

Advanced machine learning

Mattias Ermakov Thing

April 7, 2024

1 Introduction, GitHub, and neural networks

The first lecture is focused on establishing prerequisites and leveling out the playing field such that everyone is on the same page as we dive into the details in the following lectures. We take a look at the structure of the course and GitHub where the course material is located. Then we will review the basics of neural network which will be the focus of this course.

1.1 Course overview

This course aim to introduce the students to various of methods of machine learning, primarily focused on neural networks and ways of training them. By the end of the course students should know about various types of neural network architectures and when to use a given architecture. We also look at ways to improve neural networks with physics guidance. The course consists of five two hour lectures with the following content:

- Introduction, GitHub, and neural networks
- Machine learning and methods of training
- Neural network architectures
- Physics guidance in neural networks
- TBD

Each lecture will come with relevant code and exercises to prepare for the next lesson. Although the exercises are not mandatory, it is highly recommended to do as it will improve practical skills. After the first lesson we will start the lectures by discussing the exercise from the previous lecture. The course material is available at: <https://github.com/CP3-Origins/advanced-machine-learning>. Beyond the lecture notes Google is your friend.

1.2 Prerequisites

This course will use Python and basic programming skills in this language is expected. To develop machine learning algorithms we will use **Keras** and **TensorFlow 2** [2, 1], which means students must have Python 3.8 installed or a newer version. The university provides the paid version of PyCharm as an IDE for Python, but you can use whatever for writing code.

The required packages for the course can be found in the `requirements.txt` file in the `code` folder. From that directory you can install the packages with pip using: `pip install -r requirements.txt`

The course material is located on GitHub (see the link in the course overview), and students can find the lecture notes in the `lecture_notes` folder. The relevant code be found in the subfolders of the `code` folder.

Students are expected to know calculus, linear algebra, and the basics of neural networks. This course builds on concepts from FY555 (Introduction to Python, machine learning and data handling for the physical sciences), but it is not necessary to have taken that course to follow.

1.3 Introduction to Git

All course material for this course is on GitHub and it is recommended you know how to access this material. GitHub is a developer platform using Git to manage projects. A link to the homepage of Git can be found here: <https://git-scm.com/>.

Git is a free and open source distributed version control system. This means that it helps us manage versions of project and collaborate in an asynchronous fashion. Each contributor has a local version of the repository, and can develop locally on that version, and then share changes in a controlled and versioned manner. Git is a key tool for code development and if you ever plan to work in IT, it is something you need to know. It can also be very useful for doing scientific work in groups if code is involved. Also, it allows you to save code and other files in the cloud as a backup. Two of the most popular platforms for using Git is GitHub and GitLab. In this course we use GitHub. Students are encourages to study Git and version control.

There are five main commands you will need to use Git: `git clone`, `git pull`, `git add`, `git push`, and `git push`.

To get started you need to download and install Git. Having installed Git you can clone the course repository. This can be done several ways:

- Using a Git plugin in your IDE
- With a `git clone` command in your terminal
- By downloading the repository as a zip file (not recommended)

If you just download the repository as a zip file, you are not leveraging Git. This method works if you simply want to download the course content, but will not teach you how to use Git.

In the following we will consider how to work with Git using the terminal. Assume you want to download the course content from GitHub. Then you would go to the directory where you want to download the repository and execute the following command:

```
git clone git@github.com:CP3-Origins/advanced-machine-learning.git
```

This will download the repository on your device. Since this course will be updated and improved throughout the course, there might be changes to the course. To update to the latest version you can use the following command anywhere inside the Git project by executing the `git pull` command.

This will however not work, if you have locally modified a part of the repository, which you are trying to update. This is to avoid overwriting changes when merging the new file with your local files. In this case you need to resolve any merge issues, such that the files are merged in the desired way without having losing any work due to file overwriting.

If you plan to participate in exercises and develop your own networks, then it is recommended that you start by creating a fork of the repository. This will create a copy of the repository which is on your own GitHub account. The only downside is that you need to remember update your own fork to get the updates from the main repository on CP3. Other than that you can clone your forked repository and use pull to update your repository.

Suppose you have made your own script with some neural network as part of an exercise, you can now upload that to your fork and save your changes to the cloud. In this case, if you lose your computer, you will still have everything from this course including any code you have created.

To do this you need to push your code from your local machine to your fork on GitHub. To get an overview of file changes in the Git project, you can use the command `git status`. Then you can add the files you want to save using the command `git add <path/to/file>`. You can specify a folder or file which you want to add. You can execute `git add .` if you want all changes to be added.

Next, you need to commit the code to Git. This is for version control. Together with this it is recommended that you add a short note on what your change does. An example would be `git commit -m "Adds a network to model rainfall"`, where `-m` is the message flag and the part in double quotation is the commit message.

Finally, you can use the push command to push the changes using the command `git push`. This will push your local changes to your fork and you should be able to see the changes if you open your repository on GitHub.

The reason for using a fork is that it creates a personal copy for each

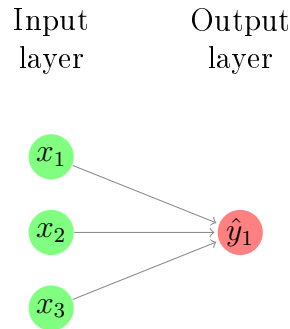


Figure 1: A simple neural network with an input layer, and a single neuron in the output layer.

student. We cannot have each student to save their work on the main course repository. This would also cause collisions if students used the same file names. You own your fork and you can invite colleagues to your participate if you want to work on the exercise together.

1.4 Neural networks

Before going into a more thorough overview of machine learning (ML) methods, let us recall the basics of neural networks and look at implementing neural networks. In the next lecture we will get back to a general discussion of neural networks.

1.4.1 Theory

As the name suggests, neural networks are composed of neurons similar to our brain. Any neuron network uses the same basic building block, the single neuron. In some applications such as in robots less than 100 may be sufficient [3], whereas image classification networks may contain thousands of neurons [5]. Thus, to understand neural networks we should take our time to understand how they work. Neurons are fundamentally doing linear regression, and later we discuss how nonlinear behavior is obtained as this is key to the power of neural networks. As a simple example, one can consider a single neuron "network" shown in Figure 1. This "network" takes in three inputs, x_i , where $i = 1, 2, 3$, and computes a predicted value \hat{y} . The word network is in quotation as one would need multiple interconnected neurons to properly talk of a network.

In the case of this single neuron with three inputs shown in Figure 1 the equation becomes,

$$z(x_i) = w_1x_1 + w_2x_2 + w_3x_3 + b, \quad (1.1)$$

where each input is associated with the weight w_i which modulates the influence of the input, and then there is a bias term, b . This can be written in

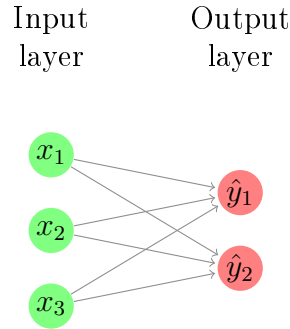


Figure 2: A simple neural network with an input layer, and two neurons in the output layer.

a compact matrix form as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + [b] = w_{ij}x_i + b_j \cdot \vec{1}, \quad (1.2)$$

where an additional j index is added to indicate the number of neurons, but since there is only one neuron it is redundant. The use of this index becomes apparent when considering a network like in Figure 2, but with two fully connected output nodes, $j = 2$. In this case, the equation would be,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \quad (1.3)$$

This highlights the benefit of the notation with the weight matrix as w_{ij} , the input vector as x_i , and the bias vector as b_j . A technical note would be that, in practice, it is more efficient to skip the process of adding the bias terms explicitly. One can exploit the definition of matrix multiplication and write it as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} & b_1 \\ w_{12} & w_{22} & w_{32} & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \quad (1.4)$$

From this we can then also consider more complex networks with hidden layers as in Figure 3. In this case we would first compute the first the values at the hidden layers $z_1(x_i) = a_i$ and then we can use that result to compute that result $z_2(a_i) = \hat{y}$.

A linear equation is not sufficient to produce complex nonlinear decision boundaries, therefore it is key to modify the result $z_j(x)$ such nonlinearity is achieved. This is very simply to prove, consider our simple network with one

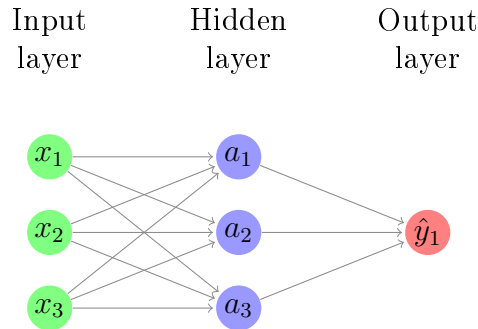


Figure 3: A simple neural network with an input layer, a hidden layer, and a single neuron in the output layer.

hidden layers. The math becomes,

$$\hat{y} = z_2(z_1(x_i)) = w_{i2} \left(w_{i1}x_i + b_1 \cdot \vec{1} \right) + b_2 \cdot \vec{1}. \quad (1.5)$$

The problem here is that while we have different weights and biases, due to the lack of nonlinearity, this hidden actually achieve nothing because we can redefine this equation to be just like a network without a hidden layer,

$$\hat{y} = z_2(z_1(x_i)) = \underbrace{w_{i2}w_{i1}x_i}_{w_{i1}x_i} + \underbrace{w_{i2}b_1 \cdot \vec{1} + b_2 \cdot \vec{1}}_{b_1 \cdot \vec{1}}. \quad (1.6)$$

This is perfectly valid because the latter terms is nothing but a sum of trainable variable, which is nothing but a number, and then first part contains the input times two different weight, but again we can just redefine that as a single weight. Therefore, without any nonlinearity we can add an arbitrary number of hidden layers and achieve the equivalent of having no hidden layers.

This is fixed using nonlinear activation functions, $a(z)$, such as the sigmoid function, $\sigma(x)$, and the rectifier linear (ReLU) function, $\text{ReLU}(x)$. There are many more options, and it is up to the designer of the network to determine the best activation function. The activation function, f , can be applied on a per-neuron basis, but it is commonly applied across the layers,

$$a(z_j(x_i)) = f(w_{ij}x_i + b_j \cdot \vec{1}). \quad (1.7)$$

Each layer of the neural network is generally computed in this way. One can then stack the layers, $a(z_j)^n$, where n is the number of layers. Deep neural networks then have three layers or more, $n \geq 3$. The output of the neural network is then computed by iteratively computing the layers from input to output, with the previous layer being the input to the next layer.

1.4.2 Tensorflow and keras

For this course we will be using **Keras** and **TensorFlow 2** [2, 1] to develop neural network. There are also other packages such as **PyTorch** [4] which you

are free to use, but it will not be used in the example code of this course. At the end of the day it is all linear algebra, but there are some differences in the packages. In general Keras provides a more beginner-friendly interface, but we will introduce advanced features which can be done in both TensorFlow and PyTorch. Some of the exercises are will be suitable to do in PyTorch and you are welcome to do so.

This weeks exercise is about introducing the basics of coding with Keras and TensorFlow 2. This example has also been used in FY555, but in this course we will consider a different exercise.

For this first part we will use Keras and some sklearn functions to started. First we import dependencies:

```
1 import matplotlib.pyplot as plt
2 from sklearn.datasets import load_iris
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import StandardScaler,
   LabelEncoder
5 from keras.models import Sequential
6 from keras.layers import Dense
7 from keras.utils import to_categorical
8
```

Then we download the iris dataset from sklearn as our simple training set, and convert the text labels to numbers using encoding.

```
1 # Data (as pandas dataframes)
2 X = load_iris().data
3 y = load_iris().target
4
5 # Convert string labels to numeric values using Label
   Encoding
6 label_encoder = LabelEncoder()
7 y = label_encoder.fit_transform(y)
8
9 # Convert labels to one-hot encoding for categorical
   crossentropy
10 y_one_hot = to_categorical(y)
11
```

Next step is to split the dataset into training and test. The input data is then normalized before testing.

```
1 # Split the data into training and testing sets
2 X_train, X_test, y_train, y_test = train_test_split(X,
   y_one_hot, test_size=0.2, random_state=42)
3
4 # Standardize the features (optional but often recommended)
5 scaler = StandardScaler()
6 X_train = scaler.fit_transform(X_train)
7 X_test = scaler.transform(X_test)
8
```

In the following section we create the network using the **Sequential** method. The network consists of a hidden layer with 8 nodes and a final layer of three

nodes for each possible classification category. For the hidden layer ReLu functions are used, and for the final layer the softmax activation function is used for classification.

```
1 # Create a sequential model
2 model = Sequential()
3
4 # Add the input layer and the first hidden layer
5 model.add(Dense(units=8, input_dim=4, activation='relu'))
6
7 # Add the output layer with softmax activation for
  classification
8 model.add(Dense(units=3, activation='softmax'))
9
```

After defining the model we compile it with an optimizer, in this case the adam optimizer, and then we define a loss function and metrics. Then we train the model with fit and split the training dataset into the training and validation set. Then train with batch sizes of 8 across 200 epochs.

```
1 # Compile the model with categorical crossentropy loss for
  multi-class classification
2 model.compile(optimizer='adam', loss='
  categorical_crossentropy', metrics=['accuracy'])
3
4 # Train the model
5 history = model.fit(X_train, y_train, epochs=200, batch_size
  =8, validation_split=0.2)
6
```

Having trained the model, we can evaluate the performance using the test dataset and then print the result.

```
1 # Evaluate the model on the test set
2 loss, accuracy = model.evaluate(X_test, y_test)
3 print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")
4
5 # Plot training and validation loss over epochs
6 plt.figure(figsize=(12, 6))
7
8 # Plot training & validation loss values
9 plt.subplot(1, 2, 1)
10 plt.plot(history.history['loss'])
11 plt.plot(history.history['val_loss'])
12 plt.title('Model Loss')
13 plt.xlabel('Epoch')
14 plt.ylabel('Loss')
15 plt.legend(['Train', 'Validation'], loc='upper right')
16
17 # Plot training & validation accuracy values
18 plt.subplot(1, 2, 2)
19 plt.plot(history.history['accuracy'])
20 plt.plot(history.history['val_accuracy'])
21 plt.title('Model Accuracy')
22 plt.xlabel('Epoch')
```



```
23 plt.ylabel('Accuracy')
24 plt.legend(['Train', 'Validation'], loc='lower right')
25
26 plt.tight_layout()
27 plt.show()
28
```

This example will output how the loss and accuracy improves across the training. Note, that this problem is linear and a neural network is a bit overkill. This simple example is to show all the parts needed to have a functional setup with plotting of the training process. We will build on this in further lectures.

1.4.3 Exercises

The exercises this week aim to introduce the basic ways of creating neural networks. We will take a look at three methods, with the latter one being more advanced and what you will need to master to have maximal customization control of your network. Relevant code for this week is found in the folder **week 1**. The network you need to recreate is the network we discussed in the previous section with one hidden layer with 8 nodes, 3 nodes in the final layer, and 4 input parameters. A working version of this network can be found in Python script `main.py`. This network you will have to recreate in different ways in this weeks exercises. If you can replicate the result from this example code using the method specified in the exercise you can consider the exercise complete. If you need help, consult the official documentation.

Exercise 1: Sequential class

In this exercise you need to use the sequential class as in this weeks example code. However, instead of using the `model.add` method you should create a list of layers and parse them directly to the sequential class. You should thus have a model definition like `model = Sequential([your_list])`. For this exercise the script `sequential.py` has been prepared and you need to implement code where the `TODO` comment is located.

Exercise 2: Functional interface

In this exercise you need to use the functional interface. Here you need to define your input as a layer and then the layers with neurons. Then you can define the model as `model = Model(inputs=input_layer, outputs=x)`. For this exercise the script `functional.py` has been prepared and you need to implement code where the `TODO` comment is located.

Exercise 3: Subclassing the model class

In this exercise you need to use Subclassing, where you create your network as a classing inheriting from the model class. Here you need to define your class as a subclass for the model class, e.g. `class Network(Model)`. Then you

need to declare your layers in the `__init__` function and then build the network by defining a function called `build` which takes `self`, and the input tensor as variables. Remember to return the result. Then you can define the model as `model = Network()`. For this exercise the script `subclassing.py` has been prepared and you need to implement code where the `TODO` comment is located.

2 Machine learning and methods of training

In this lecture we look at machine learning in a broader perspective to get a feeling for the different methods which one can use. Then we focus in on neural networks and deep learning, before looking at methods of training neural networks.

2.1 Machine learning

Before we go into advanced concepts of neural networks, we better take a step back to understand the bigger picture. What is the difference between machine learning (ML), artificial intelligence (AI), neural networks (NNs), and deep learning (DL)?

As it turns out, neural networks is an outlier in this list. Neural networks is a particular method of machine learning, just like linear regression and decision trees.

As shown in Figure 4, AI is the overarching name for algorithms that aim to imitate some kind of intelligence. A very simple AI that does not use any kind of fancy ML methods would be something like the following code:

```
1 if temperature > 20:
2     return 0
3 else:
4     return 1
5
```

This example code is very simple, but it shows a simple AI algorithm which for example could be used to control temperature based on some temperature reading. One can make a more complex logic, but this is how one can make basic AI algorithms, and such algorithms have often been used in the past for various of applications to control sensors or bots in computer games.

Machine learning is a more advanced subset of AI methods where the algorithm can learn from data. Such algorithms do not require developers to hard code logic into the algorithms, as the algorithm will create its own logic after been trained on data. For complex methods such as neural networks it leads to the black box issue, that the developer does not understanding how the algorithm makes its decisions, but can merely try to assert if the decision is correct or not.

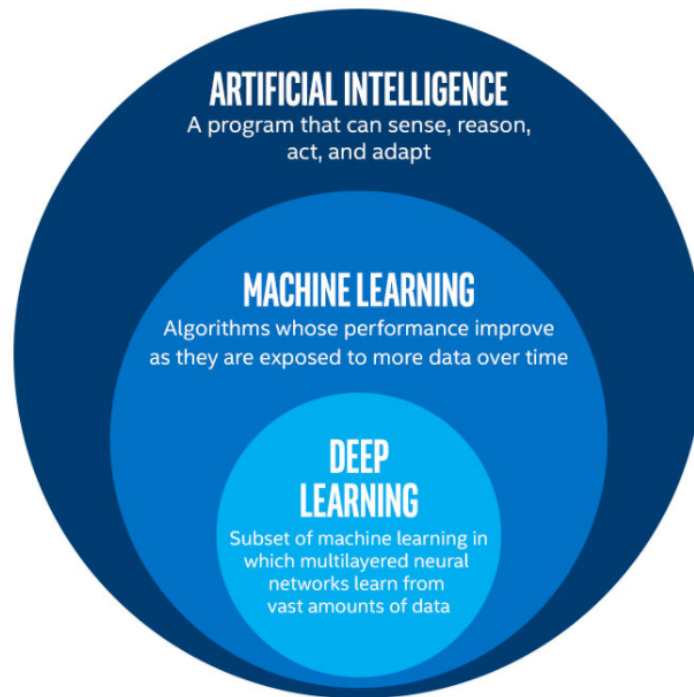


Figure 4: *An overview of the connection between AI, ML, DL. Note that neural networks are not mentioned.*

To give you an idea of ML methods here is a list of some well-known methods:

- Artificial neural networks
- Decision trees
- Genetic algorithms
- Support-vector machines
- Regression analysis
- Bayesian networks
- Gaussian processes

These different methods have different areas of applications, and many of them a valid in the same application areas. The training requirements may also differ for each method, and many of these methods can often be designed to trained in different ways.

2.2 Training of ML methods

Since the point of ML methods is to learn from data, and not work by parameters set by humans, a key part to this field is the way you train your ML method. In this section we will look at three main ways of training ML methods.

2.2.1 Supervised learning

One of the most popular ways of training neural networks is supervised training. It is very simple to do, and with more data you can generally make an increasingly good ML model with supervised learning and a ML method like a neural network. In this section we consider neural networks. Some of the pros of this method is:

- For classification we can decide the number of categories
- You can mimic anything given enough labeled data
- A trained model is often very resource effective
- We can have a good idea of the accuracy of the model

On the other hand it can also have the following downside:

- Requires a lot of labeled data
- Poor performance outside of the scope of the training
- Cannot detect new categories
- Training is time consuming

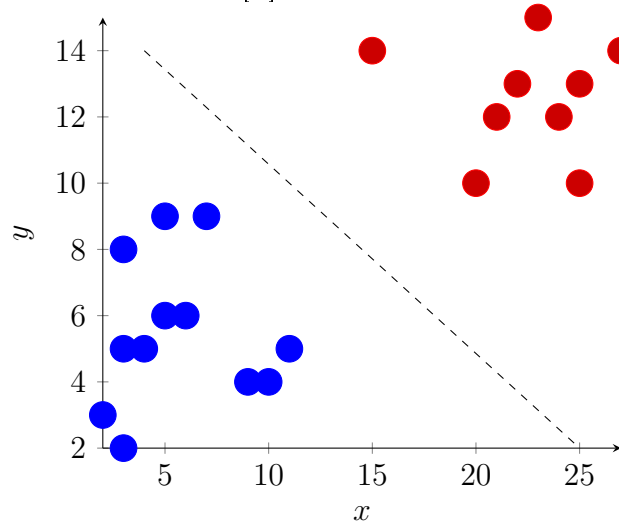
Consider an arbitrary function $f(x)$ of input x . The result of this will some value y . We are thus considering a function that maps and input x to y ,

$$f : x \rightarrow y. \quad (2.1)$$

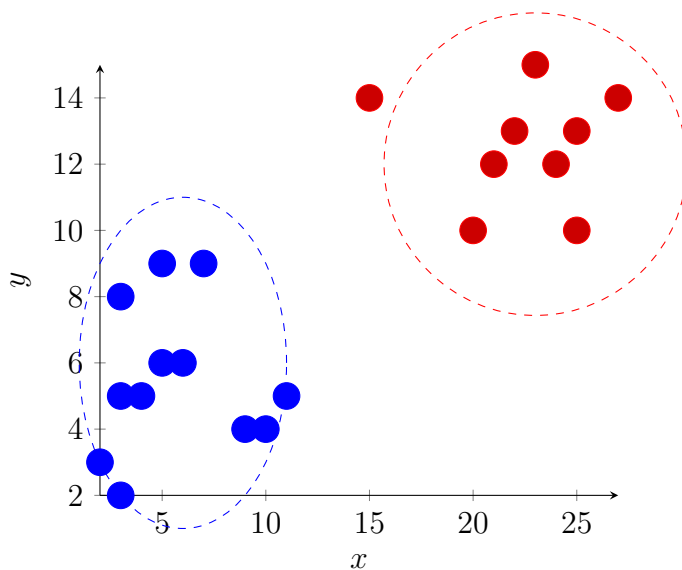
Given enough data we can train a neural network to become any function $f(x)$ and thus have the neural network imitate whatever we want. The main requirement is that our neural network have enough degrees of freedom and that we have an appropriate amount of training data.

When considering regression problems we can thus use a neural network. In some cases we might be model something using a very complicated or compute intensive mathematical model, fx fluid dynamics. In this case it is possible to make an approximate model with a neural network. It can often be orders of magnitude faster to compute a bit of matrix multiplication than solving differential equations, and with enough data one can get very close with the approximation one can do with a neural network, keeping mind that

the alternative model often also uses approximations to increase compute performance. TODO[?]



2.2.2 Unsupervised learning



2.2.3 Reinforcement learning

References

- [1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] François Chollet et al. Keras. <https://github.com/fchollet/keras>, 2015.

- [3] Jan Nagler, Anna Levina, and Marc Timme. Impact of single links in competitive percolation. *Nature Physics*, 7(3):265–270, January 2011.
- [4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [5] Òscar Lorente, Ian Riera, and Aditya Rana. Image classification with classic and deep learning techniques, 2021.