Handwritten digit classification using ANN / CNN

1. Objectives

This laboratory is focused on the issue of handwritten digit classification using artificial neural networks (ANN) or convolutional neural networks (CNN). By using the programming language Python and some dedicated machine learning platforms (*i.e.,* Tensorsflow with Keras API) the students will implement a digit recognition framework and will determine the system performances with respect to objective evaluation metrics such as: accuracy and loss.

2. Theoretical aspects

Convolutional Neural Networks (CNN) are very similar to ordinary Artificial Neural Networks (ANN) from the previous laboratory: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.

The ANNs don't scale well to full images. If we consider an image of size 28 x 28 x 1 (28 wide, 28 high, 1 color channel), so a single fully-connected neuron in a first hidden layer of a regular ANN would have 28*28*1 = 784 weights. If we consider the ANN structure presented in Fig.2.1 with 2 hidden layers of 16 neurons, and an output layer of 10 neurons. The total number of training parameters (weights) of the network can be computed as:

$$784 \cdot 16 + 16 \cdot 16 + 10 \cdot 16 = 12960$$

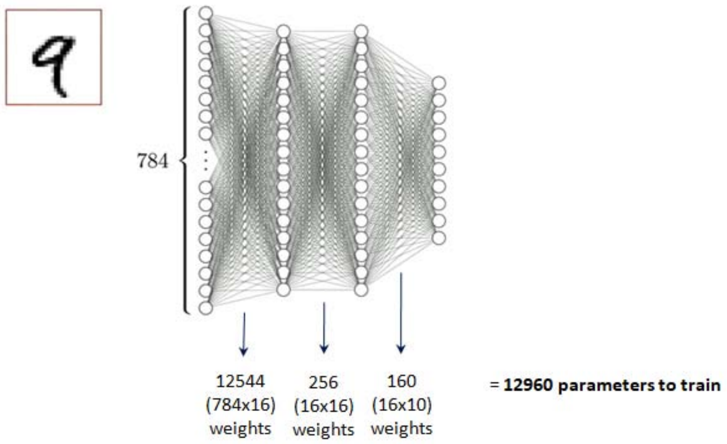


Fig. 1. ANN training for image classification

This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, *e.g.* 1280 x 720 x 3, would lead to neurons that have 1280*720*3 = 2.764.800 weights. And if we consider the same ANN architecture as presented in Fig. 2.2 the total number of parameters that require training will reach to: 44.237.216. As it can be observed the parameters would add up quickly to millions of elements! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting. In addition, the ANNs losses the spatial features of an image and cannot be trained on batches of images.

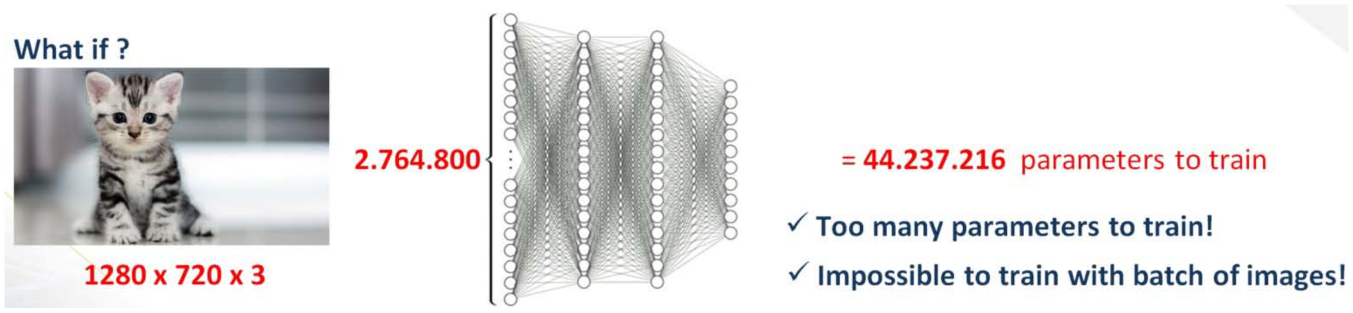

What if ?
1280 x 720 x 3
2.764.800
= 44.237.216 parameters to train
✓ Too many parameters to train!
✓ Impossible to train with batch of images!

Fig. 2.2. ANN training for large scale image classification

A Convolutional Neural Network (CNN) is a deep learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and is able to differentiate between different images (Fig. 2.3).

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. A ConvNet is able to successfully capture the *spatial dependencies* in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better. In particular, unlike a regular ANN, the layers of a ConvNet have neurons arranged in 3 dimensions: *width, height, depth*.

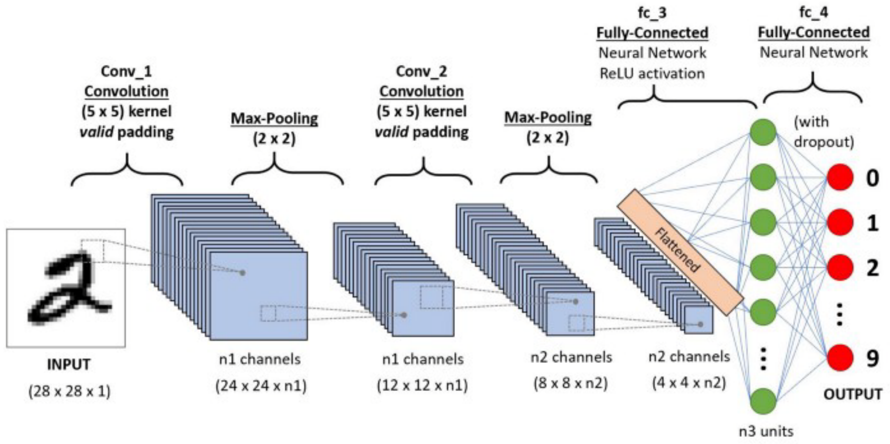


Fig. 2.3. A CNN architecture used for handwritten digits recognition

In the classical framework a basic CNN is composed of a set of operations such as: convolutions, max pooling, dropout, flatten and fully connected layers (Fig. 2.3).

3. Practical implementation

Pre-requirements: Copy the HandwrittenDigitRecognition.py Python script in the project “ArtificialIntelligenceProject” main directory (developed in the laboratory: “AND/OR gates implementation using artificial neural networks (ANN)”).

The following example aims to determine the optimal solution for classifying the handwritten digits form the MNIST dataset. In the MNIST database the images of digits were taken from a variety of scanned documents, normalized in size and centered.

Each grayscale image in the dataset is a 28 by 28 pixel square (784 pixels total). A standard split of the dataset is used to evaluate and compare models, where 60.000 images are used to train a model and a separate set of 10.000 images are used to test it.

Application 1: Consider the MNIST dataset, write a Python script (using the dedicated image machine learning platforms (*i.e.*, Keras API)) able to recognize the digit from an image applied as input. There are 10 digits (0 to 9) or 10 classes to predict. The system performance needs to be evaluated using prediction error, which is nothing more than the inverted classification accuracy.

Step 1: *Load the MNIST dataset in Keras* - Keras deep learning library provides a convenient method for loading the MNIST dataset (`from keras.datasets import mnist`). The dataset is downloaded automatically the first time the function `mnist.load_data()` is called and is stored in your home directory in ~/.keras/datasets/mnist.pkl.gz as a 15MB file. Load the MNIST dataset using the following command:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

Step 2: *Baseline model with multi-layer perceptrons* – A very simple neural network model with a single hidden layer will be trained for recognition purposes. We will use this as a baseline for comparing more complex convolutional neural network models.

Write a Python function denoted **trainAndPredictMLP** that takes as input the training and the testing dataset with their associated labels. This function will be called in the main method.

Step 3: *Reshape the MNIST dataset* - The training dataset is loaded as a 3-dimensional array structure (number of images, image width and height). For a multi-layer perceptron model we must reduce the images down into vectors of pixels. In this case the 28×28 sized image will become a 784 input vector.

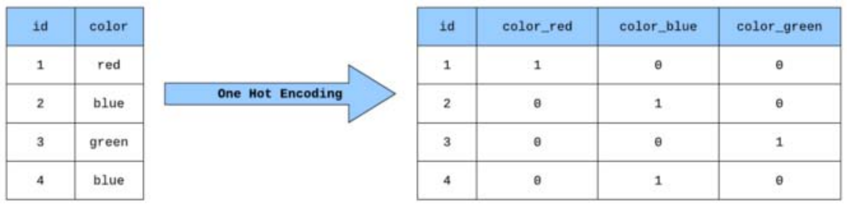
We can do this transform easily using the `reshape()` function on the NumPy array. The precision of the pixel values is set to 32 bit, the default precision used by Keras anyway.

```
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape((X_train.shape[0], num_pixels)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixels)).astype('float32')
```

Step 4: *Normalize the input values* – The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. We can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
X_train = X_train / 255
X_test = X_test / 255
```

Step 5: *Transform the classes labels (a vector of integers) into a binary matrix* – A label (an integer from 0 to 9) is associated to each image that corresponds to the digit. This is a multi-category classification problem. Although label encoding is a natural way of labeling data it has the disadvantage that the numeric values can be misinterpreted by algorithms as having some sort of hierarchy/order in them. In fact, using this encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results (predictions halfway between categories). In this case, it is a good practice to use one hot encoding of the class values, transforming the vector of class integers into a binary matrix. So, the integer encoded variable is removed and a new binary variable is added for each unique integer value (Fig. 3.1).



id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

Fig. 3.1. Data transformation using one hot encoding

In Fig. 3.1 the values in the original data are *red*, *blue* and *green*. We create a separate column for each possible value. Wherever the original value was *red*, we put a 1 in the *color_red* column.

For the MNIST dataset the data transformation from class labels to a binary matrix can easily be implemented using the built-in `np_utils.to_categorical()`function in Keras.

```
Y_train = np_utils.to_categorical(Y_train)
Y_test = np_utils.to_categorical(Y_test)
num_classes = Y_test.shape[1]
```

Step 6: *Build the model architecture* – We will define our model in a function `baseline_model`. This is handy if you want to extend the example later and try to get better accuracy scores. The function will take as input the size of the ANN input layer (784) and the number of classes of the output layer (10) and will return the model.

Step 6a: *Initialize the sequential model* - Every Keras model is either built using the Sequential class, which represents a linear stack of layers, or the functional Model class, which is more customizable. We will use the simpler Sequential model, since our network is indeed a linear stack of layers.

```
model = keras.models.Sequential()
```

Since we would like to build a standard feed-forward network, we only need the Dense type layer, which is a regular fully-connected network layer. Our model is a simple neural network with one dense hidden layer with 8 neurons and one output dense layer. A rectifier activation function is used for the neurons in the hidden layer. A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output prediction.

Step 6b: *Define a hidden dense layer with 8 neurons*

```
model.add(layers.Dense(8, input_dim=num_pixels,
          kernel_initializer='normal', activation='relu'))
```

Step 6c: *Create the output dense layer*

```
model.add(layers.Dense(num_classes, kernel_initializer='normal',
          activation='softmax'))
```

Step 6d: *Compile the model* – Before we can begin training, we need to configure the training process (within the model function `baseline_model`). We set 3 key factors for the compilation: the optimizer (*i.e.*, the Adam gradient-based optimizer), the loss function (because the ANN output contains a softmax layer, the cross-entropy loss it will be used) and the performance metric (since this is a classification problem, the accuracy metric will be used).

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
          metrics=['accuracy'])
```

The `baseline_model` function will return the compiled model object (`return model`).

Step 7: *Train the model* – Training a model in Keras literally consists only in calling the `fit()` function and specifying some training parameters as: training data (images and labels), the number of epochs to train (iterations over the entire dataset) and the batch size to use when training (number of samples per gradient update).

```
        model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=10,
batch_size=200, verbose=2)
```

The model is fit over 10 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the model performance during training. A verbose value of 2 will display on the console some information about the training process after each epoch.

Step 8: *System evaluation* – Finally, the test dataset is used to evaluate the model and a classification error rate is printed.

```
scores = model.evaluate(X_test, Y_test, verbose=0)
print("Baseline Error: {:.2f}".format(100-scores[1]*100))
```

Observation 1. The training process might take a few minutes when run on a CPU.

Observation 2. The results obtained on various computers may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Exercise 1: Modify the number of neurons on the hidden layer as specified in Table 1. What can be observed regarding the system accuracy? How about the convergence time necessary for the system to train a model?

Table 1. System performance evaluation for various numbers of neurons on the hidden layer

No of neurons	**8**	**16**	**32**	**64**	**128**
System accuracy					

Exercise 2: Modify the batch size used to train the model as specified in Table 2. Select a value of 8 neurons for the hidden layer. What can be observed regarding the system accuracy?

Table 2. System performance evaluation for different values of the batch size

Batch size	**32**	**64**	**128**	**256**	**512**
System accuracy					

Exercise 3: Modify the metric used to determine the system performance from accuracy to mean square error (mse). Has this parameter any influence over the training process?

Exercise 4: Using the default parameters specified at Application 1 (8 neurons in hidden layer, 10 epochs and a batch size of 200 images), train the model and save the associated weights in a file (`model.save_weights`), so it can be load anytime to perform further tests.

Exercise 5: Write a novel Pyhton script that loads the saved weights (`model.load_weights`) at Exercise 4 and make a prediction on the first 5 images of the testing dataset (`mnist.test_images()`).

Hint: First we have to specify the artificial neural network architecture as in Step 5 (Application 1), then load the set of weights (saved at Exercise 4) and make the prediction for the inputs using the `predict()` function. Keep in mind that the output of our network is the probability of that image to belong to the ten classes considered (because of `softmax`). So, it is necessary to use `np.argmax()` to return the class with the maximum probability score.

Application 1 has been focused on training and testing a simple multi-layer perceptron on the MNIST dataset. In Application 2 we will develop a more sophisticated convolutional neural network (CNN) having the same goal of handwritten digit classification.

Application 2: Consider the MNIST dataset. Write a Python script (using the dedicated image machine learning platforms (*i.e.*, Tensorsflow with Keras API)) able to recognize the digit from an image applied as input by using a CNN architecture. The system performance is evaluated using the prediction error, which is nothing more than the inverted classification accuracy.

Step 1: *Train and predict using a CNN network* - Write a Python function denoted **trainAndPredictCNN** that takes as input the training and the testing dataset with their associated labels (comment the line where the function trainAndPredictMLP (defined in Application 1) is called).

Step 2: *Reshape the MNIST dataset* - the MNIST dataset needs to be reshaped so that it is suitable for CNN training. In Keras, the layers used for two-dimensional convolutions expect input values with the dimensions [samples][width][height][channels]. In the case of RGB, the last dimension (channels) would be 3 for the red, green and blue components and it would be like having 3 image inputs for every color image. In the case of MNIST, each image in the dataset is a 28 by 28 pixel squares, where the pixel values are gray scale (the number of channels is set to 1).

```
# reshape the data to be of size [samples][width][height][channels]
```

Step 3: *Normalize the input values* – The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using convolutional neural networks. We can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
#normalize the input values from 0-255 to 0-1
```

Step 4: *Transform the class label (a vector of integers) into a binary matrix* – A label (an integer from 0 to 9) is associated to each image that corresponds to the digit. This is a multi-category classification problem. It is a good practice to use one hot encoding of the class values, transforming the vector of class integers into a binary matrix. This can easily do this using the built-in `np_utils.to_categorical()` function.

```
# one hot encoding - Transform the classes labels into a binary matrix
```

Step 5: *Build the model architecture* – We will define our model in a function named `CNN_model()`. This is handy if you want to extend the example later and try to get a better accuracy score. Convolutional neural networks are more complex than standard multi-layer perceptrons, so we will define a simple structure that uses all of the elements for state of the art results.

Step 5a: *Initialize the sequential model* - We will use the simple Sequential model since our network is a linear stack of layers.

The layers of the CNN architecture are presented below:

- (**Step 5b**) The first hidden layer is a convolutional layer called **Convolution2D** (Conv2D). The layer has 8 convolutional kernels, with the size of 3×3. This layer expects as input images with the structure presented above: [width][height][channels] (`input_shape=(28, 28, 1)`). The activation function is Relu (`activation='relu'`).
- (**Step 5c**) Next we define a pooling layer called **MaxPooling2D** that takes the maximum value of a block of size 2x2. The layer is configured by default with a pool size of 2×2.
- (**Step 5d**) The next layer is a regularization layer using dropout (called **Dropout**). Dropout is a technique where randomly selected neurons are ignored during training. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. The dropout layer is configured to randomly exclude 20% of neurons in order to reduce overfitting.
- (**Step 5e**) The next layer is called **Flatten** and converts the 2D matrix data to a vector. It allows the output to be processed by standard fully connected layers.
- (**Step 5f**) Next a fully connected (**Dense**) layer with 128 neurons and rectifier activation function is added (`activation='relu'`).
- (**Step 5g**) Finally, the output layer has 10 neurons for the 10 classes (`num_classes`) and a *softmax* activation function to output probability-like predictions for each class (`activation='softmax'`).

Step 5h: *Compile the model* – Before we can begin training, we need to configure the training process (within the model function `def CNN_model()`).We set 3 key factors for the compilation: the optimizer (*i.e.,* the Adam gradient-based optimizer), the loss function (because the CNN output contains a softmax layer, the cross-entropy loss it will be used) and the performance metric (since this is a classification problem, the accuracy metric will be used).

```
# Compile the model
```

Step 6: *Train the model* – Training a model in Keras literally consists only in calling the `fit()` function and specifying some training parameters as: training data (images and labels), the number of epochs to train (iterations over the entire dataset) and the batch size to use when training (number of samples per gradient update).

```
# Train the model
```

The model is fit over 5 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the model performance during training.

Step 7: *System evaluation* – Finally, the test dataset is used to evaluate the model and a classification error rate is printed.

```
# Final evaluation of the model - compute and display the prediction error
```

Exercise 6: Modify the size of the feature map within the convolutional layer as specified in Table 3. How is the system accuracy influenced by this parameter?

Table 3. System performance evaluation for various sizes of the convolutional filters

Size of the feature map	**1 x 1**	**3 x 3**	**5 x 5**	**7 x 7**	**9 x 9**
System accuracy					

Exercise 7: Modify the number of neurons on the dense hidden layer as specified in Table 4. Select a value of 3x3 neurons for convolutional kernel. What can be observed regarding the system accuracy? How about the convergence time necessary for the system to train a model?

Table 4. System performance evaluation for various numbers of neurons on the dense hidden layer

No of neurons	**16**	**64**	**128**	**256**	**512**
System accuracy					

Exercise 8: Modify the number of epochs used to train the model as specified in Table 5. Select a value of 128 neurons in the dense hidden layer. What can be observed regarding the system accuracy? How about the convergence time?

Table 5. System performance evaluation for different values of the number of epochs

No of epochs	**1**	**2**	**5**	**10**	**20**
System accuracy					

Exercise 9: Modify the CNN architecture with additional convolutional, max pooling layers and fully connected layers. The network topology can be summarized as follows:

- The first hidden layer is a convolutional layer called a **Convolution2D** (Conv2D). The layer has 30 feature maps, with the size of 5×5. This is the input layer, expecting images with the structure outline above [width][height][channels] (`input_shape=(28, 28, 1)`). The activation function is Relu (`activation='relu'`).
- Next we define a pooling layer that takes the max called **MaxPooling2D** of size 2x2. The layer is configured by default with a pool size of 2×2 (`pool_size=(2, 2)`).
- A convolutional layer called a **Convolution2D** (Conv2D). The layer has 15 feature maps, with the size of 3×3. The activation function is Relu (`activation='relu'`).
- A pooling layer that takes the max called **MaxPooling2D** of size 2x2. The layer is configured by default with a pool size of 2×2 (`pool_size=(2, 2)`).
- The next layer is a regularization layer using dropout called **Dropout**. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.
- Next is a layer that converts the 2D matrix data to a vector called **Flatten**. It allows the output to be processed by standard fully connected layers.
- Next a fully connected (**Dense**) layer with 128 neurons and rectifier activation function (`activation='relu'`).
- A fully connected (**Dense**) layer with 50 neurons and rectifier activation function (`activation='relu'`).
- Finally, the output layer has 10 neurons for the 10 classes (`num_classes`) and a *softmax* activation function to output probability-like predictions for each class (`activation='softmax'`).

Determine the system accuracy using the novel configuration of the CNN architecture.

OBSERVATION: The laboratory report (*.pdf file) will contain:

- The code for all the exercises. The code lines introduced in order to solve an exercise will be marked with a different color and will be explained with comments.
- Print screens with the results displayed in the PyCharm console.
- The exercises solutions: tables of results and responses to the questions.