Enter your Paper Title Here

Francesco Argentieri
ID 183892
Email: francesco.argentieri@studenti.unitn.it

Abstract—Abstract goes here

Index Terms—Python, Keras, Movidius, Intel, Neural Networr, Iception, TensorFlow, ImageNet, ncsdk, API, back-end.

I. Introduction

In this era, a large amount of structured and un-structured data became available. **Machine Learning** (**ML**) has evolved as a branch of artificial intelligence: it envisages the development of self-learning algorithms, which are capable of acquiring knowledge from data with the aim of making predictions. Instead of requiring a human presence who manually enact the rules and build models for the analysis of large amounts of data, machine learning offers a more efficient alternative to capture the knowledge in the data. Machine learning aims to gradually improve the performance of fore-casting models and to make data driven decisions. In this section we will examine the three different types of machine learning: *supervised learning*, *unsupervised learning* and *reinforcement learning*. Where we will show the fundamental differences between these types of learning. [1]

A. Supervised learnig

The main purpose of supervised learning is to derive a model from training data, which allows us to make predictions for data that are not available or future. Here, the term "supervision" refers to the fact that the output signal labels of the sample sets are already known. A supervised learning task, which is based on discrete class labels, is also called a classification task. Another supervised learning subcategory is regression, whose resulting signal is a continuous value. Classification is a sub-category of supervised learning, which has the goal to provide class category labels for new instances, based on observations made in the past. These labels are discrete, unordered values that can be considered as belonging to a group of instances. However, the set of class labels does not necessarily have to be a binary nature. The predictive model identified by a supervised learning algorithm can consider each class label that is present in the learning dataset of a new instance, which is not labelled.

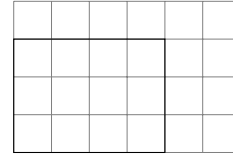A typical example of *multi-class classification* is the recognition of hand-written text. [1]

Fig. 1: supervised learning scheme

B. Reinforcement learnig

Another type of machine learning is reinforcement learning. Here, the goal is to develop a system (*agent*) for people to improve their performance. In order to do so, that system is based on interactions with the environment. Since information relating to the current state of the environment include also a *reward* signal, we can consider strengthening learning as an example of supervised learning. However, this feedback is not the correct label or the true value of truth, but it represents the quality of the measurement of the performance mea-sured by the reward function. Through interaction with the environment, an agent can then use reinforcement learning to learn a series of actions, which maximize this reward through a trial-and-error exploratory approach or deliberative planning. [1]
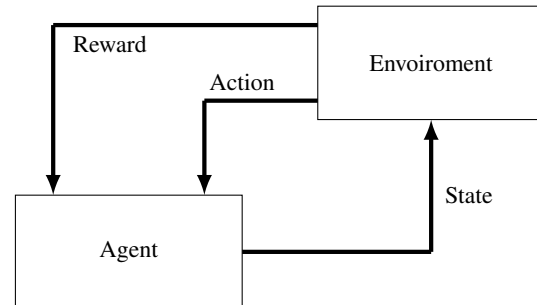

Fig. 2: reinforcement learning scheme

C. Unsupervised learning

In supervised learning, we know in advance the correct answer when we describe our model, while in reinforcement learning we define a measure, or reward, for the specific actions performed by the agent. In unsupervised learning, on the other hand, we are dealing with unlabelled data or data from the unknown structure. Using unsupervised learning techniques, we are able to observe the structure of our data, to extract meaningful information from them without being able to rely on the guide nor a variable known relative result, nor a reward function. Clustering is an exploratory technique of data analysis that allows us to organize a series of information within meaningful groups (*cluster*) without having any previous knowledge of memberships in such groups. Each cluster that can be derived during the analysis defines a group of objects that share a certain degree of similarity, but which are more dissimilar than the objects present in the other clusters, which is why clustering is sometimes called *"unsupervised classification"*. Clustering is an excellent technique for structuring information to identify meaningful relationships in the data. [1]
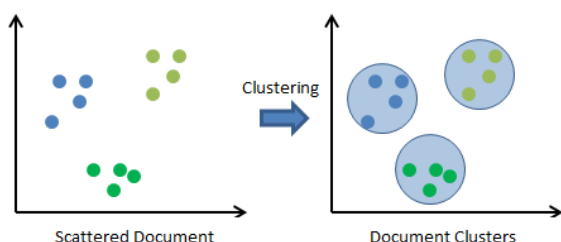

Fig. 3: example of clustering

II. Deep learning

Deep learning lies at the heart of the most advanced machine learning solutions, such as those that have learned to recognize items and images, determine the sentiment of text, and drive vehicles. These neural networks are complex and can be challenging to build, but Keras removes much of the effort. Keras acts as an API[1], letting us quickly create a network that might take hours or days to hand code in Python or other languages.

A. Introducing Keras

Let's use the definition from the documentation at keras.io. Keras is a high-level neural network API,

written in Python and capable of running on top of *TensorFlow*, *CNTK* or *Theano*. [3] That is, it lets us build neural networks easier by providing us with a high-level set of constructs. These constructs handle much of the plumbing involving in wring up neural networks and thus reduce programming errors. Also, as an API, it provides an interface that we can develop against and a detailed description of what happens when we invoke various objects and methods. Keras is Python centric in its code and is implemented as a Python library. It is imported and used just like any other Python library you might be familiar with, so the learning curve is minimal. Finally, Keras runs on top of *TensorFlow*, *CNTK*, or *Theano*. These are three of the most widely used libraries for performing work with neural networks. Keras calls these libraries to perform the actual execution of operations that create, populate, train, and evaluate the neural networks we specify in Keras. Keras utilizes either TensorFlow, CNTK, or Theano as the *backend*[2]. Keras itself does not create or execute the neural network. Rather, Keras defines an API we code against. In our code we invoke Keras methods and pass the appropriate parameters. Keras evaluates these for correctness and constructs whatever objects are required. Keras then calls the appropriate backend methods to do the actual neural network operations, such as defining structures, training the neural network model, and evaluating the trained model. Any result from these operations are returned by the backend to Keras, which processes them and returns to our code the appropriate results.

III. Neural Networks

The neural network, in figure 5, where on the left we have the input layer, which feeds data into the network. This data could be values from a data table, images from a camera, sounds from a recording, or output from a sensor. The input layer does not change the data, it simply passes it for processing by the remaining layers. The data from the input layer is passed to another layer of neurons. These layers can be of different types with the different layer types performing different transformations on the data as required by our solution. In a simple network, the input layer can be directly connected to an output layer of neurons, which provide the final outputs.

---

[1]An Application Programming Interface (API) provides an abstraction for a problem and specifies how clients should interact with the software componentes that implement a solution to that problem. [2]

[2]In software engineering, the terms front end and back end refer to the separation of concerns between the presentation layer (front end), and the data access layer (back end) of a piece of software, or the physical infrastructure or hardware. In the client–server model, the client is usually considered the front end and the server is usually considered the back end, even when some presentation work is actually done on the server. [4]
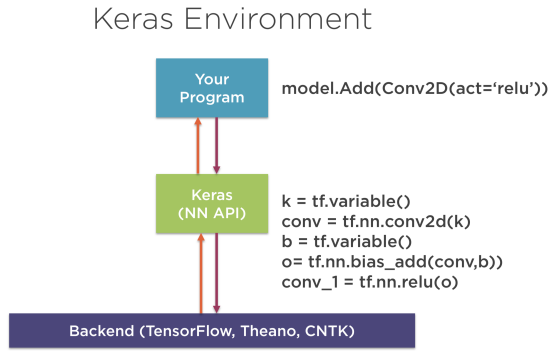
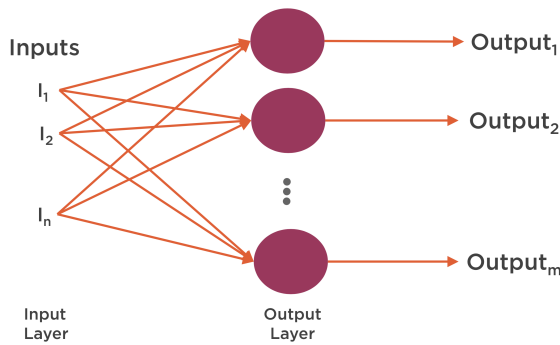## Keras Environment

Fig. 4: Keras enviroment

Fig. 5: Neural networks layers

But in most networks, the input layer is connected to hidden layers. Hidden layers are defined as not being input or output layers, and therefore, are hidden to the code that's using the neural network. The network can have many hidden layers and if there are two or more hidden layers we call the network a deep neural network. In this learning process we adjust the structures
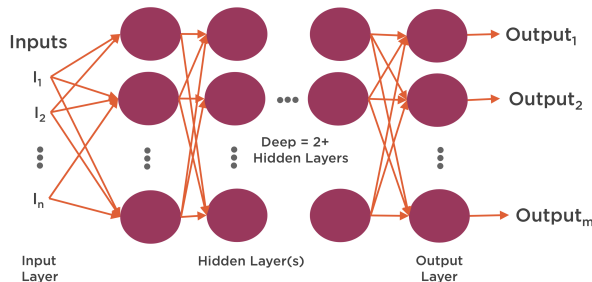

Fig. 6: Deep neural networks layers

of our neurons. To understand this, let's look at a single neuron. Here we see the inputs and outputs we showed in the network diagram, with the inputs going in and the outputs going out. However, what is not showing in this diagram is how the neurons work internally. To do that, we need to open the neuron so we can see how it is constructed. As we see, the neuron is performing a simple mathematics summing of weights times the input value and adding a bias. The product of these operations is passed through a non-linear activation function. And the output of the activation function is the output of the neuron. A key feature of the neural network is the ability to use the input data to train the weights and biases so the signal passed out of the neuron changes based on the input data. To do this training, we expose the network to data. With each set of data, an algorithm is used to adjust the weights and bias to minimize the error the network has in predicting the data's values. This is done through processes called forward propagation and back propagation. And when these processes are complete, the network is said to be trained, and the weights and biases of all the neurons have been adjusted to give the best results on the training data. So to summarize, the neural network consists of three layer types, input, hidden, and output. By passing training data through the network, the network of neurons is trained to give results with the least error. This training is done by adjusting weights and biases in the neurons, utilizing the processes of forward propagation and back propagation, and if there are two or more hidden layers in the network, we say it's a deep neural network.

IV. BUILDING CONVOLUTIONAL NEURAL NETWORK WITH KERAS

As we discussed in the last section (II-A), Keras has a series of layers that were specifically created to support convolutional neural networks. By definition, a fully connected layer connects every neuron in the layer and if it's a first layer it's connected to all the inputs. And all these connections create a problem known as the curse of dimensionality. That is, the more neurons and connections we have, the more weights we must train. This issue appears often, it is one of reasons that deep neural networks can take a long time to learn. To understand this issue a bit more, let's consider the case of working with images. Assume that we have an 8 megapixel image and we want to learn something from this image. To do that, we construct a network with a dense layer. Since every pixel in the image can contain unique data, they each contribute uniquely to determining the logic resulting from analysing the image. Therefore we need to connect each pixel to each neuron in our first layer. Let's say that we have 1000 neurons in

the first dense layer. In a dense we connect the data from each pixel to each neuron so we end up with 1000 neurons connected to 8 million pixel values, with each connection having its own weight that needs to be trained. That works out to 8 billion weights we have to train. And with most images it's even more. In a color image, each pixel has three colors. One each for red, green, and blue channels of the image. So there are actually 24 billion weights to train. Solving this explosion in the number of weights we have to train is one of the key reasons convolutional neural networks were developed.

A. How work CNNs

Convolutional neural network address our two concerns of working with image type data, namely many weights to train and being able to detect objects based on their general appearance rather than precisely matching an image. A lot of research has been performed when working with images and has resulted in subtle layers that you find in almost all convolutional neural networks. And these are the convolution layers you find in Keras. To understand the function of these layers, let's go over the structure of a convolutional neural network, which classifies objects and images. Let's walk through this diagram so we can get an overview of how convolutional neural networks work and how we implement them using Keras. We see the image data is passed through the convolutional layer, then through a non-linear ReLU activation and then to a pool layer. And then to a second convolution with ReLU layer and a second pool layer. And finally, to a fully connected layer and then to a second fully connected layer for classification. So we can divide our convolutional neural network into four operations. Convolution, non-linearity, you see via ReLU, pooling, and classification. You will find these four operations in almost all convolutional neural networks. Let's look at each operation in detail so we understand how to set parameters and pass data when we construct our convolutional neural network in Keras. Convolutional neural networks are the current state-of-art architecture for image classification. They are used in practice today in facial recognition, self driving cars, and detecting whether an object is a hot-dog.

B. Convolution

A convolution consists of a kernel (green square above), also called filter, that is applied in a sliding window fashion to extract features from the input. This filter is shifted after each operation across the input by an amount called strides. At each operation, a matrix

multiply of the kernel and current region of input is calculated. Filters can be stacked to create high-dimensional representations of the input. There are two ways of
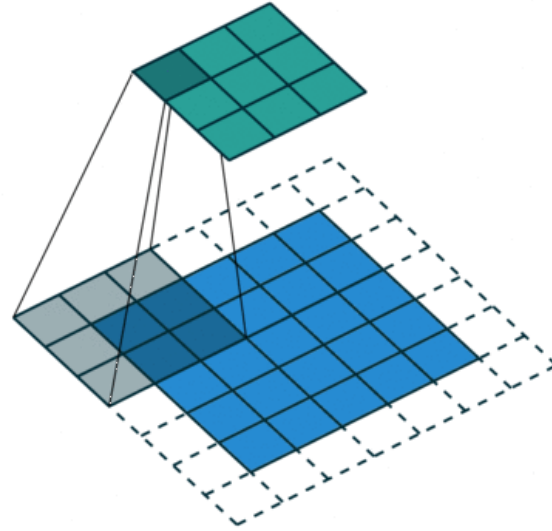

Fig. 7: Deep neural networks layers

handling differing filter size and input size, known as same padding and valid padding. Same padding will pad the input border with zeros (as seen above) to ensure the input width and height are preserved. Valid padding does not pad. Typically, you will want to use same padding or you will rapidly reduce the dimensionality of your input. Finally, an activation function (typically a ReLU) is applied to give the convolution non-linearity. ReLUs are a bit different from other activation functions, such as sigmoid or tanh, as ReLUs are one-sided. This one-sided property allows the network to create sparse representation (zero value for hidden units), increasing computational efficiency.

C. Pooling

Pooling is an operation to reduce dimensionality. It applies a function summarizing neighbouring information. Two common functions are max pooling and average pooling. By calculating the max of an input region, the output summarizes intensity of surrounding values. Pooling layers also have a kernel, padding and are moved in strides. To calculate the output size of a pooling operation, you can use the formula:

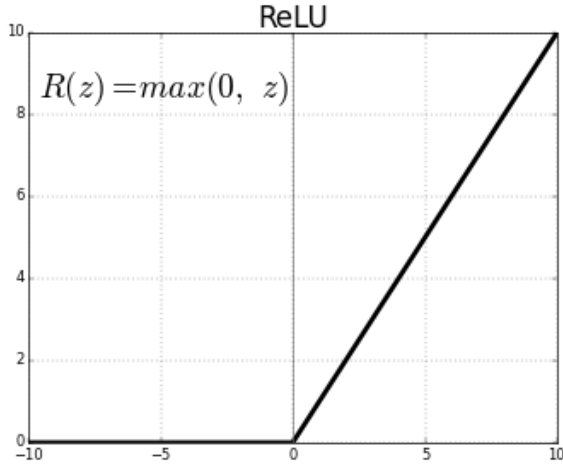$$\text{Input Width} - \text{kernel width} + 2 * \text{padding}/\text{strides} + 1 \quad (1)$$

4
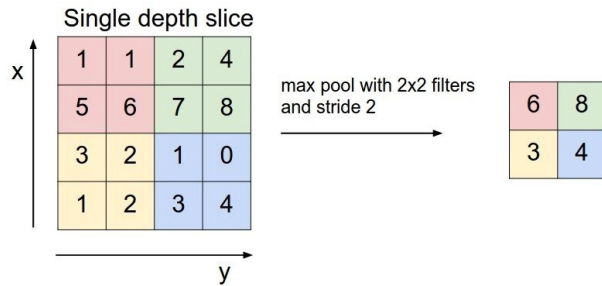
Fig. 8: Deep neural networks layers

Fig. 9: Deep neural networks layers

D. Fully Connected Layer

Fully connected layers you are likely familiar with from neural networks. Each neuron in the input is connected to each neuron in the output; fully-connected. Due to this connectivity, each neuron in the output will be used at most one time.

$$\sum_{i=1}^{n} x \cdot W + b \qquad (2)$$

In a CNN, the input is fed from the pooling layer into the fully connected layer. Depending on the task, a regression or classification algorithm can be applied to create the desired output.

V. THE PROJECT

The goal of this project is the rapid development of a neural network for low power systems, hence the need to resort to a technique known as 'fine-tuning' for the realization of our neural network. In general, if our dataset is not drastically different in context from the dataset which the pre-trained model is trained on, we
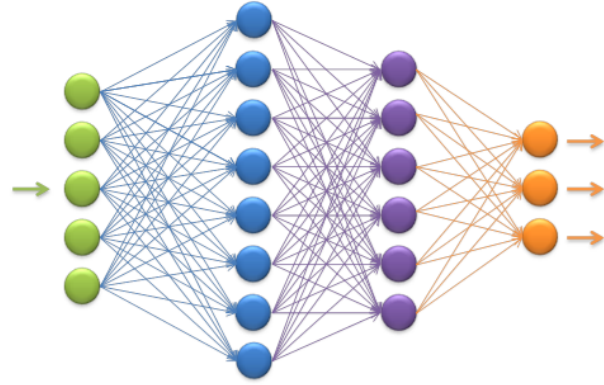

Fig. 10: Deep neural networks layers

should go for fine–tuning. Pre-trained network on a large and diverse dataset like the *ImageNet* captures universal features like curves and edges in its early layers, that are relevant and useful to most of the classification problems. Of course, if your data set represents some very specific domain, we should then consider training the network from scratch. One other concern is that if our dataset is small, fine-tuning the pre-trained network on a small dataset might lead to over-fitting, especially if the last few layers of the network are fully connected layers, as in the case for *VGG* network. If we have a few thousand raw samples, with the common data augmentation strategies implemented (translation, rotation, flipping, etc.), fine-tuning will usually get us a better result.

A. Fine-tuning Techniques

Some general guidelines for fine-tuning implementation:

- The common practice is to truncate the last layer (softmax layer) of the pre–trained network and replace it with our new *softmax* layer that are relevant to our own problem. For example, pre-trained network on *ImageNet* comes with a *softmax* layer with 1000 categories.

If our task is a classification on 10 categories, the new *softmax* layer of the network will be of 10 categories instead of 1000 categories. We then run back propagation on the network to fine-tune the pre–trained weights. Make sure cross validation is performed so that the network will be able to generalize well.

- Use a smaller learning rate to train the network. Since we expect the pre–trained weights to be quite good already as compared to randomly initialized weights, we do not want to distort them too quickly

5

and too much. A common practice is to make the initial learning rate 10 times smaller than the one used for scratch training.

- It is also a common practice to freeze the weights of the first few layers of the pre-trained network. This is because the first few layers capture universal features like curves and edges that are also relevant to our new problem. We want to keep those weights intact. Instead, we will get the network to focus on learning dataset-specific features in the subsequent layers.

VI. FINE-TUNING IN KERAS

I have implemented starter scripts for fine-tuning convnets in Keras. The scripts are hosted in my remote repository[3] page. Implementations of VGG16, VGG19, Inception-V3, and ResNet50 are included. With that, you can customize the scripts for your own fine-tuning task. Below is a detailed walk through of how to fine-tune VGG16 and Inception–V3 models using the scripts.

A. Datatset

Particular attention is needed in the construction of a good training dataset, in fact, as seen before in (I-A), we deal with a supervised learning where we know the response of our labels. A script is provided that can build a dataset divided into folders: training, validation and testing; as you can see a figure 11. A large number of figures per sample that clearly highlights the characteristics that you want to study allows a greater rate of success of the training preventing the *overfitting*. On the other hand, a large number of is not always achievable, thus using some augmentation techniques, it virtually allows to increase the observability of a images, for example, by rotating, distorting and translating it. The two train and validated folders are essential for the addition of the neural network. Instead, the test folder contains a set of images that the network has never seen and so necessary to measure the degree of confidence acquired in the network. The choice of using a structure history derives from some methods provided by the Keras API, in fact these methods used in the script guarantee greater efficiency in the management of the network training process. In fact, thanks to these it is possible to automatize the processes:

- crossing the folder structure constituting the dataset;
- creation of labels within the network;
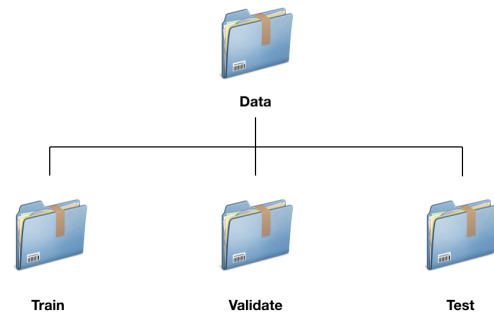- efficient memory management during the training phase;

Fig. 11: Dataset structure

- randomness of the set of samples processed;
- efficient memory management during the validation phase;
- randomness of the validated sample set;

B. Fine-tune VGG16

VGG16 is a 16-layer Covnet used by the Visual Geometry Group (VGG) at Oxford University in the 2014 "ILSVRC" (*ImageNet*) competition. The model achieves a 7.5% top five error rate on the validation set, which is a result that earned them a second place finish in the competition. For the first experiment, the network
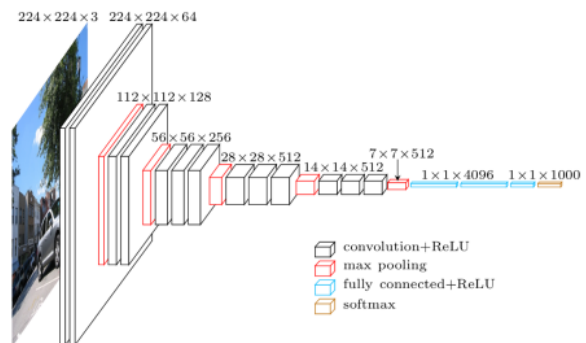

Fig. 12: Schematic Diagram of VGG16 model

was trained with a dataset from the Kaggle[4] specialist site. Withdrawing a binary dataset[5], which consists of only two classes: "dog" and "cat". After using the script, presented in section VI-A, the network is trained, then fine-tuning the model by minimizing the "*categorical cross entropy*" loss function using *stochastic gradient*

---

[3]https://github.com/frank1789/NeuralNetworks

[4]https://www.kaggle.com

[5]https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data

descent (SGD) algorithm. Notice that we use an initial learning rate of 0.0001, which is smaller than the learning rate for training scratch model usually 0.01. It was decided to use this optimization because it has better performance during the binary classification. In fact, after two hundred epochs an accuracy of about 99.96% is reached. After it is done, we use the model the make prediction on the validation set and return the score for the cross entropy loss, as shown in the figure 13.
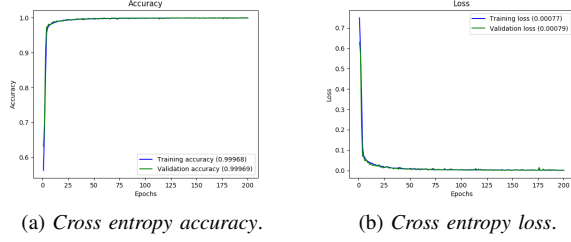

(a) *Cross entropy accuracy.* (b) *Cross entropy loss.*

Fig. 13: VGG16 binary label training result

The second experiment was instead performed on the Simpson[6] dataset, the famous animated television series, this dataset presents forty-two classes, so instead of using an optimization based on SGD it was preferred to use an optimizer *Adam* that presents better performance in case of multi–label. In this other case, after two hundred epochs an accuracy of about 99% is reached. Otherwise prediction on the validation set return the score for the cross entropy loss, as show in figures 14.
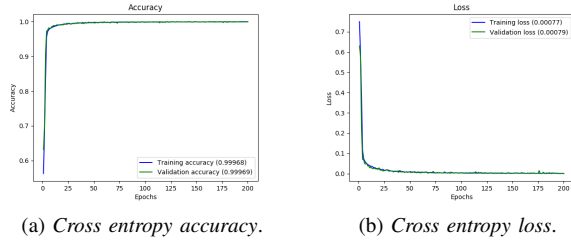

(a) *Cross entropy accuracy.* (b) *Cross entropy loss.*

Fig. 14: VGG16 multi–label training result

C. Fine-tune Inception–V3

Inception–V3 achieved the second place in the 2015 *ImageNet* competition with a 5.6% top five error rate on the validation set. The model is characterized by the usage of the Inception Module, which is a concatenation of features maps generated by kernels of varying dimensions. As described, for the neural network VGG16 in

---

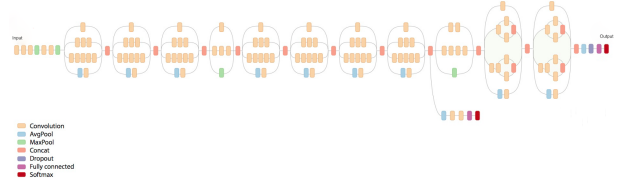[6]https://www.kaggle.com/alexattia/the-simpsons-characters-dataset

Fig. 15: Schematic Diagram of Inception–V3 model

the section (VI-B), the dual experiment was repeated for this neural network. In this way I obtain, with respect to the previous one, the results shown in figures for the binary classification. On the other side the reuses for multi-label classification where an accuracy of 99% is observed.
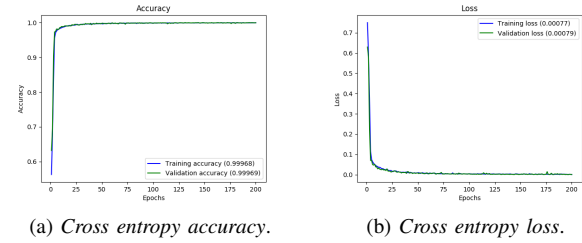

(a) *Cross entropy accuracy.* (b) *Cross entropy loss.*

Fig. 16: Inception binary label training result

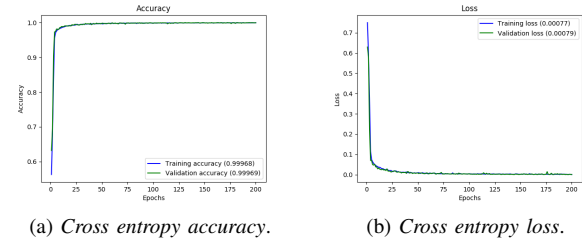(a) *Cross entropy accuracy.* (b) *Cross entropy loss.*

Fig. 17: Inception multi–label training result

VII. Intel Movidius Neural Compute Stick

Today, low-power consumption is indispensable for unmanned vehicles and IoT (Internet of Things) devices. In order to develop deep learning inference application at the edge, we can use Intels both energy efficient and low cost Movidius USB[7] stick, in figure 18. Movidius Neural Compute Stick (NCS) is produced by Intel and it can be run without any need of Intenet. This software development kit enables rapid prototyping, validation,

---

[7]Universal Serial Bus

and deployment of deep neural networks. Profiling, tuning, and compiling a DNN on a development computer with the tools are provided in the Intel Movidius Neural Compute SDK. The Movidius NCS compute capability comes from Myriad 2 VPU[8]. Intel Movidius VPUs drive the demanding workloads of modern computer vision and AI applications at ultra-low power. By coupling highly parallel programmable compute with workload-specific hardware acceleration, and co-locating these components on a common intelligent memory fabric, Movidius achieves a unique balance of power efficiency and high performance. Movidius technology allows device makers to deploy deep neural network and computer vision applications in categories such as smart-phones, drones, intelligent cameras and augmented reality devices. Running Deep Learning models efficiently on low capacity graph processors is very painful. Movidius allows us to optimize the operation of large models such as GoogLeNet. with multi-use support. It is an easy-to-use kit that allows you to design and implement applications such as classification and object recognition as physical products. We can simply think of Movidius NCS as a GPU[9] running on USB. However, training of the model is not performed on this unit, the trained model works optimally on the unit and is intended to be used in physical environments for testing purposes.

Fig. 18: Intel Movidius in package

A. Inside Intel Movidius

Movidius provides the ultimate in low-power vision processing solutions, which include the Myriad 2 family

[8]Vision Processing Unit
[9]Graphics Processing Unit

of vision processing units (VPUs) plus a comprehensive Myriad Development Kit, a reference hardware EVM and optional Machine Vision Application Packages. The Myriad 2 MA2x5x family of system-on-a-chip (SoC)[10] devices offers significant computation performance and image processing capability with a low-power footprint. The Myriad 2 line up includes the following product configurations:

- MA2150: 1 Gbit DDR
- MA2155: 1 Gbit DDR and secure boot
- MA2450: 4 Gbit DDR
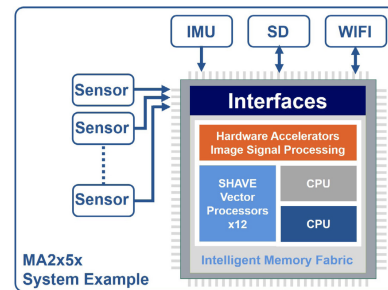- MA2455: 4 Gbit DDR and secure boot

Fig. 19: System example

Myriad 2 VPUs offer TeraFLOPS (trillions of floating-point operations per second) of performance within a nominal 1 Watt power envelope. The Myriad 2 architecture, show in figure 20, includes enough performance to support multiple cameras with flexible image signal processing pipelines for each camera, and software programmable vision processing with fixed- and floating-point data-types supported. A robust overall data-flow design ensures mitigation of processing bottlenecks.

Myriad 2 MA2x5x incorporates an innovative approach to combine image signal processing with vision processing. A set of imaging/vision hardware accelerators supports a world-class ISP pipeline without any round trips to memory; at the same time they are re-purposed to accelerate developers' vision processing algorithms in conjunction with a set of special purpose VLIW vision processor cores. All processing elements are tied together with a multi-ported memory that enables implementation of demanding applications with high efficiency. [5]

MYRIAD 2 SoC SPECIFICATIONS

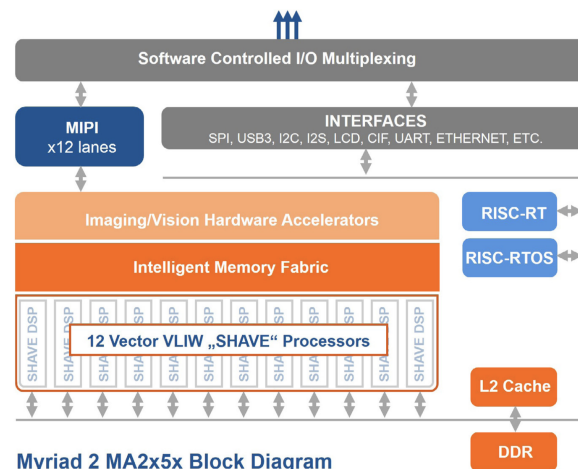- Heterogeneous, high throughput, multi-core architecture based on:

[10]System on Chip

- 12 VLIW 128-bit vector SHAVE Processors optimized for machine vision
- Configurable hardware accelerators for image and vision processing, with line-buffers enabling zero local memory access ISP mode
- 2 x 32-bit RISC processors
- Supports data and task parallelism
- Programmable Interconnect
- Support for 16/32-bit floating point and 8/16/32-bit integer operations
- Homogeneous, centralized memory architecture
- 2MB of on-chip memory
- 400 GB/sec of sustained internal memory bandwidth
- 256 KB of L2 Cache
- Power management: 20 power islands; low power states
- Nominal 600 MHz operation at 0.9 V
- Rich set of interfaces:
  - 12 Lanes MIPI, 1.5 Gbps per lane configurable as CSI-2 or DSI
  - I2C, SPI for control and configuration
  - I2S for audio input
  - Banks of configurable GPIO, PWM
  - USB3 with integrated PHY
  - 2-Slot SDIO
  - Debug interface
  - 1 Gbit Ethernet
- Available package configurations:
  - MA2150/MA2155: 6.5mm x 6.5mm 0.4mm pitch, 225 Ball BGA 1Gb LPDDR II
  - MA2450/MA2455: 8mm x 9.5mm 0.5mm pitch, 270 Ball BGA, 4Gb LPDDR III
- Advanced low-power 28nm HPC process node

B. Result

Fig. 20: MYRIAD 2 SoC architecture

REFERENCES

[1] S. Raschka, *Machine learning con Python. Costruire algoritmi per generare conoscenza*, ser. Guida completa. Apogeo, 2016. [Online]. Available: https://books.google.it/books?id=G7OvDAEACAAJ
[2] M. Reddy, *API Design for C++*. Elsevier Science, 2011. [Online]. Available: https://books.google.it/books?id=IY29LylT85wC
[3] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://keras.io
[4] A. Butterfield, G. Ngondi, and A. Kerr, *A Dictionary of Computer Science*, ser. Oxford Paperback Reference. Oxford University Press, 2016. [Online]. Available: https://books.google.it/books?id=GDgICwAAQBAJ
[5] Intel, "Movidius," 2016. [Online]. Available: https://www.movidius.com/myriad2