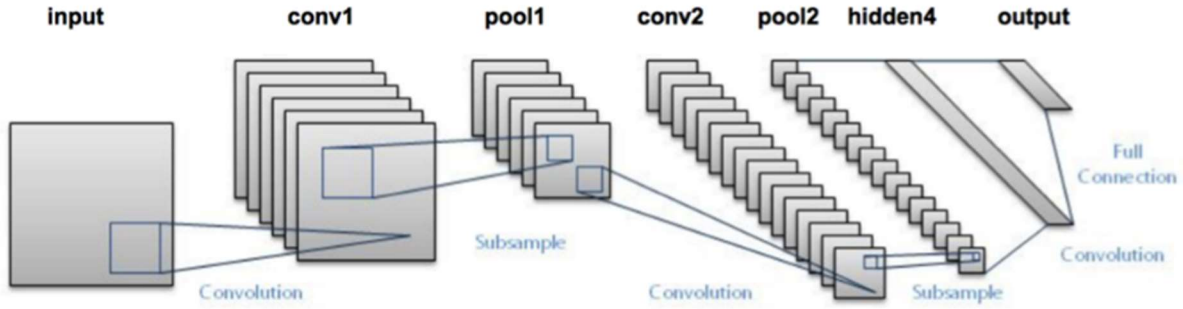
Sequence 6 exercises

Convolutional Neural Networks

N.B: For any CNN development, we are going to use the Keras library from Tensorflow. Help and description about Keras can be founded at *www.keras.io*.

1. CNN for digit recognition (homework)

The problem with the perceptron network is that it does not take into account the spatial organization of image pixels. But in an image, neighboring pixels are often correlated. The purpose of this first part is to implement the following CNN architecture for the purpose of handwritten digit recognition by taking into account the spatial pixel organization also.



Where conv1 is made of 32 filters of size 3x3 acting on 28x28 grey level images and followed by a relu activation function, pool1 is a maxpooling step of size 2x2, conv2 is made of 64 filters of size 3x3 followed by a relu activation function, pool2 is a maxpooling step of size 2x2, hidden4 is a fully connected layer with 128 neurons and the relu activation function and output is a fully connected layer with 10 neurons and the softmax activation function.

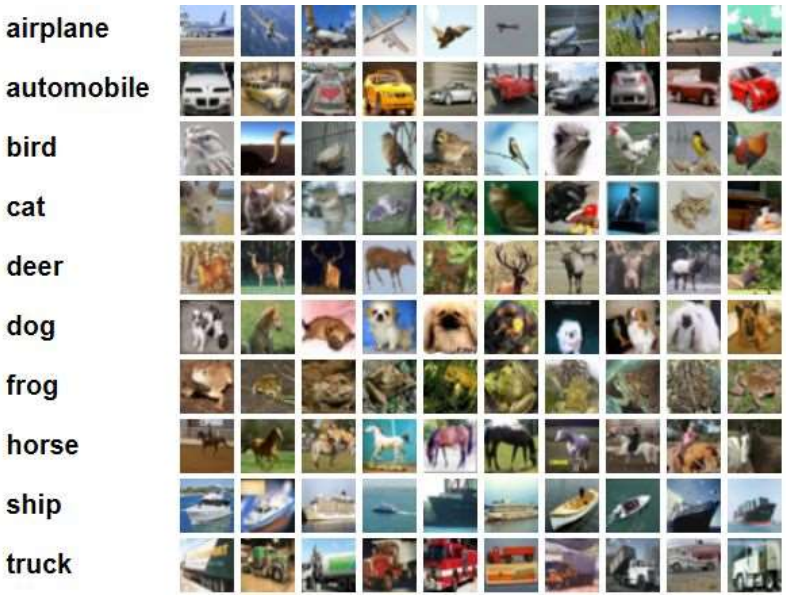
This network has been implemented in the file ***CNN for digit reco.py*** of the directory ***CNN_codes***.

1.1 Analyze the network complexity (cf. use the *model.summary()* in order to study the parameter numbers). Which is the most parameter demanding layer?

1.2 Run the program with the default hyper parameters and report the obtained accuracies. Comment the result and the accuracy and loss evolution curves. Compare the performances with those obtained when using a Neural Network only (cf using the perceptron). Conclusion?

2. CNN for objet classification (classwork)

In this section, the main goal will be image classification. We are going to work with the CIFAR 10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html). The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The 10 considered classes are the following:



In directory ***CNN_codes***, the file ***CNN student code for obj classif.py*** presents the global architecture of a program developed for designing, running and evaluating the performances of a CNN. This program is made of several steps:

- ✓ Data loading cf. *load_dataset()*
- ✓ Data normalization cf. *prep_pixels()*
- ✓ CNN design cf. *define_model()*
- ✓ CNN training monitoring cf. *summarize_diagnostic()*

2.1 Designing a basic network based on VGG structure

In order to design the CNN, a good starting point is the general architectural principles of the VGG model [1]. The architecture involves stacking convolutional layers with small 3×3 filters followed by a max pooling layer. Together, these layers form a block, and these blocks can be repeated where the number of filters in each block is increased with the depth of the network such as 32, 64, 128, 256 for the first four blocks of the model. Padding is used on the convolutional layers to ensure the height and width of the output feature maps match the inputs.

In the first attempt of classification, design a CNN with two VGG blocks made of the following structure for the feature extraction part:

- ✓ Con2D_1: 32 filters of size 3x3 and Relu function
- ✓ Con2D_2: 32 filters of size 3x3 and Relu function
- ✓ MaxPooling 2x2

After the first VGG block made of 32 filters of size 3x3, add a second one:

- ✓ Con2D_1: 64 filters of size 3x3 and Relu function

- ✓ Con2D_2: 64 filters of size 3x3 and Relu function
- ✓ MaxPooling 2x2

Regarding the classification part, put:

- ✓ A fully connected layer FC1 with 128 neurons and the relu function as the hidden layer
- ✓ A fully connected layer FC2 with 10 neurons and the softmax function as the output layer

2.1.1 Performances of the baseline model

Design this CNN by completing the *define_model()* function.

Run the program with batch size = 32 and epoch = 20 and SGD as the optimizer. Comment the obtained results, in particular the evolution of the loss and accuracy functions along with the iterations.

NB: in this exercise, the batch size is going to remain the same all the time. The purpose here is not to tune the batch size optimal value.

Rmk: the running process might be very long on your own computer. It is possible to use an external GPU in order to speed up the process. To do that, go to: https://colab.research.google.com/notebooks/intro.ipynb and connect to your google account. Colab offers the opportunity to execute python code on an GPU or on an TPU for free. To do that, you have to create a Jupiter notebook file (extension .ipynb).

Once your program is ready, before running it, please go to Execution Menu → Modify the execution type and in the window Notebook Parameter, choose GPU or TPU as material accelerator.

It should take around 5 min to train the model on Google Colab.

2.1.2 Changing the optimizer

Run the program again by changing the SGD optimizer by the Adam optimizer which is the optimizer which is currently used with CNN. Report and comment the performances.

2.1.3 Introducing batch normalization (BN)

Go back to the SGD optimizer and introduce a batch normalization step between the two first CONV layers and the two second CONV layers and just before the first fully connected layer. Run the new network and report the performances.

2.2 Improved model with regularization technics

For the following questions, the considered network is the baseline one with the SGD optimizer and without Batch Normalization.

2.2.1 Early stopping

The simplest solution in order to solve the overfitting problem is to stop the learning process as soon as the two loss and accuracy curves start to go in different directions. That's what is called "early stopping". Go back your first trial with the basic CNN. By analyzing the loss and accuracy curves, after how many epochs do you think it would be good to stop the learning process in order to limit overfitting? What are the performances then?

It is possible to make the program stop automatically when overfitting appears by adding the following lines in your code before the model fitting. Do not forget to import the *EarlyStopping*() function from the keras.callbacks module.

```
# simple early stopping
callbacks = [EarlyStopping(monitor='val_loss', patience=10)]
# fit model
print('Model training...')
history = model.fit(trainX, trainY, epochs=50, batch_size=32,
                    validation_data=(testX, testY), verbose=1, callbacks=callbacks)
```

If you look at the proposed code, you might notice that the number of epochs has been increased up to 50. And the line about *EarlyStopping* means that the validation loss is going to be monitored from epoch to epoch and if during 10 consecutive epochs (cf. patience=10), the curve increases, the algo will stop because it means that overfitting is arriving so it is better to stop the training right now.

Modify your own code in order to add *EarlyStopping* to the basic network and give the performances and the number of epochs before stopping. Comment.

2.2.2 Dropout

Another way to solve the problem of overfitting is to introduce some regularization process. Here we are going to use the dropout technic. Dropout is a simple technique that randomly drops nodes out of the network during the training step. It has a regularizing effect as the remaining nodes must adapt to pick-up the slack of the removed nodes. A new hyper-parameter is introduced which is the dropout probability.

Go back to the CNN and add a dropout step after the Maxpooling step and after FC1, the first fully connected layer. This is just a proposal. There is no theoretical method to know the optimal location and number of dropout steps. Use the *Dropout()* function from the keras.layers module and run the program with the dropout probability of 20% (20% of the nodes are randomly removed).

Run the program with 40 epochs. As a matter of fact, with dropout, the convergence is slower so it is necessary to increase the epoch number. Report and comment the results.

2.2.3 Data augmentation

Another way to prevent from overfitting is to increase the number of training data. Since it is not always possible to acquire (and label) new data, the idea is to proceed to what is called data augmentation. This means making copies of the training samples with small random modifications. This has a regularizing effect as it both expands the training dataset and allows the model to learn the same general features, although in a more generalized manner.

In our case, the types of random augmentations that could be useful include horizontal flips, minor shifts of the image, and perhaps small zooming or cropping of the image. Be careful not to distort the images too much so that the same label can be kept for the augmented samples.

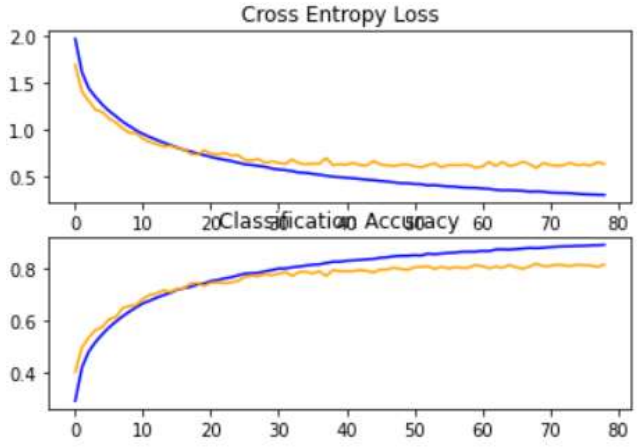
Rmk: the data augmentation process concerns the training dataset only and not the testing dataset.

Data augmentation can be implemented using the ImageDataGenerator class. For example, the code below implements simple data augmentation of the training set with horizontal flips and 10% shifts in the height and width of images.

```
# run the whole process to train and test the model
# load dataset
trainX, trainY, testX, testY = load_dataset()
# prepare pixel data
trainX, testX = prep_pixels(trainX, testX)
# define model
model = define_model()
# create data generator
datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1,
                         horizontal_flip=True)
# prepare iterator
it_train = datagen.flow(trainX, trainY, batch_size=32)
# Define now our callbacks
callbacks = [EarlyStopping(monitor='val_loss', patience=10)]

# fit model
steps = int(trainX.shape[0] / 32)
history = model.fit_generator(it_train, steps_per_epoch=steps, epochs=100,
                          validation_data=(testX, testY), verbose=1, callbacks=callbacks)
```

The file ***A piece of code for data_augment.py*** contains the code above. Open it and update your current file in order to take into account data augmentation. Start your code running to be sure that everything is fine and then stop it after the iterative process has begun if it is too slow. Unfortunately, even with using Colab, the training time is often very long. So you will find below if necessary the corresponding results after completing the process.



The process stops after 79 epochs and the performances are 92% for the train_acc and 82% for the test_acc. Comment the results.

2.3 Performance summary

Summarize your results in the following table. According to your experimentations and analyses, what is the best regularization technic to use?

Model	Training accuracy (%)	Testing accuracy (%)	Overfitting (Y/N)
Baseline CNN			
CNN + Adam			
CNN + BN			
CNN + dropout			
CNN + data augmentation			

2.4 Saving the model and making prediction

Once the model has been rightly learned, the estimated weights and biases have to be saved for further use. To do that, just add the following instruction at the end of your code: model.save('final_model.h5').

Select your best model and save it. Now we are going to use the learned model in order to predict the label of the following totally new image saved as ***sample_image.png***. This means that this image does not belong neither to the training set nor to the validation set.

Run the code ***CNN for obj_classif_prediction.py*** in order to use your model to predict the class of this image.

[1] K. Simonyan, A. Zisserman "Very Deep Convolutional Networks for Large Scale Image Recognition", ICLR proc., San Diego, USA, May 2015, https://arxiv.org/pdf/1409.1556.pdf