Comparison of Tensorflow and PyTorch in Convolutional Neural Network - based Applications

Mihai Cristian CHIRODEA
Master Student at Management in Information Technology, Electrical Engineering and Information Technology Faculty
University of Oradea
Oradea, Romania
m.chirodea@gmail.com

Ovidiu Constantin NOVAC
Department of Computers and Information Technology, Electrical Engineering and Information Technology Faculty,
University of Oradea
Oradea, Romania
ovnovac@uoradea.ro

Cornelia Mihaela NOVAC
Department of Electrical Engineering, Electrical Engineering and InformationTechnology Faculty,
University of Oradea
Oradea, Romania
mnovac@uoradea.ro

Nicu BIZON
Faculty of Electronics, Telecommunication and Computer Science
University of Pitesti
Pitesti, Romania
nicubizon@yahoo.com

Mihai OPROESCU
Faculty of Electronics, Telecommunication and Computer Science
University of Pitesti
Pitesti, Romania
moproescu@yahoo.com

Cornelia Emilia GORDAN
Department of Electronics and Telecommunications, Electrical Engineering and Information Technology Faculty,
University of Oradea
Oradea, Romania
cgordan@uoradea.ro

2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI) | 978-1-6654-2534-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/ECAI52376.2021.9515098

***Abstract*— In this paper, we present a comparison between the PyTorch and TensorFlow environments, used in defining neural networks. The purpose is to find whether the choice of a library affects the overall performance of the system both during training and design. To do so, our approach involves analysis of the processes involved when creating a neural network, as well as taking measurements and monitoring its evolution over the epochs. Advantages and disadvantages are then extracted from the results and are then used to draw conclusions.**

Keywords—Convolutional Neural Network (CNN); TensorFlow; PyTorch; network training; network design;

I. Introduction to Neural Networks

A neural network represents a type of "Machine Learning" algorithm that tries to recreate the neural paths of a brain, in turn of mimicking its processing abilities [1].

The process through which this is achieved has at its core approximations of multiple mathematical functions that try to achieve a desired outcome. Such approximations are known as layers in the neural network and have a set number of variables ("weights") that influence them. There are multiple strategies that define the general process of updating the weights, but they mainly fall into three main categories, "supervised learning", "unsupervised learning" and "semi-supervised learning".

In the supervised approach the network is trained using pairs of input-output examples. In order to update the weights, a general error of the system ("loss") is calculated and it describes how accurate the network's output is compared to the actual output in the pair. This result is then used to calculate each update value and in turn leading to a good approximation of the function mapping inputs to the outputs. This method is highly dependent on the examples given to it as similar input-output pairs will lead to the resulting functions being tuned specifically to them resulting in unexpected outputs when different examples are given to the network.

The semi-supervised approach is similar to the previous as it also provides a set of input-output examples, however, unlike the last one, the number of examples is relatively small and is used in conjunction with a much bigger example set that has no output given. The labeled part is usually created manually, as a consequence of being unable to train on only unlabeled data and doing so, leads co considerable improvements.

The final category is the unsupervised approach. In it, the inputs are unlabeled and the system learns to find patterns in the input in order to classify it. This type of learning is usually employed in environments where data needs to be separated based on similarity.

Another important aspect that pertains to the issue of updating the weights is how much should the ouputs of each layer influence said updates. Simply updating them without a clear strategy would lead to the inability to reach a satisfactory solution as the values with which to update with would eventually become to big ("exploding gradients") or too small ("vanishing gradients"). To solve this, numerous strategies have been created, their purpose being to minimize network error, while keeping the gradients under control. These methods are known as optimizers and in order to minimize the loss of the system, they make use of a learning rate, which limits how much can the weights be updated with.

With all that in mind, creating a neural network from scratch would involve a lot of time and effort and because of this, libraries designed to wrap the essential components (i.e: layers, initialization strategies, loss functions, optimizers, etc.) into an easy to use interface have seen a lot of use during development. However, as more options have been made available over the years, it's become increasingly difficult to make a decision on which library to use on a given task. As such, a careful analysis needs to be performed to find whether choosing a library over another affects the overall performance of the system during design development, as well as during the training phase.

Therefore, the main objective of the paper is to perform a detailed comparison between two of the more popular neural network libraries, those being PyTorch and TensorFlow, in terms of training performance and network design. Additionally, to ensure the validity of the analysis, a secondary objective is also defined and pertains to the idea of removing the external factors that could influence the comparison.

Achieving those objectives revealed that some differences between the two libraries do exists and they have an influence over performance when designing and training a neural network. The process through which this results are obtained involves defining the comparison factors that represent important aspects from the two phases. Then, a short description of the two libraries is presented to highlight their strong points and weaknesses and lastly, both are compared side-by-side using the previously defined factors to extract the differences between them.

II. Comparing Neural Network Libraries

LabVIEW To provide a fair testing environment there are three main prerequisites that the developed platform needs to achieve. The first one is to provide the same format of input-output pairs to both libraries. Those pairs should also be generated from the same dataset to ensure that the same features are generalized during training. Regarding the second prerequisite, the system should define the same neural network structure and training strategies in both libraries to ensure that the training processes are as close as possible. Lastly, the tests should be run on the same hardware, to avoid any performance discrepancies that could arise due to a video card or CPU difference, which would invalidate the results.

Achieving these objectives will generate better overall results as the only remaining factors that could influence the outcome are the ones that the analysis focuses on and which are described in the following sub-sections.

A. *User-friendliness*

Refers to how easy it is for a user to describe their network using the library's functions. Without having any prior knowledge of how it works, he should be able to easily distinguish key functions such as network layers and loss strategies. This factor is important to the overall user experience, especially for those looking to create their first neural network application as it makes the individual components much clearer, hence why it is included in the analysis.

B. *Available Documentation*

The documentation that is made available for the library is another factor that influences user experience during design, however, unlike user-friendliness, this one is focused on both beginner lever users, as well as more experienced users that are transitioning from another library. Its role is to provide a gateway to all that the library has to offer, in a clear and concise manner, such that users can overcome the various difficulties that appear when learning how to use it. A well-documented framework will minimize the time spent researching for solutions on specific subjects, making the learning experience as well as the transitioning experience way easier on the user.

C. *Ease of Integration*

The last factor that is included in the analysis of the design phase is ease of integration, which pertains to how easy it is for an existing system to incorporate the new library into it. During development, an easy to add library will provide a more efficient way to switch from one framework to another, or, in the case where the system is currently being built, a way to create the neural network faster, leading to a longer refinement phase.

D. *Overall Training Time*

Overall trainning time refers to how much time does it take for the program to complete its training. This includes the average time it takes to obtain one output from the network, as well as the average time to complete one epoch of training. Measuring those times will offer great insight into how each library performs given the same hardware specifications.

E. *Overall Accuracy*

Overall accuracy is another important factor that influences system performance as having small training times, but low accuracy can be efficient in terms of speed but leads to the system not outputting the desired results, making the goal of the system unreachable in most cases. As such, average accuracy is also included in the analysis, so the system should provide a way to follow its development over the epochs.

F. *Execution Time During Evaluation*

Depending on the task of the system, the time it takes for a result to be generated during actual use, can heavily influence the viability of said system. For example, in a depth estimation system that is mounted on a car, a few seconds of delay can lead to the safety systems being deployed too late, possibly resulting in irreparable damage. Because of this, execution time can be the deciding factor that determines whether a library is chosen over another, hence why it is included in the analysis.

III. PyTorch

Launched in 2016, PyTorch is a library that achieves a balance between usability and speed by adhering to four main principles [2].

The first one is the familiarity with the pythonic way of defining simple interfaces for the modules it offers. This way, users already familiar with the Python programming language are very easily introduced to the multitude of tools that PyTorch offers.

The same approach of simplicity is made apparent with the way the complexity of machine learning is hidden, which serves as another principle and helps in easily identifying and utilizing neural network concepts.

Regarding performance, sacrificing speed for a simpler design is another core fundamental and is acceptable if the impact on performance is not substantial. This way, the library manages to remain competitive, while keeping a simpler, more user-friendly approach.

The same idea of keeping the API simple also provides the possibility of quickly adding or modifying functionalities, which is the last principle of PyTorch and with it, the library can keep up with new developments in the field of AI. This overall approach of simplicity to development ensures that any network architecture can be easily created as layers or even losses, be they custom or not, are easily described in a class format.

IV. TensorFlow

TensorFlow version 1.0 was first released in 2015, its purpose being the ability to execute any machine learning algorithm on a variety of platforms [3]. This approach led to a lot of flexibility from a development standpoint as a lot of tools that could result in the same outcome were made available to the user. As an example, a neural network model could be created either from using Contrib, Keras, or even defining them manually via layers. However, for a beginner user, the fact that there were multiple options to choose from meant that a lot of confusion was created when starting to develop on the library. Another drawback for this decision was the fact that in order to train the network, a separate code that created a session needed to be developed. In doing so, debugging was quite difficult as mistakes now had to be searched both during definition and execution, leading to the library not being very user-friendly.

All the shortcomings described above were solved when TensorFlow 2.0 was released in 2019 [4]. The new version brought major overhauls that aimed to bring the library in line with the other options available on the market. First, multiple libraries were removed from the framework with the intent of diminishing the number of options a user had when developing a model, in turn eliminating the confusion that was created. Next, the two-step approach of building and executing using sessions was reworked, with the execution step now being automatically called, leading to a one step building process that was much more traceable. Adding to that, the library also offered a more "Pythonic" way of defining neural networks through classes, like was the case for PyTorch, making TensorFlow more user-friendly in the process. Another new feature was the abstraction of the model, leading to the possibility of running the same defined system on a local machine, or on a multi-GPU server environment. Lastly, the way input data is handled was also significantly improved by defining a simpler pipeline. Originally, to feed data into the network, dummy variables needed to be created, which would then be filled with real data during session execution. This again created confusion, so it was replaced with the TensorFlow Datasets module which provided an easier way to use and even add data.

With all the new changes, TensorFlow now offers a similar experience to PyTorch, that being a simple, beginner-friendly, interface, while at the same time being efficient in terms of performance.

V. Comparing PyTorch and TensorFlow

As previously stated in Chapter II, comparing the two libraries first requires the system to provide the same neural network architecture, the same input data format as well as the same hardware specifications.

Starting with the first requirement, the defined network is inspired from the work of Zbontar & LeCun [5] in which they described an architecture meant to be used in depth estimation applications. Based on a stereo image pair of the same size, with one image being taken from a right perspective, while another from a left perspective and both describing the same scenery, the network outputs a matrix corresponding to the size of the images that describes the depth of objects in the image. The values contained are bounded between 0 and 255, each position representing a distance, with smaller values meaning the object is farther away, while bigger ones describing the object as being closer. One such value is obtained by comparing a pixel in the left image with neighboring pixels from the right image, the intent being to identify its true correspondent and to do so, the network outputs a set of similarity values between pixel pairs, which are then compared to output the depth matrix.

To achieve a mapping from a stereo image pair to similarity matrix, the network uses four convolutional layers, with three of them being followed by a ReLU (Rectified Linear Unit) activation function. Each image is passed through these layers and the resulting values are combined using a Cosine Similarity measure to obtain the similarity matrix (Fig. 1).

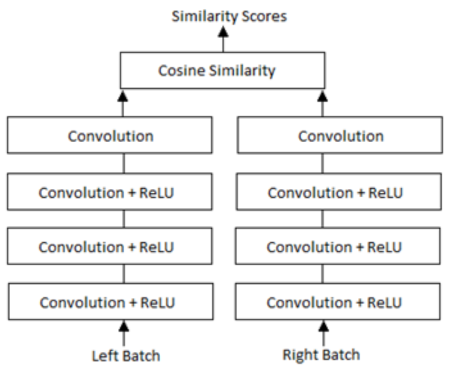

Fig. 1. The architecture of the network.

Regarding the second requirement, training images are taken from the KITTI 2015 dataset [6], which consists of 200 stereo images along with their corresponding true depth. An algorithm used by both libraries then processes said images by calculating their standard deviation (std) and mean, followed by dividing them into multiple 9x9 windows from which said mean and std are subtracted. The last two operations are necessary to normalize the images as the values are brought into a much more manageable range, as well as being centered at the same value, resulting in the gradients of the network being kept in similar ranges and under control. Implementing those steps also had the effect of multiplying the total number of examples in the input dataset from the original 200 to approximately 17 million, which allowed for better control over how they are passed to the network as well as offering more diversity and consequently, better generalization. For evaluation, a different dataset had to be used to present the network with new images that were not previously trained on as it could influence the results. To do so, the 2012 version of the KITTI dataset was used as it provided the same format as the one used during training, leading to no additional implementation changes being needed between the training and evaluation modes.

Lastly, the system was developed on a computer running Windows 10 Pro version 20H2 and used PyCharm Professional Edition with Anaconda plugin, version 2020.1.3 as its IDE. The system was also implemented in such a way that the libraries could be interchanged, allowing it to be run on the same hardware, without requiring another project to be created. While this approach made the project easier to manage, it also added the complexity of having to install two network libraries on one project, which meant that little help could be found online if problems arose.

Regarding hardware specifications, the system was tested on the following:

- **CPU**: AMD Ryzen 7 1700X, 3.80 GHz
- **GPU**: ASUS GeForce GTX 1070 Ti A8G, 8GB GDDR5
- **RAM**: 16 GB 3200 MHz

- **STORAGE**: 500GB SSD

With the steps described above defined, the tests were then performed by carefully analyzing each framework, following the guidelines described in Chapter 2.

A. *User-friendliness in PyTorch vs TensorFlow*

Building the network and training step in each library meant that an optimizer, loss, scheduler, and model had to be defined. Both PyTorch and TensorFlow offered solutions to implement those components and were generally straight forward.

First, the model and loss in both cases were custom ones, so their implementation relied solely on how the data structures were defined. For the network models, both libraries offered the possibility to build using the predefined Convolutional layers in a class, by extending the default module. Doing so, the complexity of the architecture was hidden by a simple to access interface, requiring only 2 methods for passing data and an initialization. Regarding the loss function, as it only required the result from the network, the libraries had little effect on the implementation, leading to the two modules becoming nearly identical.

The same outcome was obtained when defining the optimizer as in both cases, it was a simple class instantiation. However, in the case of the scheduler, the two libraries opted for different methods. The design required an implementation which given one or more epoch boundaries, would change the learning rate of the optimizer by multiplying the last value with 0.1. Doing so, the network would train faster in the beginning stages and slowdown in the latter epochs, avoiding an overfit. When using PyTorch, this requirement was accomplished using a simple initialization of the MultiStepLR class, while on TensorFlow, even though the default schedulers were defined using the same strategy, the current version did not contain a compatible implementation to the required structure. Because of that, a custom scheduler had to be defined through extending the default module.

Lastly, the implementation for the training step again presented some differences in the execution strategies employed. PyTorch offered a straightforward flow of data, from passing data through the network and computing the loss to updating the parameters using the backwards and optimize functions. Doing so, each important step was covered and clearly highlighted by each function call, making it very easy to identify each role. When implementing the same in TensorFlow, the same strategy of presenting a clear role for each function was present, however, to a lesser degree. While the main flow was the same and could be traced, the function calls themselves were more complex especially for generating the gradients.

B. *Documentation Support*

During development, the available documentation for each library was used as a support for dealing with the problems that arose due to user error. Both offered plenty of solutions that were found with minimal effort and clearly explained how the components worked and how they should be used. There was only one instance where this was not the applicable, and it happened when using TensorFlow to define the scheduler, resulting in a solution not being found so easily. This case was however dismissed as the library did not have by default the required behavior in any of the schedulers made available, hence it could not be found.

C. *Integration with the Developed Platform*

As the system was defined with shared data inputs and outputs, the two libraries needed to be integrated in a such a way that they would not interfere with each other during usage and that the user had control over what library to use. To do so, the defined classes were organized by category and where this was not possible, a condition was used to decide which one should take priority.

With all the above and since both libraries offered easy ways to initialize and use the components, no problems regarding integration were found during implementation.

However, the installation of each library presented different results. While PyTorch did not have a lot of external requirements as most were included with its installation, the same could not be said about TensorFlow. The main problem that appeared pertained to the external requirements which were a lot more difficult to meet. Installing CUDA to use the GPU proved to be the most time consuming as even though there were explicit tutorials on the library's website, they were focused on adding it locally, which did not help in this case as PyTorch already provided it via pip. Moreover, even when the link was successfully added, the TensorFlow version available on Windows was lower that what was available for Linux, hence it was not updated to support the latest CUDA, leading to an incompatibility between the two libraries. A solution was found in the end; however, it required a lot of time and effort to find as neither documentation had any information regarding this case.

D. *Training & Execution Times*

Regarding training times, multiple measurements were taken for each library during training, at different checkpoints in the system. The main strategy used to get said measurement differs depending on the input data, however, it generally consists of timestamps being taken before and after a point of interest and then using them to calculate how much time has passed, with the intent of determining if any differences exist between the two libraries.

With all the above, three checkpoints were defined, with the first being the total training time, which measured the duration for the whole training cycle. During implementation, timestamps were taken at the beginning and the end of the main epoch loop, the result then being used directly in the comparison.

Using the same principles, the next measurement was the time taken for one epoch to be completed. In this case, the timestamps were situated at the beginning and end of each epoch, with each measure then being averaged to obtain a general result.

The final training time was the time it took for data to be run through the neural network and optimization step. As in the previous case, an average was computed at the end, using as input the timestamps of each run.

For execution time, the same approach as total training time was used, with the result representing how much time was spent generating the output depth map, using a pair of images as input.

All the obtained times were collected in TABLE 1, which shows a side-by-side comparison between PyTorch and TensorFlow for each category taken under consideration. Times are converted in hours, minutes, seconds format, except for network and optimization times, which are represented in seconds only.

As the illustration shows in Table I, PyTorch is faster during execution in all categories, with a 25.5% difference in total training time, 26.1% in average epoch times, 25.1% in network and optimization times and 77.7% when outputting a depth map, compared to TensorFlow.

TABLE I. EXECUTION TIMES OF PYTORCH & TENSORFLOW

Measurement	PyTorch time	TensorFlow time
Total training time	16h 58m 51.032s	21h 56m 53.515s
Average epoch time	1h 10m 48.891s	1h 32m 6.028s
Network & Optimization time	0.0008693s	0.0011189s
Execution time	1.1743711s	2.6666322s

E. *Accuracy during Training & Execution*

During training, the evolution of each network was monitored using charts that mapped the respective accuracy to each epoch. The outcome revealed a similar progress path in both libraries, with the TensorFlow implementation having a lead of around 0.5% in accuracy, compared to PyTorch. Fig. 2 presents the charts in more detail.

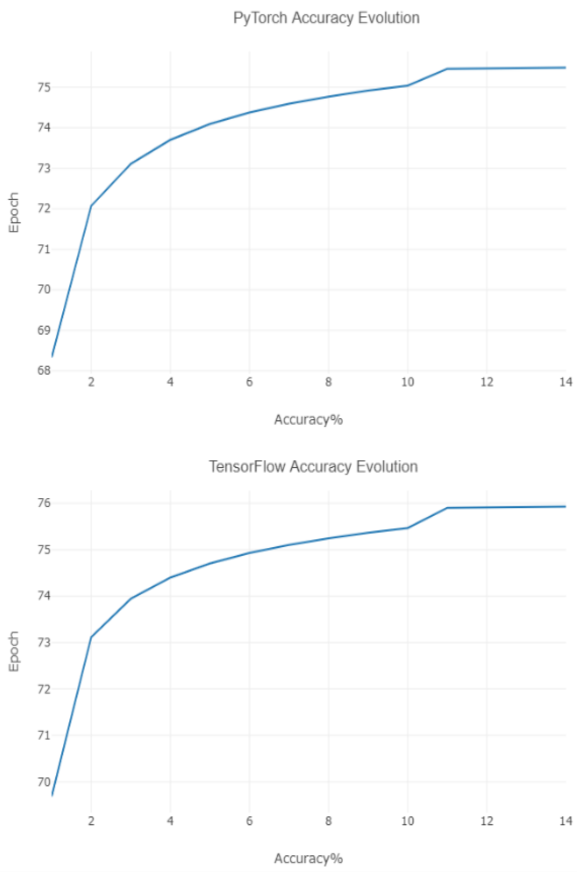

Fig. 2. Accuracy evolution over the epochs.

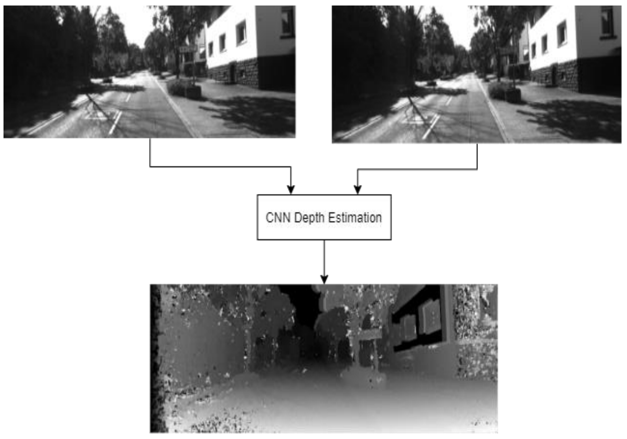

Fig. 3. Visual depth map obtained from the network.

Regarding execution mode, the tests performed involved obtaining a visual depth map of the input, an example of which can be seen in Fig. 3. To better represent the output of the network, error rate was used instead of accuracy, following the guidelines described by Zbontar & all [5]. The following set of equations detail how the new measurement was calculated:

$$E(i,j) = \begin{cases} 1, & \text{if } |Map(i,j) - Ground(i,j)| >= 3 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$error\_rate = \frac{\sum_{i=0}^{h-1} \sum_{j=0}^{w-1} E(i,j)}{total\_count} \times 100 \quad (2)$$

$$average\_error\_rate = \frac{\sum_{1}^{n} error\_rate}{count\_error\_rate} \quad (3)$$

As shown by the three equations, the error rate is first calculated by counting the positions where the value differs by more than 3 units compared to the true depth map (ground truth). The resulting number is then divided by the total number of positions in the map and multiplied by 100, resulting in the percentage of erroneous positions. Lastly, the average error rate over a dataset is then calculated using the third equation and the obtained result.

With all the above, two tests were performed on the trained networks, with the first monitoring the accuracy of a single output, while the second averaging the accuracy over the whole evaluation dataset. TABLE 2 shows the results of the execution, with the two libraries being compared side-by-side.

TABLE II. EXECUTION TIMES OF PYTORCH & TENSORFLOW

Test type	PyTorch error rate	TensorFlow error rate
Single image	11.6752832%	11.5439943%
Whole dataset	17.8751599%	17.6685158%

In both runs, TensorFlow maintained a lead, with the single image test resulting in a 1.13% difference, while the dataset test yielding 1.16%.

CONCLUSIONS

The comparison shows that there are quite a few differences between the two libraries.

While PyTorch offered a more beginner-friendly experience, TensorFlow 2.0 did not reach the same level as it presented a more complex dataflow and did not offer as many options when defining a scheduler but did allow for a custom one to be defined. TensorFlow was also harder to integrate in the system compared to PyTorch as it required more external dependencies and for this case, it also generated incompatibilities.

During training and execution PyTorch was faster than TensorFlow but had worse accuracy and both had very similar training paths.

These results reveal that the choice of library does affect the system during training and execution, however not to a degree that would make one a better choice than another. The remaining deciding factor is then user experience, with PyTorch being more oriented toward beginners, while TensorFlow offering more overall options at the cost of user friendliness.

In conclusion, different libraries do not affect system performance during training, however they have an effect during design which is directly proportional to user experience.

REFERENCES

[1] K. Gurney, An introduction to neural networks. CRC press, 1997.

[2] A. Paszke et all, “Pytorch:An imperative style, high-performance deep learning library,” 2019.

[3] M. Abadi et all, “Tensorflow: A system for large-scale machine learning,” 2016.

[4] https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8

[5] J. Zbontar and Y. LeCun, “Stereo matching by training a convolutional neural network to compare image patches,” The journal of machine learning research, vol. 17, no. 1, pp. 2287–2318, 2016.

[6] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, “Vision meets robotics: The kitti dataset,” The International Journal of Robotics Research, vol. 32, no. 11, pp. 1231–1237, 2013