Comparing TensorFlow and PyTorch for Image Recognition in NAO Robot Soccer

Vitor Amadeu Souza[1], **Hebert Azevedo Sá**[1]

[1]LIARC – Instituto Militar de Engenharia (IME)
22.290-270 – Rio de Janeiro – RJ – Brazil

`vitor.souza@ime.eb.br, azevedo@ime.eb.br`

Abstract. *The study compares TensorFlow and PyTorch in image recognition tasks within NAO robot soccer. Image classes were created and trained using data augmentation to enhance robustness and generalization. The analysis considered training time, classification accuracy, and adaptability to different lighting conditions and angles. The results showed that TensorFlow outperformed PyTorch, achieving higher accuracy and better adaptation to challenging scenarios, making it more suitable for computer vision in dynamic environments. The novelty of this study lies in evaluating these frameworks on the NAO robot's specific hardware, under realistic robotic conditions.*

1. Introduction

Convolutional Neural Networks (CNNs) have emerged as one of the leading techniques in deep learning for image recognition [LeCun et al. 2015, Krizhevsky et al. 2012]. These networks are particularly effective in tasks such as image classification and object detection, which are vital for a wide range of applications, including robotics. Two of the primary frameworks used to develop and train CNNs are TensorFlow and PyTorch [Abadi et al. 2016, Paszke et al. 2019]. Both frameworks are highly popular in the research and industry communities, providing unique features and optimizations for deep learning tasks. Several studies have conducted comparisons between these frameworks, evaluating their performance and suitability for various applications. Notably, previous works such as those by Florencio et al. [Florencio et al. 2019] and Dai et al. [Dai et al. 2022], along with other benchmark studies, provide valuable insights into the strengths and weaknesses of TensorFlow and PyTorch in image recognition tasks.

This paper presents a comparative analysis of TensorFlow and PyTorch, focusing on their performance in image recognition tasks in the context of NAO robot soccer [RoboCup Federation 2023, Wei et al. 2021]. Specifically, the study examines the creation and training of image classification models that identify key elements on the soccer field, such as the ball, opposing robots, and goalposts, from images captured during robot soccer matches. The use of data augmentation techniques is also explored as a means of enhancing the robustness and generalization of the models under varying conditions [Shorten and Khoshgoftaar 2019, Taylor and Nitschke 2018].

A key contribution of this work is the evaluation of TensorFlow and PyTorch specifically within the context of RoboCup robot soccer. In particular, it demonstrates that TensorFlow achieved superior performance compared to PyTorch, with better results in classification accuracy and adaptability to dynamic conditions, such as varying lighting and camera angles. These findings will be presented and discussed in detail throughout the paper. The study also explores the trade-offs between training efficiency and flexibility, with TensorFlow excelling in training time and performance optimization, while PyTorch offers more flexibility and ease of use, especially in research settings.

This comparison is essential for computer vision applications in dynamic environments, such as robot soccer, where real-time performance is crucial. The insights gained will assist in selecting the most suitable framework based on specific needs, considering training time, prediction speed, and model adaptability in real-world scenarios [Ren et al. 2017, Seidenari et al. 2012].

2. NAO Robot Overview

The NAO robot is equipped with two CMOS cameras, one superior and one inferior, which provide its computer vision capability [Gouaillier et al. 2011]. Fig. 1 demonstrates these cameras.

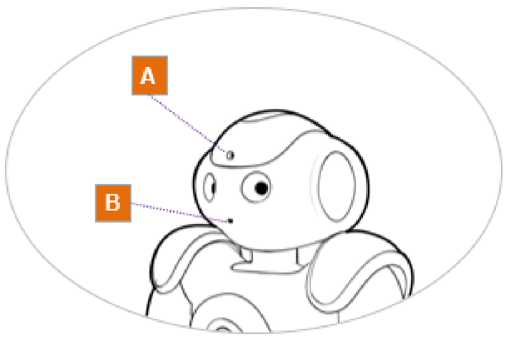

Figure 1. Superior and inferior cameras of NAO Robot. [Aldebaran 2024]

The accuracy of the Field of View (FOV) measurements is essential for image processing and object recognition on the field, as it allows developers to properly calibrate detection and tracking algorithms [Schwarz et al. 2019]. A detailed understanding of these technical parameters contributes significantly to the robustness of the robot vision system during soccer matches [Hartley and Zisserman 2004]. Fig. 2 illustrates the technical specifications of the camera's FOV, an essential element for the robot's vision system. These specifications are crucial for developing efficient computer vision algorithms [Aldebaran Robotics 2021].

The internal architecture of the NAO robot, presents a hybrid system composed of various integrated components. One of the key elements is the CMOS camera, which plays a fundamental role in the robot's visual recognition capability [Ivaldi et al. 2015]. The CMOS camera is connected to the CPU board processor, which is responsible for processing images and other sensory data. This processing is done by an Intel Atom E3845 processor and an ARM-7 microcontroller, which communicates with the various modules via I2C interfaces [Cui et al. 2013, Degallier et al. 2008].

The dsPIC modules control the motors and sensors distributed throughout the robot's body, such as the head, shoulders, elbows, hips, knees, and ankles. This hybrid architecture allows the NAO robot to perform a wide range of movements and interactions with its environment [Bhattacharya and Vidyarthi 2014, Seet et al. 2012].

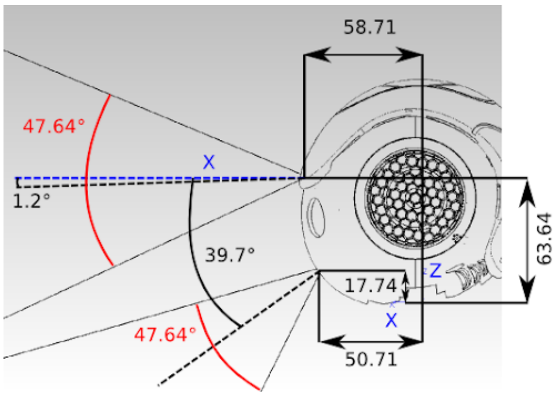

Figure 2. The presented measurements show a diagonal field of view of 47.64°. The camera has an effective viewing area extending 39.7° from the focal point. [NAO Vision]

Efficiency in visual recognition is critical for the performance of the NAO robot in robot soccer applications. The more precise and faster the computer vision system, the better the robot's ability to identify and react to objects, players, and the ball on the field [Seet et al. 2012, Fuke and Tamada 2010]. This directly reflects the robot's robustness and performance during matches.

Therefore, the comparison between deep learning frameworks like TensorFlow and PyTorch is relevant to this research, as it can provide important data on which approach is more suitable for optimizing the robot's computer vision system, improving its image recognition capability in a dynamic environment like robot soccer [Degallier et al. 2008, Bhattacharya and Vidyarthi 2014].

The results obtained offer a critical insight into the advantages and limitations of TensorFlow and PyTorch, helping in the selection of the most appropriate framework for computer vision applications in challenging environments, such as robot soccer [Seet et al. 2012, Fuke and Tamada 2010].

3. Dataset and Data Preparation

For the training and evaluation of the neural networks implemented in TensorFlow and PyTorch, the official dataset provided by the RoboCup Federation was used [Kitano et al. 1998]. This dataset contains images captured during official robot soccer matches, representing different lighting conditions, angles, and game scenarios [Asada et al. 2019]. The original dataset contained unlabeled images, which were manually analyzed and categorized into four main classes that represent the most frequently identified elements during a robot soccer match: ball, robot, post, and line [Röfer et al. 2019]. The distribution of images across the classes is presented in Table 1.

It is important to note that the "post" class has a significantly smaller number of images compared to the other classes. This is due to the limitation of the original dataset, which contained few captures of this specific element [RoboCup TC 2023].

Table 1. Image distribution by class in the dataset

Class	Number of Images
Ball	263
Line	267
Robot	375
Post	40

4. Methodology

Two source codes were written in Python for image recognition using TensorFlow and PyTorch. Both versions follow a similar structure for building and training a convolutional neural network (CNN).

The code implemented for TensorFlow employs a system to train and make predictions using a CNN aimed at classifying images into four categories: “ball”, “post”, “line”, and “robot”. The network is trained with images containing variations in lighting and noise. The process begins with importing essential libraries, such as TensorFlow, which is used to build and train the model, along with other libraries like os, time, matplotlib, and seaborn for file manipulation, runtime monitoring, and graphical visualization, respectively.

The neural network is built in the function `create_model()`, where three convolutional layers are defined, followed by pooling layers, along with two fully connected layers. The convolutional layers use 3×3 filters with the `ReLU` activation function and `same` padding, ensuring that the dimensions of the image remain constant after convolution. The pooling layers, in turn, apply a 2×2 window to reduce the dimensionality of the images. After the convolutional part, the network is flattened using a `Flatten` layer to prepare the input for the dense layers, which are responsible for performing the classification. The final layer uses the `softmax` activation function, suitable for multi-class classification problems.

For training, the code uses the `ImageDataGenerator` class from Keras, which applies data augmentation techniques such as rotations, horizontal and vertical shifts, brightness adjustments, cropping, zooming, and horizontal flipping to increase the variety of the training set and prevent *overfitting*. The `flow_from_directory` function is used to load the images from the training folder and perform the necessary preprocessing, such as resizing the images to 150×150 pixels and normalizing the pixel values.

Before starting the training process, the code checks if a pre-trained model already exists in the specified path. If the model has been trained previously, it is loaded, and its structure is displayed. Otherwise, the model is trained with the loaded data and, at the end, saved in a file for later use. The total training time is also recorded.

After training, the code generates a confusion matrix, which is an important tool for evaluating the model’s performance regarding its predictions. To do this, the model makes predictions on the training dataset, and the matrix is generated by comparing the actual classes with the predicted ones. The confusion matrix is displayed as a graph using the seaborn library for better visualization.

Finally, the code includes a function named `make_predication()` that allows making predictions on individual images, providing the predicted class and the associated confidence level. The function loads the image, performs the prediction, and displays the predicted class name along with the prediction's confidence. If the confidence is too low (below 50%), the model indicates that the prediction is "unknown". The execution time for the prediction is also calculated.

In the PyTorch code, the first part defines a class called `CNNModel`, which inherits from `nn.Module` in PyTorch. The model consists of three convolutional layers (`conv1`, `conv2`, `conv3`), each followed by a pooling layer to reduce the image dimensions. The convolutional layers use 3×3 filters and `ReLU` activation, while the pooling is done with a 2×2 window. The network also includes two fully connected layers (`fc1` and `fc2`), with the last one having 4 neurons, corresponding to the 4 output classes. The model is prepared to run on a GPU if available.

The second part of the code handles the training of the model. For this, the dataset is organized in the `dataset/train` folder using the `ImageFolder` library, which can load images from directories and associate them with their respective classes. Additionally, data augmentation transformations are applied, such as random rotation, horizontal flipping, brightness and contrast adjustments, and translation, to increase the variability of the images and prevent overfitting. The code uses PyTorch's `DataLoader` to load the data in batches and then trains the model for a specified number of epochs. At each epoch, the model performs forward propagation, calculates the error using the `CrossEntropyLoss` function, and adjusts its parameters using the `Adam` optimizer. At the end of each epoch, the loss and accuracy are displayed, and the model is saved in a file named `model_py.pth` if the training is completed.

The third part of the code implements the function named `make_predication()`, which allows classifying new individual images. The function loads the image, applies the same preprocessing transformations used during training, and makes predictions using the trained model. The confidence of the prediction is calculated using the `softmax` function, and if the confidence is below 50%, the class is considered "unknown". Otherwise, the predicted class and its confidence are displayed. The execution time of the prediction is also calculated to assess the efficiency of the model.

The code also includes the generation and display of a confusion matrix after training. To do this, the actual labels and the predictions made by the model are compared, and the matrix is visualized using the seaborn library.

The code developed in both approaches not only enables model training but also facilitates making predictions, thus applying the model to new images from the dataset.

An important consideration in both cases is the control of overfitting. The use of techniques such as data augmentation and a controlled number of epochs are essential to ensure that the model learns the key features of the data without overfitting to the training set. Both models are also designed to make predictions efficiently, using transformed and normalized images as input. Predictions are made using the `predict` function in TensorFlow and the `forward` method in PyTorch, and the results include the predicted class and the confidence associated with the prediction. If the confidence is below a predefined threshold of 50%, the model indicates that the class is unknown.

All the files used in the development of the code and for training the models are available for download at [GitHub].

5. Overview of Neural Network Architectures

The model summary in TensorFlow describes a convolutional neural network (CNN) consisting of several layers, with a total of 2,711,172 trainable parameters. The model begins with a convolutional layer (`Conv2D`) that applies 32 filters of size 3×3 to an input image of size 150×150, resulting in 896 parameters. Next, a pooling layer (`MaxPooling2D`) reduces the image resolution to 75×75 while maintaining the 32 channels. The next convolutional layer (`Conv2D1`) applies 64 filters of size 3×3 to the 75×75 image, resulting in 18,496 parameters. Another pooling layer reduces the resolution to 37×37 while maintaining 64 channels. The third convolutional layer (`Conv2D2`) applies 64 filters of size 3×3 to the 37×37 image, totaling 36,928 parameters, followed by another pooling layer, which reduces the final resolution to 18×18.

After the convolutional and pooling layers, the `Flatten` layer converts the output into a one-dimensional vector with 20,736 values. This vector is fed into a dense layer (`Dense`) with 128 units, resulting in 2,654,336 parameters, and finally, a last dense layer (`Dense`) performs the classification with 4 output units, corresponding to 516 parameters. The model totals 2,711,172 trainable parameters and has no non-trainable parameters.

The equivalent model in PyTorch follows the same overall structure but uses the framework's specific terminology. It also begins with a convolutional layer (`Conv2d-1`) that applies 32 filters of size 3×3 to a 150×150 image, resulting in 896 parameters. The pooling layers (`MaxPool2d-2`, `MaxPool2d-4`, `MaxPool2d-6`) progressively reduce the image resolution, while the convolutional layers (`Conv2d-3`, `Conv2d-5`) increase the depth of the representation. After the convolutional layers, the linear layer (`Linear-7`) transforms the data into a vector of 128 units, with 2,654,336 parameters, and the final linear layer (`Linear-8`) performs the classification with 4 units, totaling 516 parameters. The PyTorch-trained model also has 2,711,172 trainable parameters.

Both models have quite similar structures, with minor differences in nomenclature between the TensorFlow and PyTorch frameworks. Both models feature an efficient architecture for image classification tasks, with a large number of trainable parameters, enabling the capture of detailed information from the input images.

This model is suitable for image classification tasks with multiple classes, being efficient at learning patterns in data with high variability, such as images of different categories observed by the robot (ball, post, line, robot). Its architecture allows for the extraction of detailed features from input images, making it capable of distinguishing between these categories effectively. The large number of trainable parameters enables the model to capture complex patterns, which is particularly important when dealing with images that vary in terms of lighting, angle, and other factors. Thus, it is well-suited for applications where precise object recognition is needed in dynamic environments.

6. Confusion Matrix

A confusion matrix is an essential tool in evaluating machine learning classification models, enabling the visualization and analysis of algorithm performance. It is a table that

Table 2. Summary of Neural Network Architectures

Parameter	TensorFlow Name	PyTorch Name	Numeric Value
1st Convolutional Layer	Conv2D	Conv2d-1	896
1st Pooling Layer	MaxPooling2D	MaxPool2d-2	-
2nd Convolutional Layer	Conv2D1	Conv2d-3	18,496
2nd Pooling Layer	MaxPooling2D	MaxPool2d-4	-
3rd Convolutional Layer	Conv2D2	Conv2d-5	36,928
3rd Pooling Layer	MaxPooling2D	MaxPool2d-6	-
Flatten Layer	-	-	20,736
1st Dense Layer	Dense	Linear-7	2,654,336
Output Layer (Classification)	Dense	Linear-8	516
Total Trainable Parameters	-	-	2,711,172

compares the predictions made by a model with the actual observed values, making it easier to identify correct classifications and errors. This matrix consists of four main components: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). The arrangement of these elements allows not only the measurement of the model's accuracy but also the calculation of additional metrics such as precision, recall, and F1-score, which are crucial for a more detailed performance evaluation.

The importance of the confusion matrix lies in its ability to provide insights into the specific types of errors made by the model. For example, in contexts where the consequences of false positives and false negatives are unequal, analyzing the matrix helps adjust the model to minimize the impact of these errors. Furthermore, the matrix is widely applicable to various classification algorithms, such as Naïve Bayes, decision trees, and logistic regression, making it a versatile tool in data science.

Another relevant aspect is that the confusion matrix can be used in multiclass classification problems, where it is possible to observe the model's performance across multiple categories simultaneously. This flexibility makes the matrix a preferred choice for data scientists when evaluating models in different scenarios.

For this article, the confusion matrix was obtained for each source code, with the one shown in Fig. 3 representing the evaluation for TensorFlow.

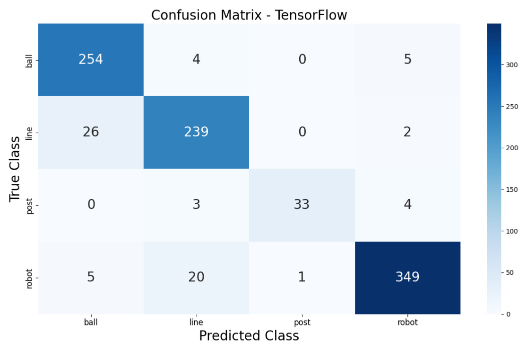

Figure 3. Confusion Matrix - TensorFlow.

The result in Fig. 3 indicates that the TensorFlow-trained model performs well in classifying the different entities. It can be observed that the “ball” class has high accuracy, with 254 examples correctly classified. The “line” class, on the other hand, shows a moderate performance, with 239 correctly identified examples. The low confusion between the “post” and “robot” classes is also noteworthy, with only 1 and 11 incorrectly classified examples, respectively. Overall, the confusion matrix suggests that the model has a good ability to separate the different entities, with particularly strong performance in classifying the “ball” class. However, there may be room for improvement in the classification of the “line” class, where there is higher confusion. A more in-depth analysis of the examples from this class could help identify opportunities to improve the model.

The result of the confusion matrix generated by the PyTorch-trained model is shown in Fig. 4.

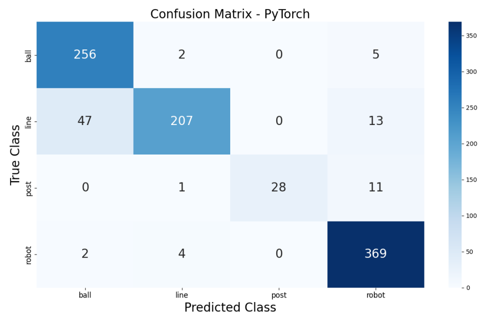

Figure 4. Confusion Matrix - PyTorch.

The PyTorch-trained model represented in this confusion matrix shows satisfactory performance in classifying the different entities. The “ball” class has the highest number of correct predictions, with 256 examples correctly classified, indicating that the model has an excellent ability to recognize this type of object. The “line” class also demonstrates good results, with 207 examples correctly classified, showing that the model can efficiently differentiate this class. The “post” class was entirely classified correctly, while the “robot” class had a few incorrect classifications.

7. Results

The results of the experiments conducted with the two Python codebases, one using TensorFlow and the other PyTorch, were based on predictions with unseen images that were not part of the training set. Each of the model’s classes — “ball”, “post”, “line”, and “robot” — was tested with two images simulating typical lighting variations of a soccer field, where the light intensity changes throughout the day, creating scenarios of both high and low luminosity. One image with excessive brightness and another with low luminosity were used to simulate these extreme conditions.

These tests aimed to evaluate the generalization ability of the models, that is, to check whether they could maintain correct predictions even in adversarial and unseen conditions. The training time for each model can be seen in Table 3. In this approach, image capture was performed on the NAO robot, and the captured images were transmitted via TCP/IP to a computer equipped with an Intel i5 5th generation processor and 16 GB of RAM, where the classification was executed.

Table 3. Training time for the implemented models.

Training	TensorFlow	PyTorch
Time (seconds)	1255 s	1733 s

Training time is an important factor when evaluating the efficiency of model implementation in different frameworks. In this case, the PyTorch-trained model took 1733 seconds, while the same training in TensorFlow was completed in 1255 seconds. This difference suggests that TensorFlow may have advantages in terms of internal optimizations and performance, depending on the configuration and hardware used. However, PyTorch is often preferred for its flexibility and ease of debugging, which may justify its use in more complex research applications and prototypes, despite the slightly longer execution time.

The prediction results for 8 images after 60 epochs of training can be seen in Table 4. Each framework was evaluated in terms of accuracy, prediction reliability, and response time.

Table 4. Prediction results for 8 images after 60 epochs of training in TensorFlow and PyTorch frameworks.

Prediction	TensorFlow	PyTorch
Ball with high brightness	1 (0.98) 0.15s	1 (0.93) 0.01s
Ball with low brightness	1 (0.88) 0.07s	1 (0.56) 0.01s
Line with high brightness	1 (0.76) 0.07s	1 (0.54) 0.01s
Line with low brightness	1 (0.59) 0.07s	0 (Incorrect prediction)
Robot with high brightness	1 (1.00) 0.07s	1 (1.00) 0.01s
Robot with low brightness	1 (1.00) 0.07s	1 (0.52) 0.01s
Post with high brightness	1 (0.98) 0.07s	1 (0.98) 0.02s
Post with low brightness	1 (0.91) 0.07s	0 (Incorrect prediction)
Accuracy	**100%**	**75%**

The TensorFlow-trained model achieved an accuracy of 100%, indicating that all images were correctly identified. The confidence values varied from 0.59 to 1.00, showing that despite variations in confidence levels, the model managed to identify all classes, regardless of lighting conditions. The prediction time for the TensorFlow-trained model ranged from 0.07s to 0.15s.

On the other hand, the model trained with PyTorch achieved an accuracy of 75%, as it failed to correctly identify two images: the line with low brightness and the post with low brightness. The confidence values for the correctly identified images ranged from 0.52 to 1.00, notably lower in some categories compared to the TensorFlow-trained model. However, the prediction time with the PyTorch-trained model was consistently faster, ranging from 0.01s to 0.02s.

The analysis of the results suggests that the TensorFlow-trained model performed more robustly in terms of accuracy and reliability, correctly identifying all images regardless of the conditions. On the other hand, the PyTorch-trained model, although more efficient in terms of prediction time, showed a significant drop in accuracy, especially for images with low lighting. This may indicate the need for adjustments in training parameters or network architecture to improve robustness under adverse conditions.

8. Conclusion

The results presented in this study indicate that the TensorFlow-trained model demonstrated superior performance during the 60 epochs of training. The analysis of the accuracy and loss curves revealed rapid progress in learning, with a remarkable improvement in accuracy during the early epochs, reaching approximately 90% after the training stabilized. This behavior suggests that the model was able to efficiently learn the patterns in the training data.

The gradual reduction in loss, with stabilization over time, reinforces the observation that the model converged to a local minimum, demonstrating its effectiveness in solving the proposed problem. The saturation behavior observed in the later epochs suggests that most of the learning occurs in the initial stages, with optimization becoming marginally slower after the 20th epoch.

The comparison of prediction results between the TensorFlow and PyTorch frameworks, presented in Table 4, highlights TensorFlow's superiority, achieving 100% accuracy on the 8 tested images after 60 epochs of training. PyTorch, on the other hand, achieved 75% accuracy, with some incorrect predictions, especially on images with low brightness, as shown in the table. This further emphasizes that TensorFlow proved more effective at recognizing complex patterns, while PyTorch showed more inconsistent results.

Therefore, this study contributes to the understanding of deep learning methods and highlights TensorFlow as a powerful tool for achieving good results in neural network models, especially those used in RoboCup. The detailed analysis of accuracy and loss curves, along with the comparison of prediction results, provides an important foundation for future optimizations in model training, allowing for more efficient use of computational resources and ensuring the development of robust and well-generalized models.

References

Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", Nature, vol. 521, no. 7553, pp. 436–444, 2015.

A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks", in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

M. Abadi et al., "TensorFlow: A system for large-scale machine learning", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library", in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.

RoboCup Federation, "Standard Platform League Rules", in *RoboCup Soccer Humanoid League Rules*, 2023.

D. Wei et al., “A survey on vision-based robotic grasping and manipulation”, *IEEE Access*, vol. 9, pp. 123214-123234, 2021.

C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning”, *Journal of Big Data*, vol. 6, no. 1, pp. 1-48, 2019.

L. Taylor and G. Nitschke, “Improving Deep Learning with Generic Data Augmentation”, in *IEEE Symposium Series on Computational Intelligence (SSCI)*, 2018, pp. 1542-1547.

Z. Zhang et al., “Deep Learning on Mobile Devices: A Review”, in *IEEE International Conference on Multimedia and Expo (ICME)*, 2019, pp. 1516-1521.

S. Liu and W. Deng, “Very deep convolutional neural network based image classification using small training sample size”, in *3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, 2015, pp. 730-734.

Y. Jia et al., “Caffe: Convolutional architecture for fast feature embedding”, in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, pp. 675–678.

K. He et al., “Deep Residual Learning for Image Recognition”, in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770-778.

S. Ren et al., “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137-1149, 2017.

L. Seidenari et al., “Real-time object detection for robotic applications”, in *IEEE International Conference on Robotics and Automation*, 2012, pp. 4750–4755.

D. Gouaillier et al., “The NAO humanoid: a combination of performance and affordability”, IEEE Robotics & Automation Magazine, vol. 18, no. 3, pp. 12-25, 2011.

Aldebaran Robotics, “NAO Technical Documentation”, Softbank Robotics, Technical Report, 2021.

M. Schwarz et al., “NimbRo-OP2X: Adult-sized Open-source 3D Printed Humanoid Robot”, IEEE-RAS International Conference on Humanoid Robots, 2019.

R. Hartley and A. Zisserman, “Multiple View Geometry in Computer Vision”, Cambridge University Press, 2nd Edition, 2004.

H. Ishihara et al., “Real-time visual processing for humanoid robots using a GPU-embedded computer”, in IEEE-RAS International Conference on Humanoid Robots, 2008, pp. 305-310.

S. Ivaldi et al., “Anticipatory models of human movements and dynamics: the roadmap of the AnDy project”, in IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids), 2015, pp. 688-695.

Y. Cui et al., “The design and implementation of a low-cost, high-performance control system for the NAO humanoid robot”, in 2013 IEEE International Conference on Robotics and Biomimetics (ROBIO), 2013, pp. 698-703.

S. Degallier et al., “Towards a bio-inspired control of a humanoid robot”, in 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2008, pp. 5446-5451.

S. Bhattacharya and N. Vidyarthi, “A survey on humanoid robot control architectures”, in 2014 International Conference on Computing for Sustainable Global Development (INDIACom), 2014, pp. 837-842.

G. Seet, H. Fang, and C. Xiao, “A review on visual perception for robotic soccer”, in 2012 International Conference on Control, Automation and Information Sciences (ICCAIS), 2012, pp. 18-23.

S. Fuke and M. Tamada, “Visual perception and recognition for a humanoid robot”, in 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2010, pp. 3553-3558.

Aldebaran, “NAO Cameras,” available at: `http://doc.aldebaran.com/2-8/family/nao_technical/video_naov6.html`. [Accessed: Dec. 31, 2024].

S. Bianco et al., “Benchmark Analysis of Representative Deep Neural Network Architectures”, *IEEE Access*, vol. 6, pp. 64270-64277, 2018.

M. J. Shafiee et al., “Evolution in Groups: A deeper look at synaptic cluster driven evolution of deep neural networks”, *Future Generation Computer Systems*, vol. 98, pp. 430-440, 2019.

H. Kitano et al., “RoboCup: A challenge problem for AI and robotics”, in *RoboCup 1997: Robot Soccer World Cup I*, Springer, 1998, pp. 1-19.

M. Asada et al., “RoboCup: Today and tomorrow – What we have learned from RoboCup competitions”, *Artificial Life and Robotics*, vol. 24, no. 1, pp. 51-58, 2019.

T. Röfer et al., “B-Human Team Report and Code Release 2019”, *B-Human*, Technical Report, 2019.

RoboCup Technical Committee, “RoboCup Soccer Standard Platform League (NAO) Technical Report”, *RoboCup Federation*, Technical Report, 2023.

H. He and E. A. Garcia, “Learning from Imbalanced Data”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263-1284, 2009.

S. Liu et al., “Deep Learning in Object Detection: A Survey”, in *International Conference on Information Technology in Medicine and Education (ITME)*, 2019, pp. 427-431.

V. Souza and H. S. Azevedo, “Benchmark codes and data,” available at: `https://github.com/vitor-souza-ime/benchmark`. [Accessed: Dec. 31, 2024].

ResearchGate, NAO Vision, ”NAO robot head camera field of view,” available at: `https://www.researchgate.net/figure/NAO-robot-head-camera-field-of-view_fig2_329517532`. [Accessed: Dec. 31, 2024].

F. Florencio, T. Valenç, E. D. Moreno, and M. C. Junior, “Performance Analysis of Deep Learning Libraries: TensorFlow and PyTorch,” in *Journal of Computer Science*, vol. 15, no. 6, pp. 785-799, 2019. `https://doi.org/10.3844/jcssp.2019.785.799`

H. Dai, X. Peng, X. Shi, et al., “Reveal training performance mystery between TensorFlow and PyTorch in the single GPU environment,” in *Sci. China Inf. Sci.*, vol. 65, p. 112103, 2022. `https://doi.org/10.1007/s11432-020-3182-1`