Clothing classification using CNN

1. Objectives

This laboratory is focused on the issue of clothing classification using convolutional neural networks (CNN). By using Python as programming language and some dedicated machine learning platforms (*i.e.,* Tensorsflow with Keras API) the students will develop (from scratch), evaluate and use dedicated deep convolutional neural networks for the clothing classification problem. We show how to develop a robust test framework for estimating the performance of the model, how to improve the model, how to save the model and later load it to make predictions on new data. Finally, the system will be evaluated with respect to objective evaluation metrics such as: accuracy and loss.

2. Theoretical aspects

K-fold cross-validation – is a re-sampling procedure used to evaluate machine learning models on a limited data samples. The algorithm involves randomly dividing the set of observations into k groups, or folds, of approximately equal size. The first fold is treated as a validation set and the method is fit on the remaining k − 1 folds. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. When a specific value for k is chosen, it may be used in place of k in the reference to the model, such as k=10 becoming 10-fold cross-validation.

Cross-validation is primarily used in applied machine learning to estimate the performance of a machine learning model on unseen data. It is a popular method because it is simple to understand and because it generally results in a less biased or less optimistic estimation of the model skills than other methods, such as a simple train/test split.

The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups.
3. For each unique group:
    1. Take one group as a data set;
    2. Take the remaining groups as a training data set;
    3. Fit a model on the training set and evaluate it on the test set;
    4. Retain the evaluation score and discard the model.
4. Summarize the skill of the model using the sample of model evaluation scores.

Importantly, each observation in the data sample is assigned to an individual group and stays in that group for the duration of the procedure. This means that each sample is given the opportunity to be used in the hold out set one time and used to train the model k-1 times.

3. Practical implementation

Pre-requirements: Copy the FashionRecognition.py file in the project "ArtificialIntelligenceProject" main directory (developed in the laboratory: "Handwritten digit classification using ANN/CNN").

The following example presents all the steps necessary to develop from scratch a convolutional neural network (CNN) dedicated for clothing classification. The example uses the Zalando's Fashion-MNIST dataset. The database consists of a training set of 60.000 examples and a testing dataset of 10.000 examples. Each example is a 28 x 28 grayscale image, associated with a label from 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original handwritten digit MNIST dataset for benchmarking machine learning algorithms.

Fig. 1. Sample images taken from the Fashion MNIST dataset

Application 1: Consider the Fashion-MNIST dataset, complete the FashionRecognition.py script (using the dedicated machine learning platforms (*i.e.*, Tensorsflow with Keras API)) in order to be able to recognize the category of an unknown image applied as input. The mapping classes labels (integers from 0 to 9) are as follows: 0 - T-shirt/top, 1 - Trouser, 2 - Pullover, 3 - Dress, 4 - Coat, 5 - Sandal, 6 - Shirt, 7 - Sneaker, 8 - Bag and 9 - Ankle boot. The system performance will be evaluated using the accuracy metric.

Step 1: *Import the dependences necessary to develop the CNN architecture* – The following classes and functions will be needed in order to construct the CNN architecture:

```
from numpy import mean
from numpy import std
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
from keras.utils import np_utils
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
```

Step 2: *Load the Fashion-MNIST dataset in Keras* – The Keras deep learning library provides a convenience method for loading the Fashion-MNIST dataset (`from keras.datasets import fashion_mnist`). The dataset is downloaded automatically the first time the function `fashion_mnist.load_data()` is called and is stored in your home directory. Load the Fashion-MNIST dataset using the following command:

```
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
```

Some information about the dataset can be obtained by applying the following instructions:

```
# summarize loaded dataset
print('Train: X={}, Y={}'.format(trainX.shape, trainY.shape))
print('Test: X={}, Y={}'.format(testX.shape, testY.shape))
```

There are 60.000 examples in the training dataset and 10.000 images in the testing dataset. The images are indeed square with 28×28 pixels.

Exercise 1: In order to have a graphical description of the Fashion MNIST dataset display the first 9 images existent in the `trainX` variable using the OpenCV library.

Observation 1: As it can be observed the images are all pre-segmented (*i.e.*, each image contains a single item of clothing) and are presented in grayscale format.

Write a Python function denoted **prepareData** that takes as input the training and the testing image datasets (`def prepareData(trainX, trainY, testX, testY)`) and process the images as presented in the Steps 3-5. The function returns the same set of variables: `trainX, trainY, testX, testY.`

Step 3: *Reshape the Fashion MNIST dataset* - The Fashion 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 Fashion 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).

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. This involves first converting the data type from unsigned integers to floats (`astype('float32')`), then dividing the pixel values by the maximum value. In the case of Fashion - MNIST we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

Step 5: *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 clothing class. 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 be done using the built-in `np_utils.to_categorical()` function.

Step 6: *Build the model* – We will define our model in a function denoted `def defineModel()`. We will be using the Sequential model, since our network is a linear stack of layers. We start by instantiating a `Sequential` model:

```
def defineModel(input_shape, num_classes):

    # create model
    model = Sequential()
```

The model has two main aspects: the front end corresponding to the feature extraction (using the convolutional and pooling layers) and the backend represented by the classifier (for data prediction).

The layers of the CNN architecture are presented below:

- The first hidden layer is a convolutional layer called a **Convolution2D** (Conv2D). The layer has 32 feature maps, with the size of 3×3. 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'`), while the weights initialization is performed using He initialization (`kernel_initializer='he_uniform'`).
- 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.
- 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 16 neurons to interpret the low level features. The activation function is Relu (`activation='relu'`), while the weights initialization is performed using He initialization (`kernel_initializer='he_uniform'`).
- Because we would like to develop a multi-category classification, we know that we will require an output layer with 10 nodes in order to predict the probability distribution of an image belonging to each of the 10 classes. So, the output layer has 10 neurons for the 10 classes and a *softmax* activation function to output probability-like predictions for each class (`activation='softmax'`).
- Finally, we need to compile the model and configure the training process parameters. We decide 3 key factors during the compilation step: the optimizer, the loss function and the metric.

For the optimizer a conservative configuration for the stochastic gradient descent will be used with a learning rate of 0.01 and a momentum of 0.9 (`SGD(lr=0.01, momentum=0.9)`). Because the CNN contains *softmax* in the fully connected layer, the system will use as loss function the categorical cross-entropy (`loss='categorical_crossentropy'`). As performance metric we will monitor the classification accuracy (`metrics=['accuracy']`).

Step 7: *Train and evaluate the model* – The model training will be performed within a function denoted `def trainAndEvaluateClasic(trainX, trainY, testX, testY)`. The function takes as input the training and testing dataset together with the associated set of labels and returns the training accuracy score. Training a model in Keras literally consists only of calling `fit()` function and specifying some training parameters as: training data (images and labels), the number of epochs (iterations over the entire dataset) to train for or the batch size (number of samples per gradient update) to use when training.

```
def trainAndEvaluateClasic(trainX, trainY, testX, testY):

      accuracy = 0

      #TODO - Application 1 - Call the defineModel function
      model = defineModel((28, 28, 1), 10)

      # fit model
      model.fit(trainX, trainY, epochs = 5, batch_size=32, validation_data=(testX,
      testY), verbose=1)

       # evaluate model
      loss, accuracy = model.evaluate(testX, testY, verbose=1)
      print("Accuracy = {:.2f}%".format(accuracy* 100))

      return accuracy
```

Step 8: *Run the complete framework* – The framework will be defined within the main function.

```
def main():

     #TODO - Application 1 - Step 2 - Load the Fashion MNIST dataset in Keras
     (trainX, trainY), (testX, testY) = fashion_mnist.load_data()

     #TODO - Application 1 - Step 2 - Print the size of the train/test dataset
     print('Train: X={}, Y={}'.format(trainX.shape, trainY.shape))
     print('Test: X={}, Y={}'.format(testX.shape, testY.shape))

     #TODO - Application 1 - Call the prepareData method
     trainX, trainY, testX, testY = prepareData(trainX, trainY, testX, testY)

     #TODO - Application 1 - Step 7 - Train and evaluate the model in the classic
way
     trainAndEvaluateClassic(trainX, trainY, testX, testY, model)
```

Exercise 2: Modify the number of filters in the convolutional layer as specified in Table 1. How is the system accuracy influenced by this parameter? How about the convergence time?

Table 1. System performance evaluation for various numbers of filters in the convolutional layer

Number of filters	**8**	**16**	**32**	**64**	**128**
System accuracy					

Exercise 3: Modify the number of neurons in the dense hidden layer as specified in Table 2. The number of filters in the convolutional layer remains set to 32. What can be observed regarding the system accuracy?

Table 2. System performance evaluation for various numbers of neurons in the dense hidden layer

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

Exercise 4: Modify the number of epochs used to train the model as specified in Table 3. The number of neurons in the dense hidden layer is set to 16. What can be observed regarding the system accuracy? How about the convergence time?

Table 3. System performance evaluation for different values of the number of epochs

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

Exercise 5: Modify the learning rate of the stochastic gradient descend as presented in Table 4. Train the model for 5 epochs. What can be observed regarding the system accuracy and the convergence time?

Table 4. System performance evaluation for various learning rates of the SGD

Learning rate	**0.1**	**0.01**	**0.001**	**0.0001**	**0.00001**
System accuracy					

Exercise 6: Explore how adding regularization impacts model performance as compared to the baseline model. Add a **Dropout** layer on the CNN (after the **MaxPooling** layer) that randomly excludes 20% of neurons (`model.add(Dropout(0.2))`). Select for the learning rate a value of 0.01. Analyze the convergence speed with and without this regularization operation?

Modify the percentage of excluded neurons in the dropout layer as presented in Table 5. Select a value of 3x3 neurons for convolutional kernel. How is the system accuracy influenced by this parameter?

Table 5. System performance evaluation for number of neurons dropped in the dropout layer

Dropout percentage	0.1	0.2	0.3	0.4	0.5
System accuracy					

Exercise 7: Specify the optimal values for the following parameters (the number of filters in the convolutional layer, the size of the convolutional kernel, the number of neurons in the dense hidden layer, the number of epochs used for training, the learning rate) that maximize the system accuracy.

Exercise 8: Once fit, we can save the final model (architecture and weights) to an *.h5 file by calling the *save()* function on the model and pass in the selected filename (`model.save('Fashion_MNIST_model.h5')`). We can use our saved model to make a prediction on new images.

The model assumes that: (1) the new images are in grayscale and have been segmented; (2). contain one centered piece of clothing on a black background and (3) the size of the image is square of 28×28 pixels. Fig. 2 presents the query image for which the system is required to make a prediction. You can place the image in the working directory with the filename 'sample_image.png'.

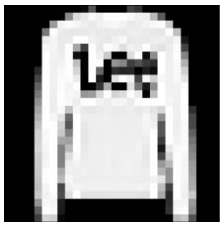

Fig. 2. Query image taken from the testing Fashion MNIST dataset

Using the pre-trained model (saved above) write a Python script able to make an automatic prediction regarding the category of the image presented in Fig.1. For this example, we expect class "*2*" that corresponds to a "*Pullover*".

Hint: First, load the image in the working space and convert it to grayscale format. In addition, impose the image size to be 28×28 pixels (as for the other images used for training). This process can be implemented within a `def load_image(filename)` function that will return the reshaped tensor `(1, 28, 28, 1)`, ready for classification purposes.

Importantly, the pixel values are prepared in the same way as the pixel values were prepared for the training dataset when fitting the final model, in this case, normalized. The CNN model saved before can be load using the following function: `model = load_model('Fashion_MNIST_model.h5')`, while the `model.predict_classes(img)` function can be used in order to make a prediction.

Application 2: Consider the example presented in Application 1. The model training has been performed on the entire training dataset (60.000 images). The evaluation has been performed on the testing dataset (10.000 images). In Application 2 a different training strategy will be considered by using three datasets: one for training, one for validation and the final one for testing purposes.

The training dataset can be regarded as a set of examples that are used for learning (data used to fit the model). The validation dataset is a set of examples used to tune the classifier parameters (*ex.* the number of hidden units in a neural network, the kernel of convolution filter …). More generically, the validation dataset contains data samples used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyper-parameters. The testing dataset represents data samples used to evaluate the final model and to determine the framework performances.

Step 1: *Multiple model training* – One popular method of creating a validation dataset is to use k-fold cross-validation algorithm. This will divide the training dataset in two parts: one used for the actual training of the model and the other that will be used to tune the model hyperparameters instead of a separate testing dataset. So, a function denoted `def trainAndEvaluateKFolds(trainX, trainY, testX, testY)` will be developed.

We first need to prepare the parameters of the cross validation data. The **KFold** class from scikit-learn can be used directly in order to split the dataset. The constructor takes as arguments the number of splits, (`k_folds`), whether or not to shuffle the sample data (`shuffle=True`) and the seed (`random_state=1`) for the pseudorandom number generator used prior to the shuffle.

For the current application, we can create an instance that splits a dataset into 5 folds, shuffles prior to the split and uses a value of 1 for the pseudorandom number generator. For 5 folds, the validation dataset is 20% of the training dataset (about 12.000 examples), close to the size of the actual testing set of the problem.

```
kfold = KFold(k_folds, shuffle=True, random_state=1)
```

Then, the training dataset `trainX` will be divided (`kfold.split(trainX)`) in two parts: `trainX_i` (for model training) and `valX_i` (for model testing) used to perform the validation of the model. For each possible split (`for train_idx, val_idx in kfold.split(trainX)`):

```
trainX_i = trainX[train_idx]
trainY_i = trainY[train_idx]
valX_i = trainX[val_idx]
valY_i = trainY[val_idx]
```

a model will be defined (`model = defineModel((28, 28, 1), 10)`) and trained. The baseline model will be trained for 5 training epochs with a default batch size of 32 examples.

```
history = model.fit(trainX_i, trainY_i, epochs=5, batch_size=32,
validation_data= (valX_i, valY_i), verbose=1)
```

The process will be repeated recursively and each model fit will be saved within a `histories` list. The test set for each fold will be used to evaluate the model.

We will later create learning curves and at the end we can estimate the performance of the final model. The performance score will be saved within a `scores` list. As such, we will keep track of the resulting history from each run, as well as the classification accuracy of the fold.

Finally, the function will return two lists of elements: with the models and with the performance accuracy scores:

```
return scores, histories
```

Step 2: *System performance presentation* – There are two key aspects that require a particular attention: the diagnostics of the learning behavior during training and the estimation of the model performance. These can be implemented using separate functions.

First, the diagnostics involve creating a line plot showing model performance on the train and validation sets during each fold of the k-fold cross-validation. These plots are important in order to determine if the model is overfitted, underfitted or if it has a good fit for the dataset.

Two figures will be displayed one for loss and the other for the accuracy. Green lines indicate the model performance on the training dataset, while the red lines indicate performance on the hold out validation dataset. The `def summarizeLearningCurvesPerformances(histories, scores)` function below creates and shows this plot given the collected training histories. Finally, the classification accuracy scores collected during each fold can be summarized by calculating the mean and standard deviation.

```
def summarizeLearningCurvesPerformances(histories, scores):

    for i in range(len(histories)):

        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='green', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='red',
    label='validation')
```

```
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(histories[i].history['accuracy'], color='green',
label='train')
    pyplot.plot(histories[i].history['val_accuracy'], color='red',
label='validation')

pyplot.show()

print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores) * 100, std(scores)
* 100, len(scores)))
```

Exercise 8: Adding padding to the convolutional operation can often result in better model performance, as more of the input image of feature maps are given an opportunity to participate or contribute to the output. By default, the convolutional operation uses '*valid*' padding, which means that convolutions are only applied where possible. This can be changed to `padding='same'` so that zero values are added around the input such that the output has the same size as the input. For Application 1, how is the system accuracy influenced by the padding operation?

Exercise 9: An increase in the number of filters used in the convolutional layer can often improve performance, as it can provide more opportunity for extracting simple features from the input images. This is especially relevant when very small filters are used, such as 3×3 pixels. By applying the padding operation (`padding='same'`) within the convolutional process, increase the number of filters (in the convolutional layer) from 32 to double that at 64. For Application 1, how is the system accuracy influenced by this parameter?