# Electrical Scheme Digitization Using Deep Learning Methods

Marko Putak, Vatroslav Zuppa Bakša and Andrea Bednjanec
Zagreb University of Applied Sciences/Automation and Process Control Engineering, Zagreb, Croatia
mputak98@gmail.com, vzuppabak@tvz.hr, andrea.bednjanec@tvz.hr

***Abstract* - The use of software tools and applications progressively became a standard in both education and industry. A solution for hand-drawn electrical scheme digitization has been proposed to match the fast-paced dynamic of the modern world in the field of electrical engineering. The aim is to notably reduce time-consuming and error-prone electrical scheme tracing from hand-drawn to simulating software. The means have been achieved through the usage of state-of-the-art deep learning model YOLOv5 for electrical elements detection along with Python and OpenCV library for data processing. The user's input is an image of a hand-drawn circuit, and the end result is an LTspice digitized electrical scheme ready for simulation.**

***Keywords - electrical scheme; digitization; deep learning; object detection; YOLOv5; LTspice***

## I. INTRODUCTION

Electrical circuits are a crucial aspect of electrical engineering. The conventional method of creating an electrical circuit involves manually drawing it on paper, followed by redrawing it in a simulation program during the later stages of analysis. This redrawing process is time-consuming, reduces efficiency, and can result in human error.

The intention to automate the process seems to be of great use for the purpose of removing the need for redrawing and setting the scheme in a state ready for simulation. A branch of computer science that offers a solution of the mentioned problem is artificial intelligence.

The proposed solution consists of using an electrical scheme image dataset to train a deep learning model for electrical component detection. Henceforth, detected components are processed to produce a fully functional digitized electrical scheme.

## II. DATASET

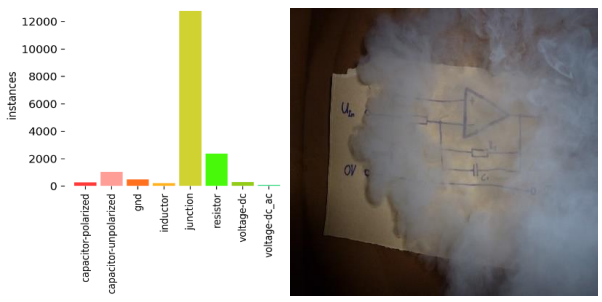
Datasets are an integral part of contemporary object recognition research [1]. Generally, the weak and unattended dataset is a bottleneck to the model's performance, leading to poor results altogether. It is of uttermost importance to squeeze in those extra hours to find a good match, as it will be significant for the rest of the project.

For this project, the CGHD1152 (Circuit Graph Hand Drawn 1152) dataset was chosen as the best available option [2]. CGHD1152 consists of 1152 *.jpg* images derived from 144 unique circuits accompanied by 48563 object annotations. Despite the dataset is divided into 45 unique classes (junction, resistor, speaker, switch, etc.), 7 classes are being used for the sake of efficiency and accuracy.

Further deconstruction and visualization of the dataset provide a clear picture of its unbalanced class distribution seen in Fig. 1 (a), and unrepresentative instances in Fig. 1 (b).

The CGHD1152 dataset was purged of outliers and unannotated images using Roboflow, a computer vision platform designed for dataset manipulation, export, and deployment [3]. Preprocessing steps, such as auto-orientation, resizing, and conversion to grayscale, were applied through Roboflow. Image augmentation was not included in the preprocessing since YOLOv5 handles it internally.

The images were divided into three standardized subsets: training, testing, and validation, comprising 70% (813 images), 15% (175 images), and 15% (174 images) of the data, respectively. In addition, roughly 1% of background images (null images) were added to reduce False Positive (FP) classification, following Ultralytics' recommendations [4].




(a) (b)

Figure 1. Dataset visualization.
(a) Class distribution, (b) unrepresentative instance

## III. YOLOv5

YOLOv5 is the state-of-the-art (SOTA) model specialized for object detection. The abbreviation YOLO is derived from "You Only Look Once", meaning that the model uses single-shot detection (SSD) to predict the presence of an object and its corresponding location [4].

What makes YOLO a superior model and the perfect candidate is the usage of a grid-like division during inference that allows the model to see the input image as a whole while maintaining record-level speed and high mean average precision (mAP) [5].

### *A. Model Architecture*

- Backbone: CSP-DarkNet53
- Neck: CSP-Path aggregation network (PAN)
- Head: YOLOv3 Head

This particular Backbone structure ensures high inference speed and accuracy while maintaining relatively low memory cost by solving the issue of repetitive gradient information passage [6].

In the Neck, the Path Aggregation Network is used to boost the information flow of the network by propagating low-level features in a feature pyramid [7].

The model Head is mainly used to perform the final detection part. It applies anchor boxes on features and generates final output vectors with class probabilities, objectness scores, and bounding boxes [8].

Considering the scarcity of data provided in the dataset[1], as well as possible applications of the model in real-time detection scenarios, the YOLOv5m version is selected based on its ideal balance of inference speed and prediction accuracy. YOLOv5m consists of 291 layers, 20.9 million parameters, and uses 48.3 GFLOPs.

### *B. Initial training*

The initial model training was accomplished with a Jupyter notebook script running on Google Colab [9][10]. The script begins with installing the dependencies and importing the dataset. Afterward, the training procedure is initialized using pre-trained weights to accelerate the training process. The simplicity provided by Ultralytics makes the training process intuitive and easy to debug where necessary. The initial weights, yolov5m.pt, were trained on the COCO dataset [11]. To understand the general behavior of the model, most hyperparameters were set to their default values, with notable exceptions being the number of epochs (150), batch size (32), image size (640), and optimizer (Stochastic Gradient Descent).

The first training took 1 hour and 3 minutes using the Tesla T4 graphics card (GPU) and yielded satisfactory results.

The best results were at epoch 136 with mean average precision at 0.5 threshold of intersection over union (mAP@0.5) equating to 0.9888 and mAP [0.5:0.95] of 0.7353, where [0.5:0.95] represents different IoU thresholds, from 0.5 to 0.95 with a step of 0.05.

### *C. Hyperparameter Evolution*

Having a well-defined base case of the model and its metrics, the usual procedure would involve tweaking and experimenting with different hyperparameter values used to train the model repeatedly in the hope of improvement. This is also known as hyperparameter tuning [12].

In order to avoid such time-consuming, exhaustive and iterative process, hyperparameter evolution offers an interesting solution [13].

The evolutionary algorithm uses a single gene to encode each hyperparameter that needs to be optimized for each individual. A range and resolution are specified for each gene to prevent searching irrelevant regions of the hyperparameter space. The initial population is generated by randomly choosing each gene from a uniform distribution, after which the fitness of each individual is assessed. The individuals with the highest fitness from the previous generation are used to form subsequent generations through selection, crossover, and mutation [14].

Ultralytics YOLOv5 repository has a built-in hyperparameter evolution function that uses modified mutation to find the best set of hyperparameter values. For the evolve parameters a number of epochs is 10 per default settings, and a number of generations is set to 150. Evolution lasted for 4 hours in total using RTX 3070Ti GPU, and the outcome was a fine-tuned set of hyperparameters where some of which are depicted in Fig. 2.

Marginal improvements were made by retraining the model on the new set of hyperparameters, as is depicted in the F1 curve in Fig. 3. The junction's F1 score is low compared to other classes due to its small and overrepresented nature, hence this significant disparity in the results.

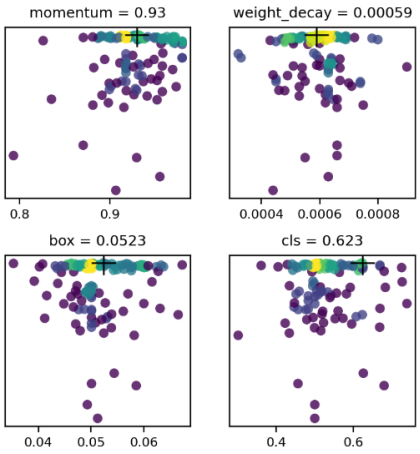


Figure 2. Hyperparameter evolution scatter plot

[1] A decent dataset contains more than 1500 images per class!

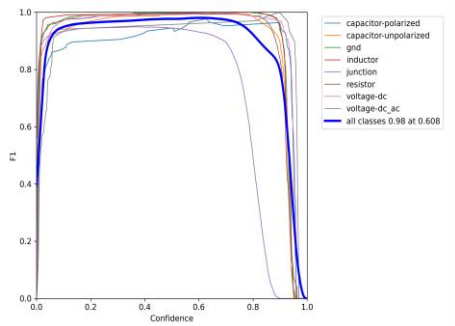



Figure 3. F1 curve of the model with tuned hyperparameters

Newly found metrics are:

- mAP@0.5 = 0.9895
- mAP@[0.5:0.95] = 0.7496
- Precision = 0.9794
- Recall = 0.9820

### *D. Final Model*

Further results and dataset analysis discovered that increasing the image size from 640 pixels to 1280 pixels has a positive impact on the training results since the majority of the bounding boxes, especially junctions, are quite small and easily misplaced. Consequentially, changing the image size to 1280 pixels and retraining the model with evolved hyperparameters gave the best results.

Training took 2 hours and 56 minutes to complete at epoch 167, and was logged using wandb, an experimental tracking tool for machine learning. This provided several valuable metrics, such as the normalized confusion matrix shown in Fig. 4 [15].

Examining the confusion matrix, it can be seen that the model's classification is highly accurate, except for background images where 76% of the time the model confused them with junctions.

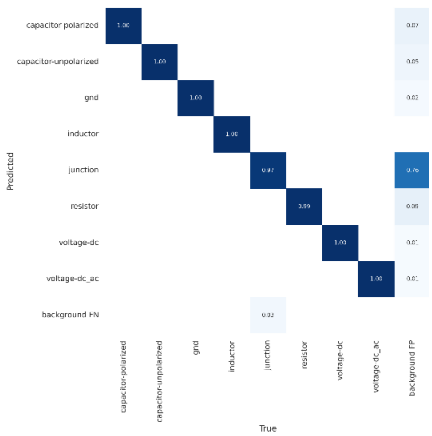



Figure 4. Confusion matrix of the final model

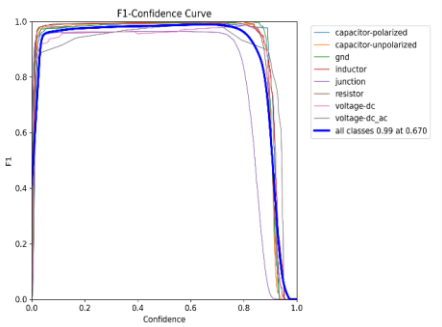



Figure 5. F1 curve of the final model

A comparison of the final model's F1 score in Fig. 5 and the evolved model's F1 score in Fig. 3 revealed that the final model had a better curvature, especially for the junction class.. This is reflected in the mAP metric, which combines precision and recall in its calculation, resulting in:

- mAP@0.5 = 0.9907
- mAP@[0.5:0.95] = 0.7614
- Precision = 0.9883
- Recall = 0.9909

making this model a perfect candidate for later use.

Overfitting occurs when a model does not generalize well from observed data to unobserved data [16]. YOLOv5 addresses this issue by using regularization techniques such as weight decay and early stopping [17]. Early stopping is a regularization method that aims to stop the training process before the model begins to overfit, specifically at the inflection point of the validation error function [18]. This is why the training ended at 167th epoch - the model stopped at the optimal time and saved the best weights to a file named *best.pt*.

The increase in image size from 640 pixels to 1280 pixels did result in improved overall accuracy for the model, however it also led to a decrease in both training and inference speed. This trade-off is commonly encountered in the field, and the choice is determined based on the specific requirements of the model application. In the case of electrical scheme digitization, the inference speed of 39.8 milliseconds is deemed appropriate for the task and therefore the model is ready to be implemented.

## IV. DIGITIZATION

With the model thoroughly evaluated and ready, the next step is to begin data processing using Python 3.10 programming language and PyTorch as the primary framework [19][20].

### *A. LTspice*

LTspice XVII, a high-performing computer program for electrical circuit simulation, has been chosen as the desired output. It is based on the SPICE simulator (Simulation Program with Integrated Circuit Emphasis), open-source computer software used to analyze and predict

the behavior of electronic circuits [21]. LTspice is widely used and accepted as the most prevalent SPICE software in the industry, making it a valuable choice due to its high user base.

Although ease of use, clarity, low computational complexity and accuracy are significant benefits of the LTspice program, the main benefit of using LTspice is its ability to render a textual type of data and create a circuit. This is achieved through a simple syntax that represents a specific circuit, making digitization possible. Fig. 6 (a) illustrates an electrical circuit in LTspice and Fig. 6 (b) its matching syntax created in the text file.

To get a better understanding of how the syntax should be written, several electrical circuits were manually drawn and inspected from their correlating text format. The conclusion was deduced for the format in which the text file should be written generally:

- <Name of the element> <coordinates>

For instance, a random wire element is written as:

- WIRE $x_s$ $y_s$ $x_e$ $y_e$

Where $x_s$, $y_s$ represents the starting point, and $x_e$ $y_e$ represents the ending point of the wire.

Similarly, electrical components are written:

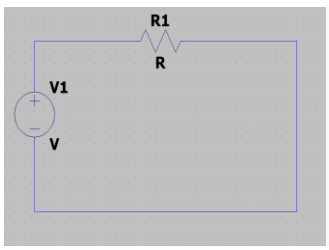
- SYMBOL res $x_c$ $y_c$ Rz

Where $x_c$, $y_c$ represents the center point and z $\in$ {0, 360} represents the rotation of the resistor.

### B. Code Implementation

The model is loaded through PyTorch's built-in function *torch.hub.load()* with custom weights as an argument. OpenCV's *.imread()* method is used to load a new image that the user wants to digitize. With the new image, the model makes predictions and visualizes the results using *.show()* method depicted in Fig. 7. Each element is surrounded by a bounding box with a confidence score.

The interpretation of the model's results is made convenient through the use of pandas, an open-source Python library for data analysis and manipulation [22]. Calling the *.pandas().xywh[0]* function on the result, a table shown in Fig. 8 is generated containing center coordinates, width, height, and other useful information regarding the predicted bounding boxes.




```
Version 4
SHEET 1 880 680
WIRE 272 80 112 80
WIRE 528 80 352 80
WIRE 112 144 112 80
WIRE 112 320 112 224
WIRE 528 320 528 80
WIRE 528 320 112 320
SYMBOL res 368 64 R90
SYMATTR InstName R1
SYMBOL voltage 112 128 R0
SYMATTR InstName V1
```

(a) (b)

Figure 6. LTspice electrical circuit.
(a) Generated circuit, (b) circuit's text format

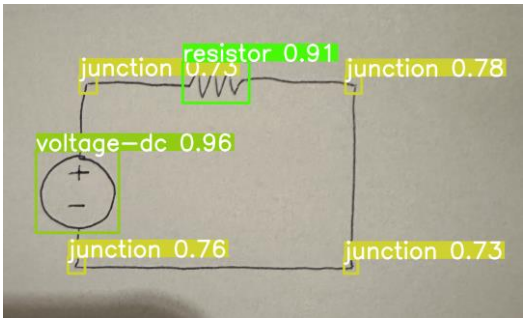



Figure 7. Visualization of model inference

As the WIRE elements in LTspice dictate the connections of the circuit and the placement of the elements, the first challenge faced was the finding of the start and end points of the wires. Upon examination of the junction class, it was deduced that the junction class was composed of two or more lines that were horizontally and vertically connected, making it a perfect match for the endpoints of the wires. Hence, by iterating over the junction class in the dataframe, the wire endpoints were derived and stored in a list.

However, simply connecting every combination of junctions is not correct, as there may not be a wire between every junction. A solution was proposed to only connect junctions that were horizontally or vertically aligned. This was achieved by finding the angle of slope formed by two given junctions.

The slope of any straight line on the *X*-*Y* plane is given by *tan* $\theta$, where $\theta$ is the angle that the straight line makes with the positive direction of *x*-axis. Hence the equation for the slope is

$$m = \tan\theta = \frac{y2-y1}{x2-x1}. \quad (1)$$

Where m is the slope of that straight line which is inclined at an angle θ with the positive direction of x-axis, and (x1, y1), (x2, y2) are coordinates of the first and second junction, respectively.

Therefore, the angle of slope is

$$\theta = \tan^{-1}(m) = \tan^{-1}\left(\frac{y2-y1}{x2-x1}\right). \quad (2)$$

The wire is classified as horizontal if the angle of slope falls within 180 ± 10% or 360 ± 10%. Conversely, the wire is classified as vertical if the angle of slope falls within 90 ± 10% or 270 ± 10%. The elimination of non-orthogonal wires leaves the initial list comprised of only valid wires.

| | xcenter | ycenter | width | height | confidence | class | name |
|---|---|---|---|---|---|---|---|
| 0 | 750.46 | 1928.72 | 403.03 | 398.10 | 0.96 | 6 | voltage-dc |
| 1 | 1422.68 | 1395.59 | 323.33 | 202.92 | 0.91 | 5 | resistor |
| 2 | 2091.57 | 1409.49 | 77.22 | 83.85 | 0.78 | 4 | junction |
| 3 | 745.30 | 2285.22 | 82.79 | 78.64 | 0.76 | 4 | junction |
| 4 | 801.75 | 1407.23 | 83.94 | 86.41 | 0.73 | 4 | junction |
| 5 | 2079.28 | 2289.68 | 66.88 | 71.54 | 0.73 | 4 | junction |

Figure 8. The model inference dataframe

For loop was included to iterate over the list and write the coordinates to a file, which was later opened in LTspice for analysis. It was discovered that the wire validation method was effective, but the wires were not perfectly horizontal or vertical as a result of the user's hand-drawing or image capturing not being properly aligned. As a result, the user was unable to interact with the wires or elements, as LTspice only allows for manipulation of vertical and horizontal straight wires.

To overcome this challenge, a wire alignment function was implemented. To align the vertical wires on the *x*-axis, a function looped through all the points' *x*-coordinates, comparing their relative positions, and making them equal if they satisfied a proximity condition given by a threshold value. A similar procedure was implemented for the alignment of the horizontal wires using the *y*-coordinates of the points.

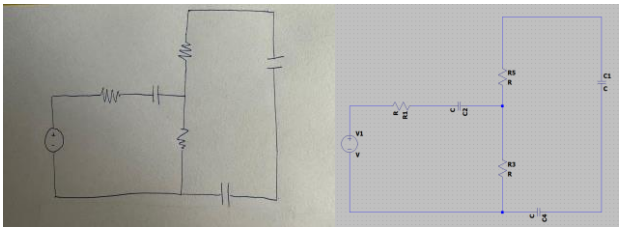
However, the aligned wires were out of reach and could not be manipulated by the user as they can only interact with wires on the edges of 16 by 16-sized grids. To address this bug and have the user be able to add, remove or modify the wires and elements, a simple wire offset method was implemented as shown in the following pseudocode:

```
for wire in all_wires_list:
    if wire[x_coordinate] % 16 == 0:
        continue
    else:
        offset_x = wire[x_coordinate] % 16
        wire[x_coordinate] -= offset_x
```

The *y*-coordinate offset correction was also applied with the same logic.

Similar to wires, electrical elements were also added to the output file using the *.write()* built-in Python function. If the element is placed on a wire, LTspice will automatically connect the element in a circuit and remove the excess wire, therefore the main challenge was determining the appropriate rotation for these elements based on whether they needed to lie on a horizontal or vertical wire. This problem was resolved by splitting the list of all valid wires into two smaller lists - *horizontal_wires* and *vertical_wires*. By comparing whether the center of the element was located between two junctions of a horizontal or a vertical wire, the appropriate rotation of the elements was assigned and included in the output file.

Upon program execution, a *.asc* file is generated in the current directory containing all the wires and elements depicted in a hand-drawn scheme and can be opened using LTspice. Fig. 9 shows an example of the input image (a) and its corresponding end product (b).



(a) (b)

Figure 9. A before-and-after example.
(a) Input image, (b) end result

### C. *Limitations*

Although providing an efficient and convenient way of digitizing electrical schemes, there are some limitations to the process.

One of the main limitations is the possibility of using only six different types of elements that occur most frequently. Excluding other elements could be problematic for users who require a wide range of elements. On the other hand, the computational expenses of providing more elements are marginal, therefore, including more elements in the future is realizable.

Another constraint is the inability to use non-orthogonal wires, such as diagonal wires, due to the method used for finding valid wires. Additionally, the current dataset is insufficient and lacks the necessary data to train the model accurately.

### D. *Results*

A solution has been developed to meet the challenge of creating a digitized and fully operational electrical circuit from hand-drawn designs. By using the highly accurate deep learning model YOLOv5, which was trained on a custom dataset, and processing the output from the model, a fully functional LTspice electrical circuit was created. The digitization process involves the user inputting an image, the model making predictions, and processing the output to produce a file named "output.asc" that can be opened in LTspice for simulation or measurement purposes.

The end result was further validated by using a subset of data from the dataset along with newly-drawn circuits. Although the validation procedure produced positive results, it also identified areas requiring improvement, specifically in the heuristics of digitization, which tend to produce errors when the drawings are complex or cluttered.

The digitization process is nearly independent of the drawing's complexity, making it a more efficient alternative to manual redrawing. In most cases, the entire process can be completed in less than a minute.

## V. Conclusion

The proposed program for hand-drawn electrical scheme digitization is a promising solution that can greatly improve efficiency and reduce errors in the field of electrical engineering. By leveraging state-of-the-art deep learning techniques for object detection, the program successfully automates the time-consuming and error-prone process of redrawing hand-drawn circuits for simulation. While further improvements and validations are required, the program has demonstrated its potential to offer an accessible and reliable solution for a wide range of users.


## Acknowledgment

I extend my heartfelt gratitude to my co-authors for their unwavering support and assistance throughout the entire project. I'd like to recognize the assistance of my fellow colleagues, and lastly, it would be remiss not to mention the support I received from Miss Bruna Duspara.

## REFERENCES


[1] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," CVPR 2011, Colorado Springs, CO, USA, 2011, pp. 1521-1528, doi: 10.1109/CVPR.2011.5995347.

[2] Thoma, Felix, Johannes Bayer, and Yakun Li. "CircuitGraphHandDrawn." OSF, 8 Aug. 2022. Web.

[3] Dwyer, B., Nelson, J. (2022), Solawetz, J., et. al. Roboflow (Version 1.0) [Software]. Available from https://roboflow.com. computer vision.

[4] Jocher, G. (2020). YOLOv5 by Ultralytics (Version 7.0) [Computer software]. https://doi.org/10.5281/zenodo.3908559

[5] Thuan, Do. "Evolution of Yolo algorithm and Yolov5: The State-of-the-Art object detention algorithm." (2021).

[6] Xu, Renjie & Lin, Haifeng & Lu, Kangjie & Cao, Lin & Liu, Yunfei. (2021). A Forest Fire Detection System Based on Ensemble Learning. Forests. 12. 217. 10.3390/f12020217.

[7] Liu, Shu, et al. "Path aggregation network for instance segmentation." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[8] Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." arXiv preprint arXiv:1804.02767 (2018).

[9] Kluyver, T., Ragan-Kelley, B., Fernando P'erez, Granger, B., Bussonnier, M., Frederic, J., … Willing, C. (2016). Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas (pp. 87–90).

[10] Bisong, E. (2019). Google Colaboratory. In: Building Machine Learning and Deep Learning Models on Google Cloud Platform. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4470-8_7

[11] Tsung-Yi Lin, Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., … Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context. CoRR, abs/1405.0312. Retrieved from http://arxiv.org/abs/1405.0312

[12] Bardenet, Rémi, et al. "Collaborative hyperparameter tuning." International conference on machine learning. PMLR, 2013.

[13] Schmidt, Mischa, et al. "On the performance of differential evolution for hyperparameter tuning." 2019 international joint conference on neural networks (IJCNN). IEEE, 2019.

[14] Young, Steven R., et al. "Optimizing deep learning hyper-parameters through an evolutionary algorithm." Proceedings of the workshop on machine learning in high-performance computing environments. 2015.

[15] L. Biewald, "Experiment Tracking with Weights and Biases," Weights & Biases. [Online]. Available: http://wandb.com/. [Accessed: 02.02.2023.]. Software available from wandb.com

[16] Ying, Xue. "An overview of overfitting and its solutions." Journal of physics: Conference series. Vol. 1168. IOP Publishing, 2019.

[17] Loshchilov, Ilya, and Frank Hutter. "Decoupled weight decay regularization." arXiv preprint arXiv:1711.05101 (2017).

[18] Yao, Yuan, Lorenzo Rosasco, and Andrea Caponnetto. "On early stopping in gradient descent learning." Constructive Approximation 26.2 (2007): 289-315.

[19] Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

[20] Paszke, Adam, et al. "Automatic differentiation in pytorch." (2017).

[21] Quarles, Thomas L., Analysis of Performance and Convergence Issues for Circuit Simulation, Memorandum No. UCB/ERL M89/42, University of California, Berkeley, April 1989.