University of CINCINNATI

AUDREN ADRIEN
2023-2024

Deep Learning:
Homework 2

Convolutional Neural Network: MNIST dataset

Content

I. Data

A. Description

The MNIST dataset is composed of 70 000 images and their labels, including 60 000 training images and 10 000 testing images. Each image has a fixed size of 28x28 pixels and 1 channel (grayscale).

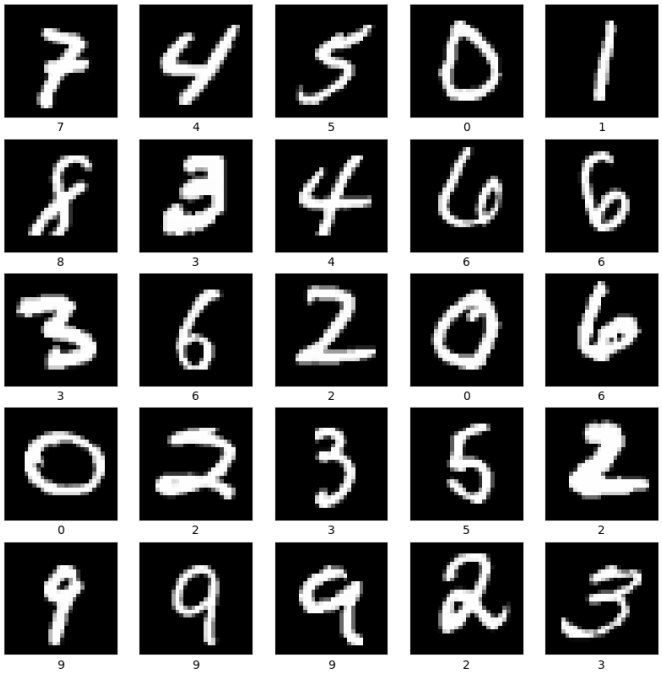

Figure 1.1: First 25 normalized images of training set.

1) There is a total of 70 000 images in this dataset.
2) The goal of this dataset is to enable a model to learn to recognize handwritten digits from 0 to 9.
3) Since each pixel has a grayscale value ranging from 0 to 255, all values are included in these boundaries.
4) Each data sample is an image of 28x28 pixels ranging from 0 to 255 in one channel.
5) Trying to define what is a missing information for this dataset is highly subjective since the digits are written by humans and thus are not perfect. Missing information could improve the robustness of our model if we consider a digit difficult to read as containing missing information.
   However, we can consider that the dataset only contains digits recognizable by humans and correctly labeled.
6) The labels are digits from 0 to 9.
7) The training and testing data is already defined by the dataset since this is a famous dataset commonly used by beginners. The fraction used for split is standard: $1/7 \approx 15\%$ for the test set and the rest for training.
8) Data is usually centered and reduced before each data analysis or treatment. Here, with grayscale's images, to center the data is not that relevant since pixels values are limited to $[\![0,255]\!]$. We will simply use a common standard of division by the max value (i.e. 255)

II. Model

A. Deep Neural Network

A first approach to create a model able to process the images is to "flatten" all of them to obtain a one-line vector of pixels values. Then we use a classic deep neural network, not that different from those used for regression but instead of a single number output we are using a n-values output vector with n the number of categories. Eventually we apply a Softmax function to assign to each image a vector of belonging probabilities.

$$image \rightarrow network\ layers \rightarrow \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} \xrightarrow{\text{Softmax}} \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \rightarrow \begin{pmatrix} p_1 \\ p_2 \\ \cdots \\ p_n \end{pmatrix}$$

Output — Belonging probabilities

```
model = models.Sequential([
      layers.Flatten(input_shape=(28, 28, 1)),
      layers.Dense(16, activation='relu'),
      layers.Dense(32, activation='relu'),
      layers.Dense(64, activation='relu'),
      layers.Dense(10, activation='softmax')
  ])
```

Figure 2.1: Architecture of the deep neural network

B. Convolutional Neural Network

The most common used model for deep learning images classification. The principle is to stack alternatively layers of convolution and layers of max pooling to extract relevant features which will enable the model to classify input images. A fully connected layer is added at the end and linked to n nodes to be able to apply the softmax function as seen previously.

We won't explain in detail how these layers work, but we will give some keys of understanding:

Regarding the convolution layer, a two-dimension kernel composed of weights which will be updated through the process of learning is "scanning" each feature map and apply a convolution operation with element-wise multiplication. A rectified linear unit function is applied to each value since a pixel value cannot be negative and this way, we are bringing non linearities to our model.

For the max pooling layer, the maximum values include in a predefined square are extracted and grouped to form the output image. This step is very close to subsampling and usually reduce by a factor of the size of the predefined square the size of the image processed.

A brief dive into how Tensorflow deals with trainable parameters and batch: When we are calling Conv2D operator with Keras API and specify the number m of output batches, Keras create

$m \times number\ of\ input\ channels$ kernels and m bias. We are applying a different kernel to each channel, add them together with one same bias for each channel and we obtain one output channel. Repeat this process m times to obtain the m channels desired. Assuming our kernels are 3x3 kernels and by noting c the number of input channel, there are $3 \times 3 \times m \times c + m$ trainable parameters.

Consider the following convolutional neural network (CNN):

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Figure 2.2: Architecture of a convolutional neural network

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 26, 26, 32)        896
```

Figure 2.3: Details of the first layer of the 2.2 model

$$3 \times 3 \times 32 \times 3 \ + \ 32 = 896$$

With MNIST dataset, we only have one channel. The number of trainable parameters obtained for the first layer of the previous model will be 320.

C. Residual Neural Network

i. Model explanation

When using a deep neural network with numerous layers, the model can experience some struggle to flow the gradient. This is especially true for the first layers during backpropagation phase. Indeed, when applying gradient descent on the first layers, we must use a super long multiplication of partial derivatives because the error between our model output and the true label depends indirectly on the weights and bias of the first layers. Computing the derivative with respect to the first weights requires us to derivate a composed functions of all the layers until we reach the last layer.

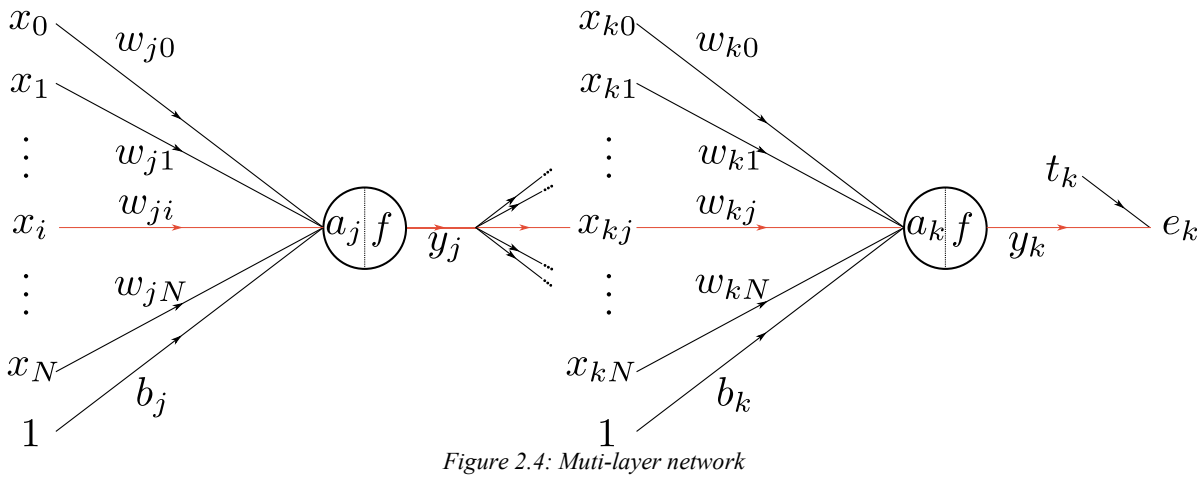

Figure 2.4: Muti-layer network

Assuming that the layers of this model are defined the following way:

$$[layer_j, layer_{j+1}, \dots, layer_{k-1}, layer_k]$$

The chain rule should look like this:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial e_k} \times \frac{\partial e_k}{\partial y_k} \times \frac{\partial y_k}{\partial a_k} \times \frac{\partial a_k}{\partial x_k} \times \frac{\partial x_k}{\partial y_{k-1}} \times \frac{\partial y_{k-1}}{\partial a_{k-1}} \times \frac{\partial a_{k-1}}{\partial x_{k-1}} \times \dots \times \frac{\partial a_{j+1}}{\partial x_{j+1}} \times \frac{\partial x_{j+1}}{\partial y_j} \times \frac{\partial y_j}{\partial a_j} \times \frac{\partial a_j}{\partial w_j}$$

The gradient can then either explodes or vanishes depending on the values we are multiplying together. To avoid vanishing of our gradient during the first layers' training we can add what we call a residual block. The idea is to add the output of the first layers to the output of the last layer. This way, when we will compute the gradient and thus multiplying small numbers together, we will still have a term that depends directly on the weights we want to update.

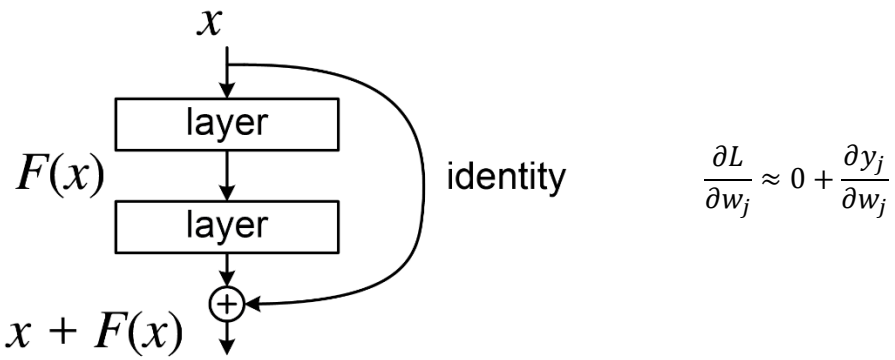

Figure 2.5: Residual block explanation

```
input_layer = tf.keras.layers.Input(shape=(28, 28, 1), name="input_layer")
l1 = layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1), padding="same", name="target_layer")(input_layer)
l2 = layers.MaxPooling2D((2, 2), padding="same")(l1)

sc_l = layers.Conv2D(128, (1, 1), activation='relu', strides = (4, 4), padding="same")(l2)  # shortcut layer

l3 = layers.Conv2D(64, (3, 3), activation='relu', padding="same")(l2)
l4 = layers.MaxPooling2D((2, 2), padding="same")(l3)
l5 = layers.Conv2D(64, (3, 3), activation='relu', padding="same")(l4)
l6 = layers.MaxPooling2D((2, 2), padding="same")(l5)
l7 = layers.Conv2D(128, (3, 3), activation='relu', padding="same")(l6)

l8 = tf.keras.layers.Add()([l7, sc_l])

l9 = layers.Conv2D(128, (3, 3), activation='relu', padding="same")(l8)
l10 = layers.MaxPooling2D((2, 2), padding="same")(l9)

l11 = layers.Flatten()(l10)
l12 = layers.Dense(128, activation='relu')(l11)
l13 = layers.Dense(10, activation='softmax')(l12)
return tf.keras.Model(input_layer, l13)
```

Figure 2.6: Residual neural network architecture

Because of the max pooling operations occurring before the identity is added, we must make that identity the same size of the output image of the layers we skipped. This can be done by using a 1x1 convolution with strides.

ii. Gradient vanishing visualization

Tensorboard is a powerful tool of metrics visualization and can help us to track measurements during the learning workflow. We are going to use this extension to track the values of kernels coefficients during the training process and we will compare these measurements when using a residual block and when not using one. We built the exact same model presented on figure 2.6 but without a residual block and we will analyze differences in learning process, if any for the first layer (name = "target_layer") because it is the most likely to suffer from gradient vanishing.

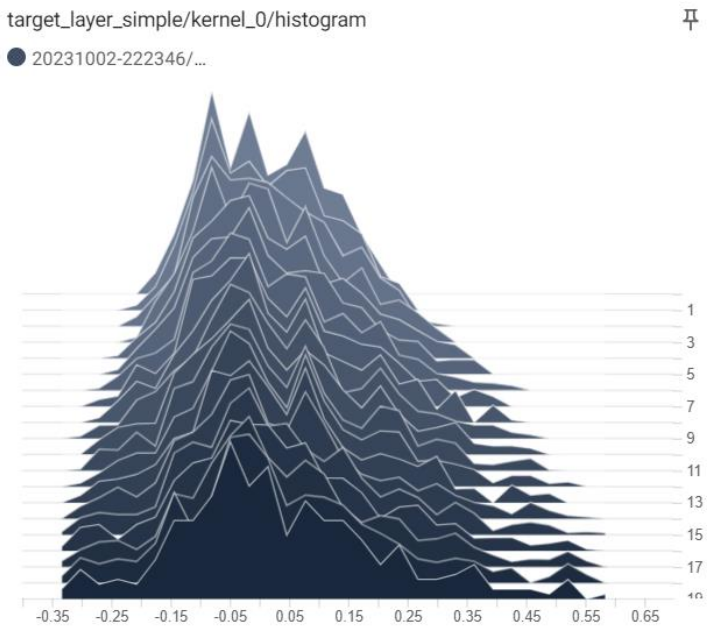

Figure 2.7: histogram of target kernel through 20 epochs (without residual block)

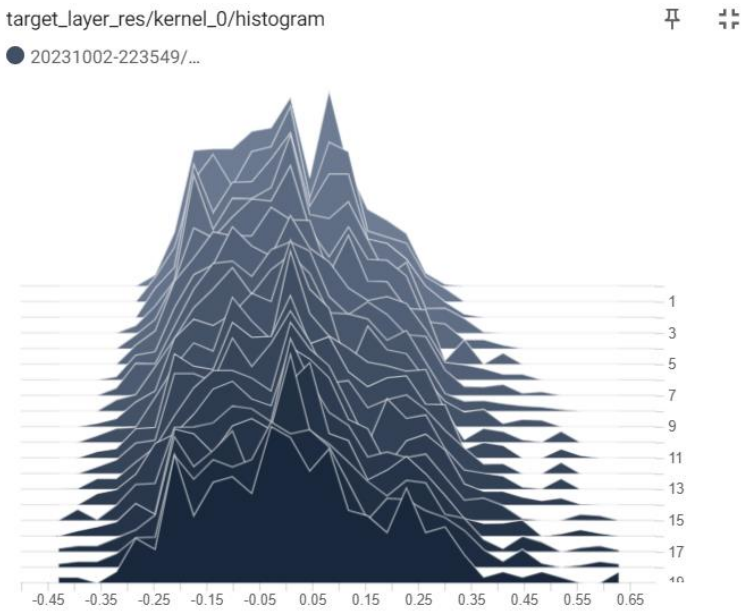

Figure 2.8: histogram of target kernel through 20 epochs (with residual block)

Even though the model with a residual block does not have a significant evolution of its first kernel, The coefficients whose value is located around 0.15 seem to evolve more quickly when there is a residual block. To see more significant differences, a more complex data set and more sophisticated models with many layers could help us to visualize the vanishing of gradients.

D. Model's Accuracy

Model	**Accuracy**
DNN	0.959
ConvNet	0.987
ResNet	0.991

Table 2.1: Registered model's accuracies obtained with a 0.001 learning rate value and 10 epochs on test data.

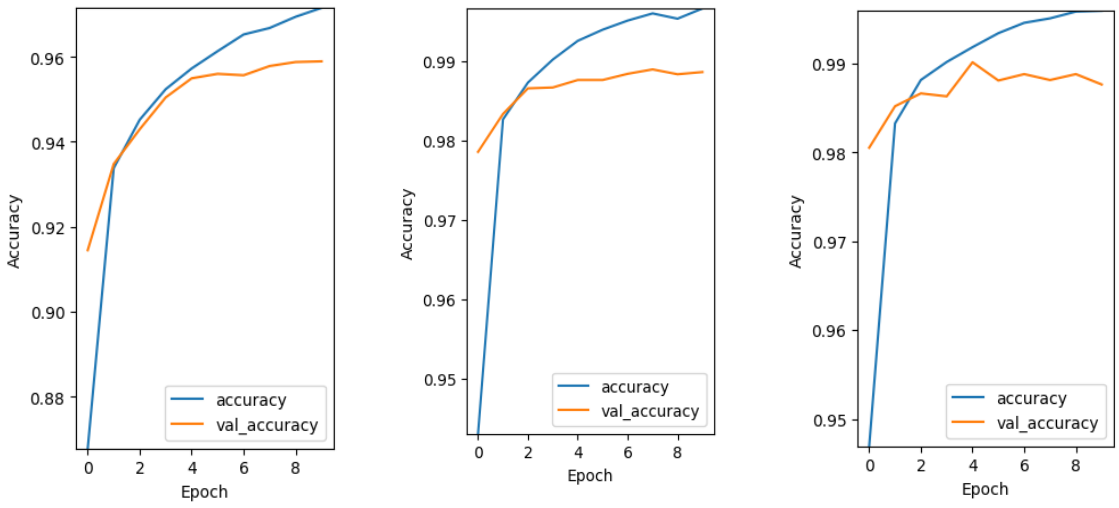

Figure 2.9: Accuracy of DNN (respectively ConvNet and Resnet) during training.

III. Objective

Let's get back to a binary classification problem. There are n examples (x_i, y_i) with x_i the feature of example i and y_i its label. This is our learning set since we are interested in a supervised learning problem.

(x_i, y_i) are drawn independently. We want to build a model parametrized by θ and able to classify the best way these examples. The probabilistic approach is to maximize the following probability:

$$P[(Y = y_1|X = x_1) \cap (Y = y_2|X = x_2) \cap \ ... \cap (Y = y_n|X = x_n)]$$

Where Y represents the output class of our model and X the input feature with which we feed to this model. When $X = x_1$ we want our model to return the correct class y_1 with the maximum probability and this for all our examples.

By independence, we have:

$$P\left[\bigcap_{i=1}^{n}(Y = y_i|X = x_i)\right] = \prod_{i=1}^{n} P\,(Y = y_i|X = x_i)$$

Since this is a binary problem, we can use a Bernouilli model. Let $P(Y = 1|x_i, \theta)$ be the probability that our model θ returns the class 1 when the input feature is x_i (respectively $1 - P(Y = 1|x_i, \theta)$ for returning class 0 when we have x_i). We will call this product the likelihood of the model since it represents how our model fits with reality. The previous quantity is equal to :

$$L = \prod_{i=1}^{n} P(Y = y_i|x_i, \theta)^{y_i}(1 - P(Y = y_i|x_i, \theta))^{1-y_i} \qquad (1)$$

We are searching for the best parameters:

$$\underset{\theta}{\text{argmax}} \prod_{i=1}^{n} P(Y = y_i|x_i, \theta)^{y_i}(1 - P(Y = y_i|x_i, \theta))^{1-y_i}$$

There are many reasons why using a logarithm function. One of them is for practical use. In machine learning, multiplying values together between zero and one can led to a numerical underflow. Since the logarithm is an increasing function, the problem is equivalent to:

$$\underset{\theta}{\text{argmax}} \sum_{i=1}^{n} y_i \log\big(P(Y = y_i|x_i, \theta)\big) + (1 - y_i)\log(\,1 - P(Y = y_i|x_i, \theta))$$

$$= \underset{\theta}{\text{argmin}} \quad -\sum_{i=1}^{n} y_i \log\big(P(Y = y_i|x_i, \theta)\big) + (1 - y_i)\log(\,1 - P(Y = y_i|x_i, \theta)) \qquad (2)$$

This is called the log-likelihood. This metric allows us to deal with binary classification problems using a logistic regression. In fact, in machine learning the probability $P(Y = y_i|x_i, \theta)$ is obtained using the sigmoid function:

$\sigma(a) = \frac{1}{1+e^{-a}}$, where a is the output returned by our model. This function then gives us a belonging probability. It belongs to class 1 with probability $P(Y = 1|x_i, \theta) = \sigma(a = f(x_i))$.

But what about multi-classes problems. The problem is no longer binary. We use as many logistic regressions as there are classes. Assuming k represents a class and f_k the logistic regression model

used to know if x_i belongs to class k or not. We then classify the feature x_i in the classes whose $\sigma(a = f_k(x_i))$ is the highest.

We can gather the left and right term of (2) using the indicator function. For each example i, we compute the following metric:

$$-\sum_{c \in classes}^{m} \mathbb{1}_{x_i \in c} \log\big(P(Y = c|x_i, \theta)\big)$$

We can either use a softmax function to obtain these probabilities: $P(Y = c|x_i, \theta)$. By summing over all the examples, we obtain:

$$\underset{\theta}{\operatorname{argmin}} \; -\sum_{i=1}^{n} \sum_{c \in classes}^{m} \mathbb{1}_{x_i \in c} \log\big(P(Y = c|x_i, \theta)\big)$$

This loss function, established to deals with multi-classes problems and commonly called "cross-entropy", will be used to train our models.

IV. Optimization

A default optimizer on Tensorflow. Adam is efficient enough to lead optimization of our model's parameters.

V. Model Selection

Model	**LR: 0.1**	**LR: 0.01**	**LR: 0.001**	**LR: 0.0001**
DNN	0.114	0.932	0.962	0.935
ConvNet	0.114	0.978	0.991	0.988
ResNet	0.098	0.958	0.99	0.99

Table 5.1: Accuracies obtained with different learning rate values and 10 epochs on test data.

VI. Model Performance

Many metrics can be used to evaluate the performance of a model. Among them, precision and recall which allows us to measure respectively how relevant a model is (i.e.among all the try, many of them are correct) and what is the reach of a model (i.e. is the model succeeding to retrieve the majority of relevant instances?). The right balance between these two indicators is the F_1-score (obtained from the F measure and the harmonic mean parameter $\alpha = \frac{1}{2}$).

$$F_1 = 2\frac{Recall \times Precision}{Recall + Precision}$$

Note that these types of metrics can only be used when facing a binary problem (True or False). Regarding a multi-class problem, we have to use a "one versus rest" approach. With the MNIST classification problem, for the 5 digit case, a true positive is an example labeled 5 and the model returned five. A false positive is an example labeled 5 and the model returned anything but 5, etc…

In addition to this metric, we can analyze a ROC curve.

"An Roc curve plots the true positive rate or sensitivity (= recall) against the false positive rate (or 1-specificity). […] An ROC curve always goes from the bottom left to the top right of the graph. For a good system, the graph climbs steeply on the left side."

Introduction to Information Retrieval
Manning – Raghavan – Schütze

The area under this curve (AUC) can be a good indicator as it represents the ability for a model to be efficient when predicting. We have a dual indicator for precision called MAP (Mean Average Precision).

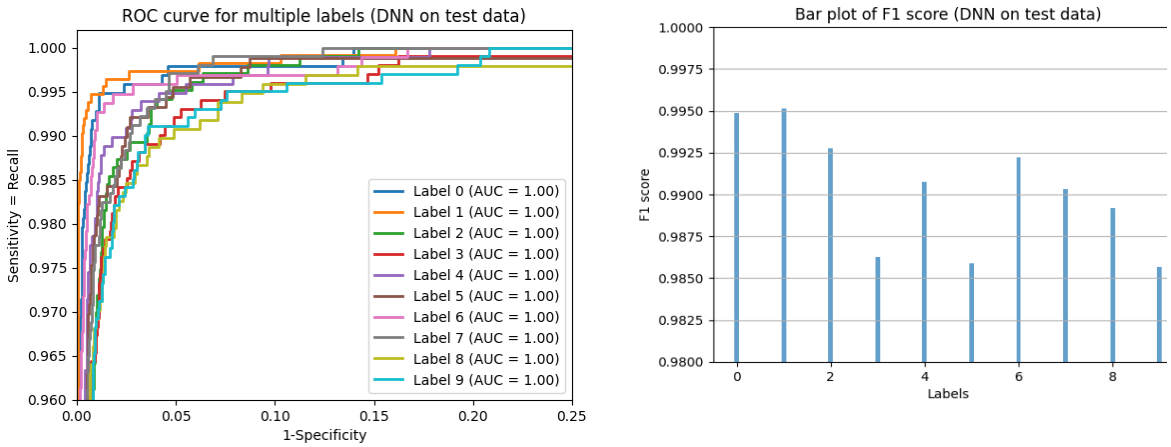

Figure 6.1 – 6.2 : ROC – F1 for DNN on test data

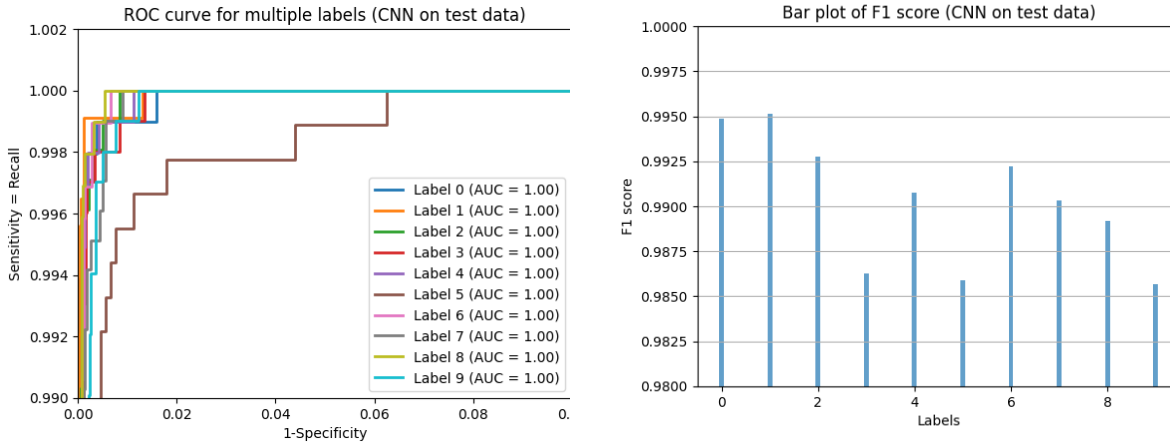

Figure 6.3 – 6.4 : ROC – F1 for CNN on test data

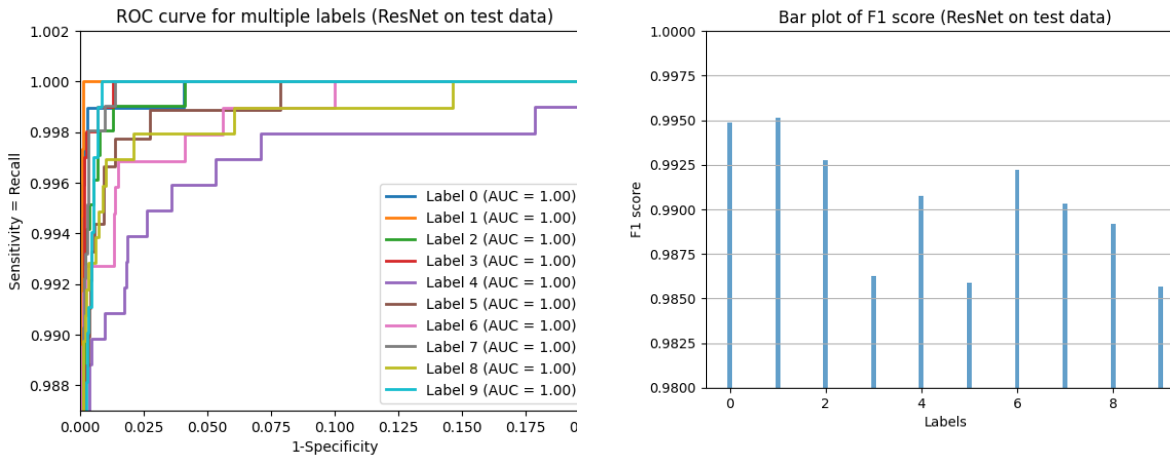

Figure 6.5 – 6.6 : ROC – F1 for ResNet on test data

Conclusion

All the three models we have presented in this report perform well. With the right learning rate and an adapted number of epoch(~10), they all succeed to reach more than 95% accuracy on test data. The figures of part 6 (ROC and histograms) do not clearly show many differences between these three models in terms of performances. When looking at accuracy scores, the deep neural network performs a little bit under the two others but nothing of incredibly significant.

Regarding the residual neural network, residual blocks have been designed to tackle vanishing gradients that appear when there are numerous layers stacked together, which is not the case here because our models remained simple due to a basic and simple data set. It is a safe bet that with more complex tasks and data set, that require a more sophisticated model with many layers, a residual neural network will be a preferred solution.