**Artificial Intelligence - ARTIN**

École Centrale de Nantes

# Perceptron and CNNs Laboratory Report

*Authors:*
LANCHA João
LOIZOU Ioannis

Date: December 18, 2023

## Abstract

In this laboratory, we are given the ”Train cat vs non-cat” dataset and a Jupyter notebook with exercises and steps to follow. The goal of this activity is to build a simple binary classifier to distinguish images of cats and images that are not of cats based on neural networks. This is done in a two-step approach. First, we create a single-neuron model, implementing most of the functions ourselves and relying only on NumPy, to better understand the logic and how to work with neural networks. Then, we use the Keras module from TensorFlow to build a convolutional neural network (CNN).

# List of Figures

# List of Tables

# Contents

Chapter 1

# Single Neuron Model

For this task, we are given a dataset of 259 images of size 64x64 and 3 channels. The first step is vectorizing and splitting the data into train and test subsets (209 images for training and 50 for testing). Once the images are vectorized, each pixel now contains 3 values (1 per channel), therefore the images are vectors of size 1x12288. Figure 1 presents an illustration of a single-neuron neural network for a 3-dimensional input vector.

Figure 1: Illustration of a single neuron neural network with a 3-dimensional feature input.

Based on the diagram in Figure 1, we can clearly see the necessary steps to perform a classification with such a model.

- Initialize weights $w_i$ and bias $b$ (we use random weight initialization and bias = 0).
- Define our activation function ($\sigma(x) = \frac{1}{1+e^{-x}}$).
- Predict ($y_{\text{pred}} = \mathbf{w}^T \cdot \mathbf{x} + b$).
- Estimate the cost using binary cross-entropy loss.

Each of these steps was defined in a function in our Jupyter notebook. Next, we perform a forward pass with the help of NumPy. Everything is vectorized, so we only need 3 lines of code, calling each of the previously defined functions.

```
w, b = initialize_parameters(train_x.shape[0])
neuron_output = neuron(w, b, train_x)
Y_pred = sigmoid(neuron_output)
```

Before the backward pass to update the weights, we need to define the compute the loss function. As previously stated, we use a binary cross-entropy loss, defined as:

$$E = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(y_{\text{pred}}^{(i)}) + (1-y^{(i)})\log(1-y_{\text{pred}}^{(i)}) \tag{1.1}$$

We then put this inside a Python function that takes as inputs the ground truths and the predictions and outputs the cost as a scalar.

Now we have everything needed to perform our backpropagation, as seen in the lectures, so we can compute the gradients of the loss function with respect to the weights and the biases. Again, this is done by declaring a Python function that takes as inputs the training set (X, Y) and the current predictions and outputs the gradients.

To calculate the gradients, we use the following definitions:

$$\frac{\partial E}{\partial w_i} = \frac{\mathbf{x} \cdot (\mathbf{y}_{\text{pred}} - \mathbf{y})^T}{m} \tag{1.2}$$

$$\frac{\partial E}{\partial b_i} = \frac{\sum_{i=1}^{m} (y_{\text{pred}}^{(i)} - y^{(i)})}{m} \tag{1.3}$$

We then define a function that takes as inputs the dataset, a desired number of iterations, and a learning rate and performs the gradient descent on each of the biases and weights. Using a learning rate of 0.005 over 2000 iterations, the cost had the following improvement.

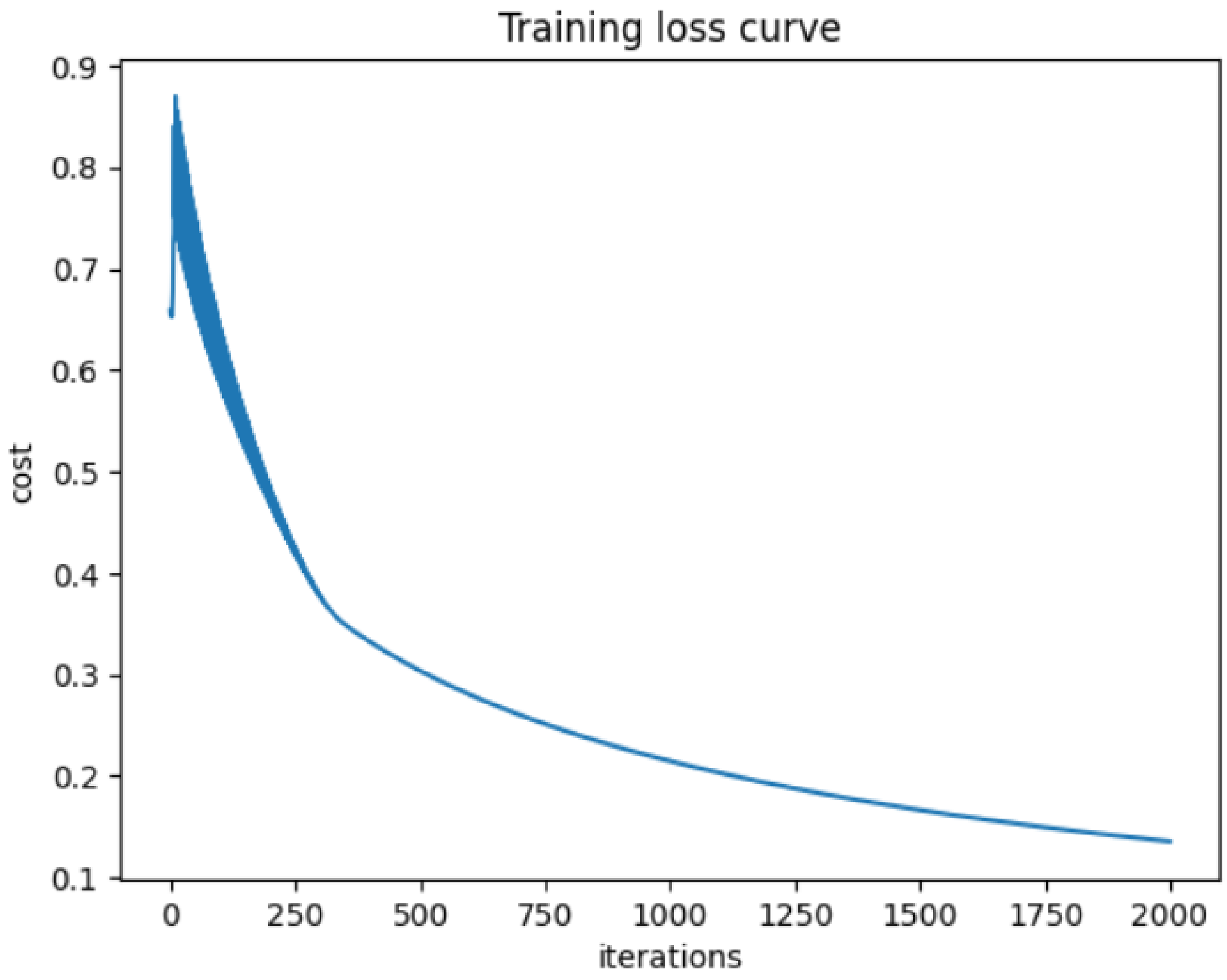

Figure 2: Evolution of the loss function at each iteration of the gradient descent.

We then predicted values using the optimized weights, which led to a training accuracy of 88.09% and a test accuracy of 64.55%.

However, the learning rate is a hyperparameter and, therefore, needs to be tuned. In order to successfully choose the best learning rate value, we took 10% of the training set to use as a validation set and performed the same evaluation using 2000 iterations per learning rate, with learning rates varying from 0.001 to 0.1 with a step of 0.005, and obtained the following.

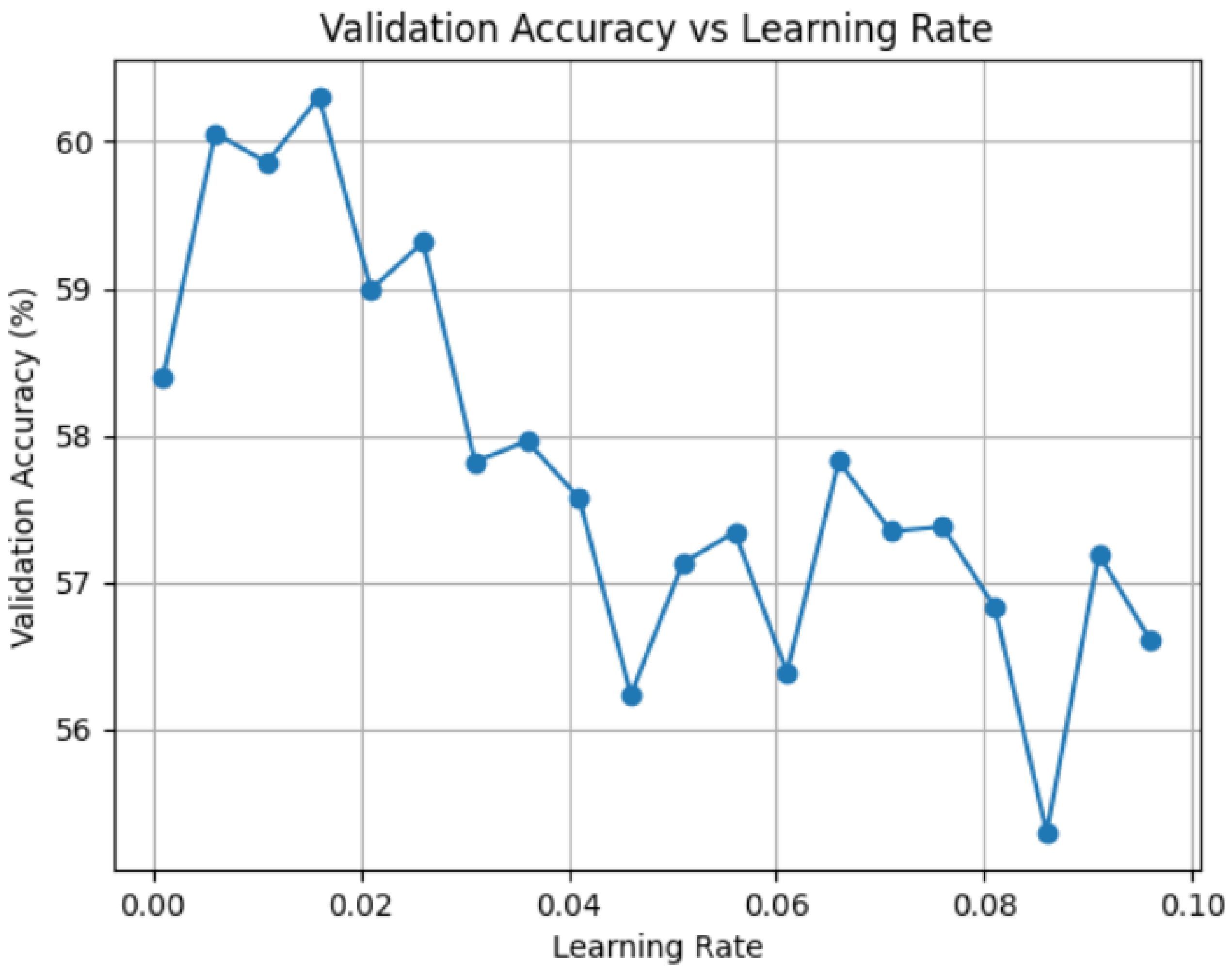

Figure 3: Validation accuracy x learning rate.

Using the best-performing learning rate (0.016) from the previous step, we can introduce it to the neural network and make predictions to compare with the case where the hyperparameter was not tuned. The results are shown in the table below.

| **Learning Rate** | **Training Accuracy** | **Test Accuracy** |
|---|---|---|
| 0.005 | 88.09% | 64.55% |
| 0.016 | 96.28% | 65.54% |

Table 1: Learning Rate and Accuracy Results

This value of learning rate is not a fixed one. Once the weights are randomly initialized, a different initialization might cause the optimal learning rate to change. As previously stated by Kinton, random weights initialization is not the best method. Instead, he proposed a pre-training step, to better initialize a NN's weights.

Chapter 2

# Convolutional Neural Networks - CNNs

## 2.1 CNNs with Keras and Tensorflow

After understanding the principles and implementation of a single-neuron model, we now implement a CNN, using Keras (a Google-developed API for the implementation of neural networks) and TensorFlow, a library particularly adapted to training and performance tracking of neural networks.

The first step is to adapt the example on the given link to the Keras website to work with our dataset. We make the necessary changes, such as changing the input shape that is (64x64x3), the activation function, and the loss function, among other parameters that can be seen in the Jupyter notebook. We then end up with the following model:

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        # Layer 1
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        # Layer 2
        layers.MaxPooling2D(pool_size=(2,2)),
        # Layer 3
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        # Layer 4
        layers.MaxPooling2D(pool_size=(2,2)),
        # Layer 5
        layers.Flatten(),
        # Layer 6
        layers.Dropout(0.5),
        # Layer 7
        layers.Dense(num_classes, activation="sigmoid")
    ]
)
```

Using the `model.summary()` function, we are able to go more in-depth into our model's characteristics. It has 44,482 parameters, all of them being trainable ones, that can be broken down in the following fashion:

- **Layers 2, 4, 5, and 6** do not introduce any parameters
- **Layer 1:** A convolutional layer with 32 3x3 filters.

  - Total Parameters $(3 \times 3 \times 3 \times 32 + 1 \times 32) = 896$ Parameters;

- **Layer 3:** A convolutional layer with 64 3x3 filters (input has 32 channels).
  - Total Parameters $(3 \times 3 \times 32 \times 64 + 1 \times 64) = 18{,}496$ Parameters;

- **Layer 7**: Dense $(2 \times 12544)$
  - Total Parameters = 25,090 Parameters

We then train the model with a mini-batch approach, using a batch size of 32 samples and 15 epochs, by calling the `fit()` function, which stores all the optimized weights and biases.

Afterwards, we define three other CNN architectures and train them using the same mini-batch approach. We compute the results in the ROC curve, which, as discussed in the previous report, is a very good evaluation measure to compare different classifiers.

- **Original**: The model proposed by the professor
- **Wider**: The second convolutional layer has increased size of kernel which is $5 \times 5$.
- **Deeper**: One extra convolutional layer is introduced and dropout between the convolutional layers to prevent overfitting.
- **No pooling**: The pooling layers are removed.

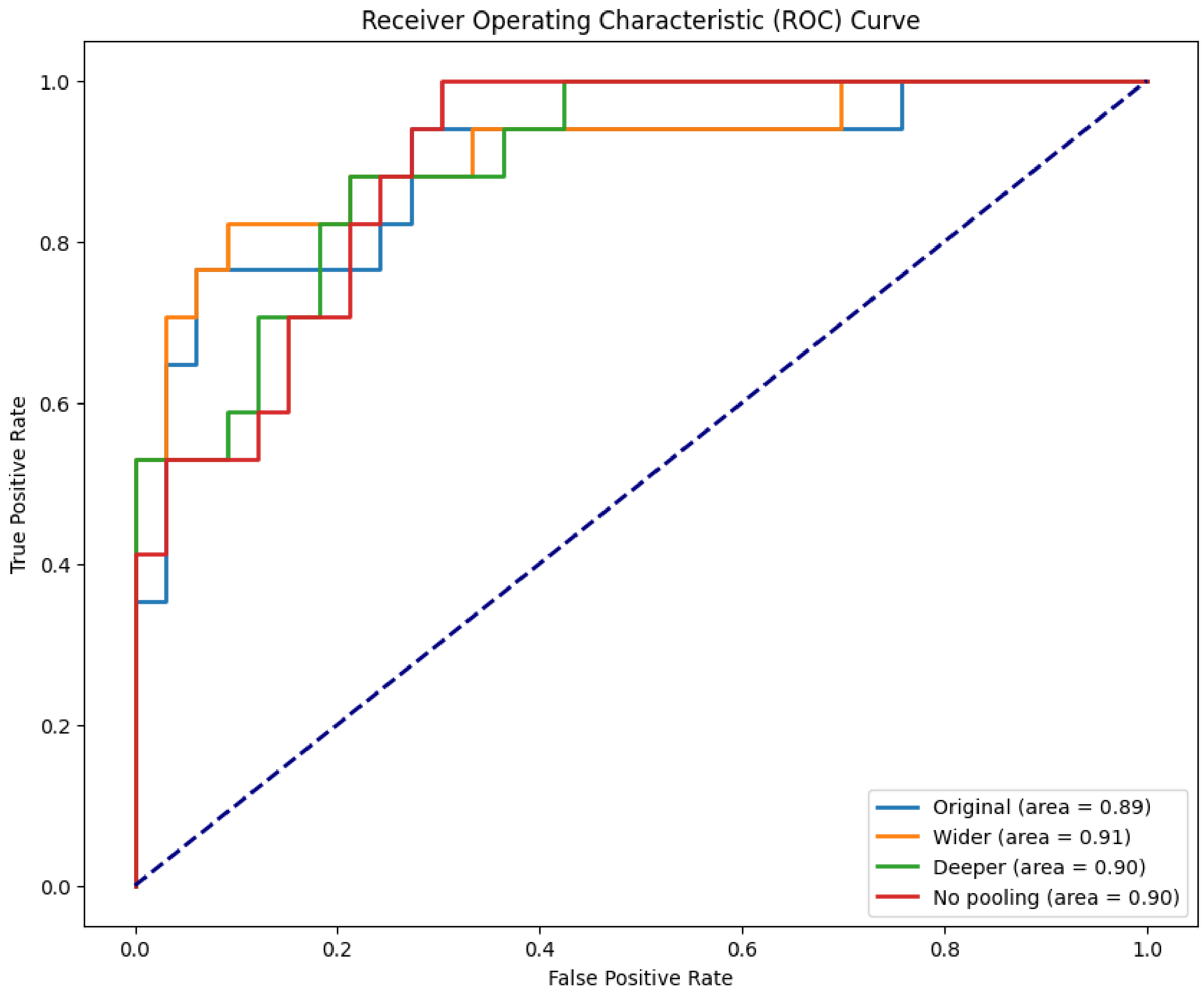

Figure 4: ROC curves for the 4 different CNN architectures tested

## 2.2 Custom training loop

For this part we replace the `fit()` function by our own tensorflow implementation. The optimizer selected was Adam with learning rate of 0.001. For the loss function Binary Cross Entropy was chosen and for the accuracy Binary Accuracy.

For the early stopping a callback function was implemented to allow the model to stop before the full number of epochs is completed. This is usefull because the validation loss may start getting worse so we need to stop earlier to prevent overfitting.

Tensorboard was also implemented using a callback function and then is called to visualized various data about the neural network model in detail.

For GradCam code tensoflow documentation was used. The reason to use GradCam is to indentify which pixels are more important during the optimaztion with the gradient descent. By creating a heatmap of the picture we can see which parts of the picture are more imporant for the neural network to indendify the cat. In our case we can clearly see in the figure below that the eyes are the most important part.

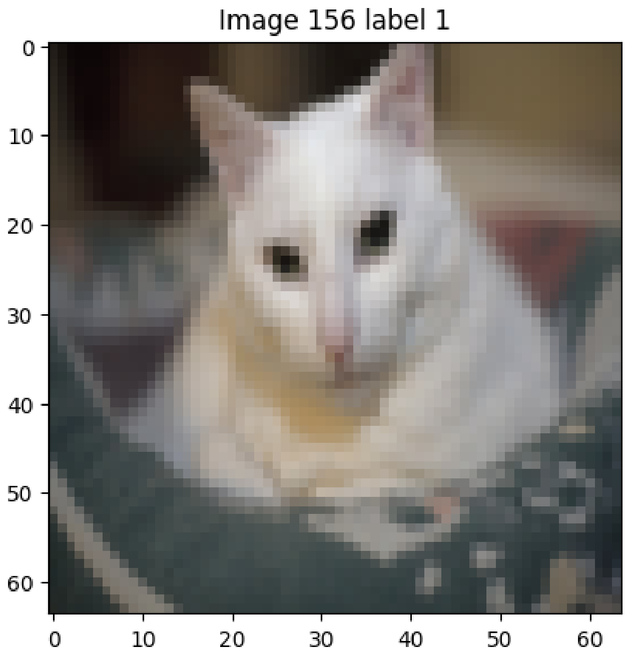

Figure 5: The picture of a cat.

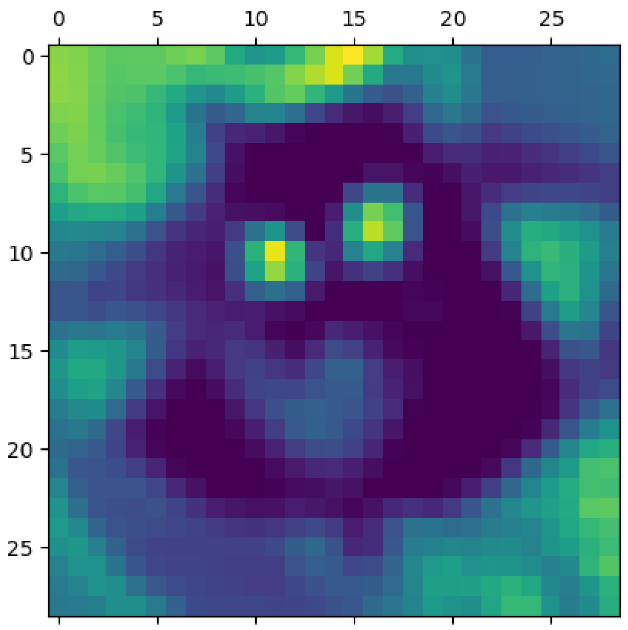

Figure 6: Heatmap of the picture using viridis scale.

# Conclusion

In this project, we showed how we can apply NN's and CNN's for image classification and how to improve the results. As we showed, hyperparameter tuning is an important step that should not be neglected. Otherwise, the model will not perform optimally. Additionally, neural networks are complex and powerful machine learning tools, that have thousands of possible architectures and parameters, and a good knowledge of their principles and applications is vital to correctly choose a suitable solution for a given problem. Therefore, an exercise like this, where we can test and compare different aspects of neural networks is highly enriching.

# Bibliography

[1] Mateus, Diana. *Artificial Intelligence*. Centrale Nantes, 2023.

[2] Wikipedia. `https://en.wikipedia.org`.

[3] scikit-learn. `https://scikit-learn.org`.