Practical session : Image classification

1 MNIST dataset classification

MNIST dataset contains 60 000 training images and 10 000 test images, of size 28×28. See examples figure 1.

Figure 1 – Extract of MNIST dataset

2 Classification using MLP

We first use the script `2_mnist_MLP.py` for classification, using MLP.

1) Complete the given script to implement a model with 1 hidden layer of 5 neurons with :

— reLU (rectified Linear Unit) activations in the hidden layer units
— categorical cross entropy loss.
— Adam optimizer, with a learning rate of 0.01.
— batch size of 100

2) How many parameters does the network have ?

3) What performance do you get ?

4) Observe the influence of different parameters on the final accuracy (number of epochs, batch size, learning rate, etc.).

5) Modify your model to have 2 hidden layers of 128 neurons each. What performances do you get ?

3 Classification using CNN

We now use the script `3_mnist_CNN.py`.

6) Build a neural network with :

— a convolutional layer with reLU activation, 32 convolutions 5×5 (stride 1×1).
— a max pooling layer 2×2 (stride de 2×2)
— a convolutional layer with reLU activation, 64 convolutions 5×5 (stride 1×1).
— a max pooling 2×2 (stride 2×2).
— a fully connected layer of 1024 neurons (activation ReLU)
— the output layer

All layers have a padding `SAME`

7) How many parameters does the network have ?

8) What performance do you get ?

Appendix : Introduction to Keras

This section gives a fast introduction to implementing neural networks with Keras.

Keras is a Python library which provides a high level API (programming interface) for implementing neural networksL It can be used on top of other libraries such as TensorFlow, CNTK ou Theano. Keras allows fast and easy neural network programming.

In Keras the reference data structure is the model. There are two main model construction modes in Keras :

— Sequential modeling, in which a model is defined as a stack of layers
— Functional modeling, which allows to build arbitrary graphs of layers for more complex models

Here we will only consider Sequential models.

Sequential models

To build a sequential model, you will import the kera module :

```
from keras.models import Sequential
```

The module `keras.layers` contains all the standard layers of neural networks (dense layers, convolutional layers, pooling layers, activations, dropout, ...)

```
from keras.layers import Dense, Activation
```

Model definition : Building the model is done by adding successive layers. For instance, a model with :

— one input layer of 100 neurons
— two hidden layers of 128 neurons with ReLU activation
— one output layer of 10 neurons with Softmax activation

will be written as :

```
model = Sequential()
model.add(Dense(units=128, activation='relu', input_dim=100))
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

The `Dense` layers used in this example correspond to fully connected layers. We will not review all layers here, but they are referenced in the online documentation. In practical sessions we will mainly use dense layers, convolutional layers and pooling layers. An example of convolutional layer is given here :

```
# 8 convolutions of size 3x3,
# with stride 1 and zero padding.
model.add(Conv2D(8, (3, 3), padding='same', strides=(1,1), input_shape=(28,28,1)))
# Activation ReLU
model.add(Activation('relu'))
# Max pooling of size 2 by 2, with stride 2
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2,2)))
```

Model definition is also the step where the weight initialisation method is defined. By default, the weight initialization is Uniform Xavier : weights are drawn randomly from a uniform distribution where the variance depends on the number of neurons of input and output of the layer. This is usually a good default choice, but if you want to change it, for example using to a Normal Xavier initialization you can use :

```
model.add(Dense(128, kernel_initializer='glorot_normal', bias_initializer='zeros'))
```

Biases usually stay initialized at zero.

Once the model built, it can be displayed using :

```
print(model.summary())
```

We can also visualize graphically the network (here registered in a .png file) :

```
import keras.utils.plot_model
plot_model(model,to_file='model.png',show_shapes=True,show_layer_names=True)
```

Training Configuration : Once the architecture definition is done, the training strategy is defined :

- the loss function) to minimize (mean squared error, cross entropy,...)
- the optimizer (SGD, RMSProp, AdaGrad, Adam...)
- the metric(s) computed to evaluate the performances. Metrics can be chosen from loss functions or other criteria such as accuracy, or even hand defined.

```
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

The choice of loss function and output activation function depend on the problem to solve (voir figure 3)

Problem	**Output**	**Output activation**	**Loss function**
Regression	Real	Linear	MSE : $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
Classification	Binary	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-(y\ log(\hat{y} + (1-y)log(1-\hat{y}))$
Classification	One label, several classes	Softmax : $f(z) = \frac{exp(z)}{\sum_i exp(z_i)}$	cross entropy : $-\sum_i^M y_i\ log(\hat{y}_i)$
Classification	Several labels, several classes	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-\sum_i^M (y_i\ log(\hat{y}_i + (1-y_i)log(1-\hat{y}_i))$

Figure 2 – Typical choices of loss function and output activation function in classification and regression problems

We can then train the model by defining the way data will be presented :

— One epoch correspond to one pass through all the data
— A batch is a subset of the dataset which will be used in one iteration of the optimization process. Thus the batch size will define the number of examples which will be used at each weight update step.

```
model.fit(x_train, y_train, epochs=3, batch_size=32)
```

In the script below, `x_train` and `y_train` are arrays of training data and labels.

Performances Evaluation : During training we observe the loss value and metrics values through the epochs (depending on the `verbose` mode) Once the model is trained, we can evaluate its performance on the test set : Une fois le modèle entrainé, on pourra évaluer sa performance sur les données de la base de test :

```
measures_perf_test = model.evaluate(x_test, y_test)
print('test results: ', measures_perf_test)
```

Using a trained network : To make predictions using a trained model :

```
predictions = model.predict(x_new)
```

Save : Saving weights :

```
model.save_weights("./trained_models/my_model")
```