* were used, features contains the first two columns, and the labels contains the last. You can think of features as the input, and the classifier and labels as the output

we want it.

Scikit-learn uses real-valued features, so let's change the variable tupes of all features to *ints* types intead of *strings* types. Here let's use **0** for **bumpy** and **1** for **smooth**. Doing the same for labels, we have **0** for **apple** and **1** for **orange**.

```
import sklearn
features = [[140, 1], [130, 1], [150, 0], [170, 0]]
labels = [0, 0, 1, 1]
```

2.6.2. Training a classifier

This second step uses the written features to train a classifier. This particular case, the type of classifier that will be used is called *decision tree*.

Decision tree

More details about the mechanism of this *Decision tree* will be explain on the next section. For now, think of a classifier as a **box of rules**. That's because there are many different types of classifier, but the input and ouput are always the same.

Importing the tree

To import this *decision tree*, change the *import* line in the code, and call the function as follows:

```
from sklearn import tree
features = [[140, 1], [130, 1], [150, 0], [170, 0]]
labels = [0, 0, 1, 1]
clf = tree.DecisionTreeClassifier()
```

At this point, it's just an empty box of rules. It does not know anything about **apples** and **oranges** yet. To train it, we will need to use a learning algorithm. If the classifier is a box of rules, you can think of learning algorithm as the **procedure that creates them**. It does that just by finding patterns in the training data. For example, it might notice **oranges** tend to weight more, so it will create a rule saying that the **heavier** fruits are more likely to be an **orange**.

Fit Function

In this *scikit* library, the training algorithm is included in the classifier object, and it's called *Fit*, as follows:

```
from sklearn import tree
features = [[140, 1], [130, 1], [150, 0], [170, 0]]
labels = [0, 0, 1, 1]
clf = tree.DecisionTreeClassifier()
clf = clf.fit(features, labels)
```

You can think of *Fit* as being a synonym for "find patterns in data". The next section get into the details of how this happens under the hood.

At this point, the classifier is trained. So let's take it for a spin and use it to classify a new fruit. The input to the classifier is the features for a new example, let's also pass the *argument variables* to make it easier, for measuring how much time the board took for training the data.

```
import sys, time
from sklearn import tree
features = [[140, 1], [130, 1], [150, 0], [170, 0]]
labels = [0, 0, 1, 1]
start_time = time.time()
clf = tree.DecisionTreeClassifier()
clf = clf.fit(features, labels)
weight = sys.argv[1]
texture = sys.argv[2]
print clf.predict([[weight, texture]])
print ("Time for training data: %s seconds" % (time.time() - start_time))
```

2.6.3. Output fruit

Let's say the fruit we want to classify is **160 grams** and **bumpy**, the output will be **0** if it's an **apple** or **1** it's an **orange**. Before hiting enter and see what the classifier predicts, let's think for a second.

If you had to guess, what would you say the output should be? To figure that out, compare this fruit to our training data. It looks like it's similiar to an orange, because it's heavy and bumpy. That is what I'd guess anyway, and if we hit enter:

```
root@dorta:/home/machine_learning#
root@dorta:/home/machine_learning# python test.py 160 1
[1]
Time for training data: 0.00309199999901 seconds
root@dorta:/home/machine_learning#
```

The result is **[1]**. It's what our classifier predicts as well (recall we used **0** for **apple**, and **1** for **orange**). The first *Machine Learning* program is created. It is possible to create a new classifier for a new program, just by changing the training data. That makes this approach more reusable than writing new rules for each problem.

This fruit example uses a table of features instead of pictures to be simple. It is possible to use pictures as training data. It would be something like:

Table 2. Picture as Training Data

Picture	**Label**
orange_image.jpg	orange
orange_image.jpg	orange
apple_image.jpg	apple

Picture	Label
apple_image.jpg	apple

This training data will be approchead on further examples. This code is more general, the neat thing is that programming for *Machine Learning* is not hard, but to get it right, you need to understand a few important concepts on the next section.

The next document will contain examples for: unsupervised learning, semi supervised learning and reinforcement.

3. Contact

In case you need help: <diego.dorta@nxp.com>