COVIS Lab 3
Introduction to Deep Leaning

Prof. Vincent Frémont - ECN/LS2N

1 Introduction

The main objective of this lab session is to have a first contact with machine learning using both Fully Connected Neural Networks (FCN) and Deep Convolutionnal Neural Nets (DCNN). The output of this lab session is a short report (1 or 2 pages) containing performances comparison in terms of precision, time, etc, between the 2 approaches (FCN and DCNN). It is also recommended to make comment on the effects of varying parameters such as the number of layers, the size of convolutionnal mask, etc. This lab session is inspired from the TensorFlow website [1], and you can also find some help on the Keras website [2].

2 Keras install and TensorFlow test

All the session will be done using the TensorFlow environment (and so with Python) and the high-level layer Keras. So the first step is to install Keras using the console :

```
$ pip install keras
```

Now test the environment and some python libs like numpy and matplotlib. If they are not install, just do it with pip as before :

```
$ python
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> print(tf.__version__)
```

All the instructions you are testing can be written in a single python file (e.g. myFile.py) that can be run with :

```
$ python myFile.py
```

1. https ://www.tensorflow.org/tutorials/keras/basic_classification
2. https ://keras.io/#keras-the-python-deep-learning-library

3 The MNIST Dataset

To test your neural nets, you will use the MNIST dataset. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning, as you will use it in this lab session.

Figure 1: *MNIST examples.*

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image of size 28×28. You can access the MNIST data directly from TensorFlow, just import and load the data :

```
>>> mnist = keras.datasets.mnist
>>> (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Each image is mapped to a single label. Since the class names are not included with the dataset, store them here to use later when plotting the images :

```
>>> class_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255 :

```
>>> plt.figure()
>>> plt.imshow(train_images[0])
>>> plt.colorbar()
>>> plt.grid(False)
>>> plt.show()
```

We scale these values to a range of 0 to 1 before feeding to the neural network model. For this, cast the datatype of the image components from an integer to a float, and divide by 255. It's important that the training set and the testing set are preprocessed in the same way :

```
>>> train_images = train_images / 255.0
>>> test_images = test_images / 255.0
```

Display the first 25 images from the training set and display the class name below each image. Verify that the data is in the correct format and we're ready to build and train the network.

```
>>> plt.figure(figsize=(10,10))
>>> for i in range(25):
>>>  plt.subplot(5,5,i+1)
>>>  plt.xticks([])
>>>  plt.yticks([])
>>>  plt.grid(False)
>>>  plt.imshow(train_images[i], cmap=plt.cm.binary)
>>>  plt.xlabel(class_names[train_labels[i]])
>>> plt.show()
```

4 Design and test your Fully Connected Neural Network

Building the neural network requires configuring the layers of the model (as shown in Fig. 2), then compiling the model.

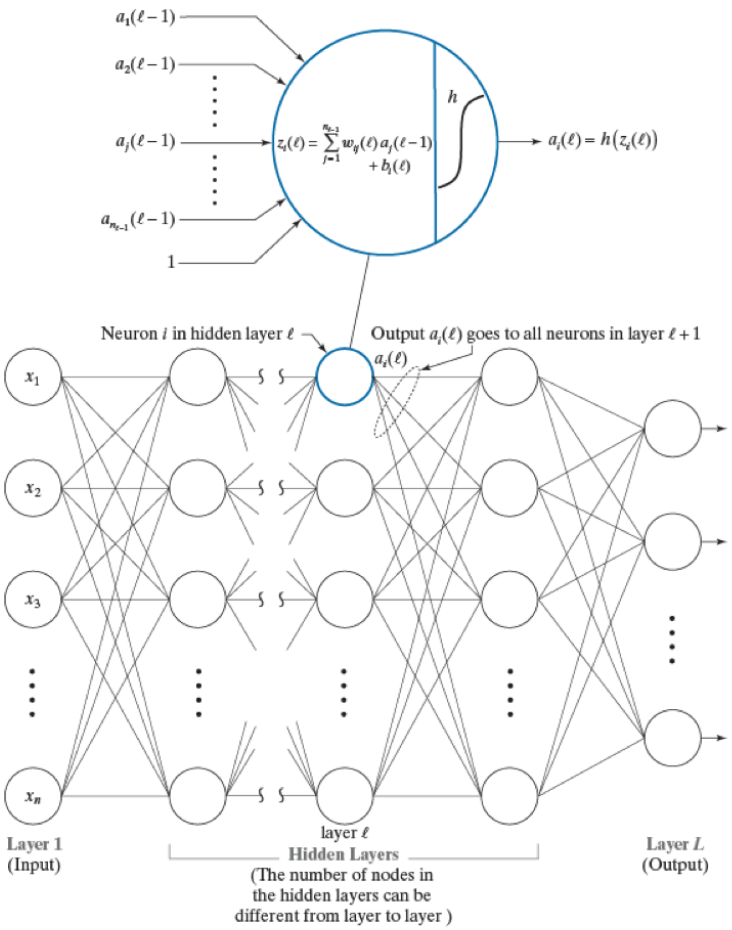

Figure 2: *Example of FCN.*

The basic building block of a neural network is the layer. Layers extract representations from the data fed into them. And, hopefully, these representations are more meaningful for the problem at hand.
Most of deep learning consists of chaining together simple layers. Most layers, like tf.keras.layers.Dense, have parameters that are learned during training.

```
>>> model = keras.Sequential([
keras.layers.Flatten(input_shape=(28, 28)),
keras.layers.Dense(128, activation=tf.nn.relu),
keras.layers.Dense(10, activation=tf.nn.softmax) ])
```

The first layer in this network, tf.keras.layers.Flatten, transforms the format of the images from a 2d-array (of 28 by 28 pixels), to a 1d-array of 28 * 28 = 784 pixels. Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn ; it only reformats the data.
After the pixels are flattened, the network consists of a sequence of two tf.keras.layers.Dense layers. These are densely-connected, or fully-connected, neural layers. The first Dense layer has 128 nodes (or neurons). The second (and last) layer is a 10-node softmax layer—this

returns an array of 10 probability scores that sum to 1. Each node contains a score that indicates the probability that the current image belongs to one of the 10 classes.
Before the model is ready for training, it needs a few more settings. These are added during the model's compile step :

- — Loss function —This measures how accurate the model is during training. We want to minimize this function to "steer" the model in the right direction.
- — Optimizer —This is how the model is updated based on the data it sees and its loss function.
- — Metrics —Used to monitor the training and testing steps. The following example uses accuracy, the fraction of the images that are correctly classified.

```
>>> model.compile(optimizer=tf.train.AdamOptimizer(),
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Training the neural network model requires the following steps :

1. Feed the training data to the model—in this example, the train_images and train_labels arrays.
2. The model learns to associate images and labels.
3. We ask the model to make predictions about a test set—in this example, the test_images array. We verify that the predictions match the labels from the test_labels array.

To start training, call the model.fit method—the model is "fit" to the training data :

```
>>> model.fit(train_images, train_labels, epochs=5)
```

Next, compare how the model performs on the test dataset :

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print('Test accuracy:', test_acc)
```

This gap between training accuracy and test accuracy is an example of overfitting. Overfitting is when a machine learning model performs worse on new data than on their training data.
With the model trained, we can use it to make predictions about some images.

```
>>> predictions = model.predict(test_images)
```

A prediction is an array of 10 numbers. These describe the "confidence" of the model that the image corresponds to each of the 10 different characters. We can see which label has the highest confidence value.We can graph this to look at the full set of 10 channels.

```
>>> def plot_image(i, predictions_array, true_label, img):
predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
plt.grid(False)
plt.xticks([])
plt.yticks([])

plt.imshow(img, cmap=plt.cm.binary)
```

```
predicted_label = np.argmax(predictions_array)
if predicted_label == true_label:
color = 'blue'
else:
color = 'red'
plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
100*np.max(predictions_array),
class_names[true_label]),
color=color)
>>> def plot_value_array(i, predictions_array, true_label):
predictions_array, true_label = predictions_array[i], true_label[i]
plt.grid(False)
plt.xticks([])
plt.yticks([])
thisplot = plt.bar(range(10), predictions_array, color="#777777")
plt.ylim([0, 1])
predicted_label = np.argmax(predictions_array)
thisplot[predicted_label].set_color('red')
thisplot[true_label].set_color('blue')
```

Let's plot several images with their predictions. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percent (out of 100) for the predicted label. Note that it can be wrong even when very confident.

```
>>> # Plot the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
plt.subplot(num_rows, 2*num_cols, 2*i+1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(num_rows, 2*num_cols, 2*i+2)
plot_value_array(i, predictions, test_labels)
```

5 Design and test your Deep Neural Network

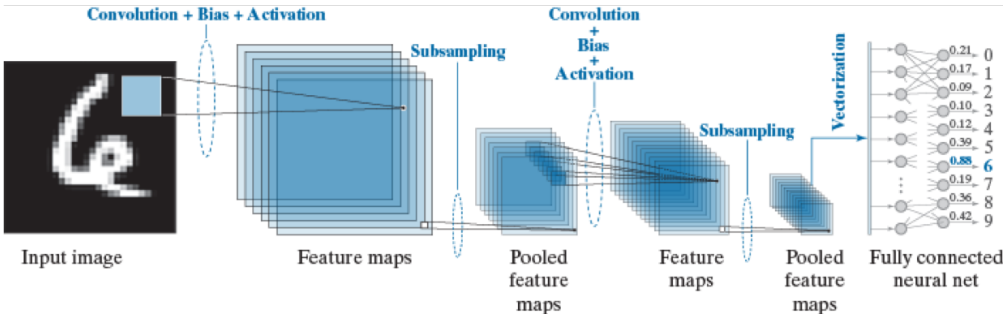

Figure 3: *LeNet2 Architecture.*

As for the FCN, building the convolutional neural network requires configuring the layers of the model (as shown in Fig. 3), then compiling the model. The complete script is available in the file "mnist_cnn.py", that is similar to the one you wrote for the FCN, except the model defined below :

```
>>> model = Sequential()
>>> model.add(Conv2D(32, kernel_size=(3, 3),
activation='relu',
input_shape=input_shape))
>>> model.add(MaxPooling2D(pool_size=(2, 2)))
>>> model.add(Conv2D(64, (3, 3), activation='relu'))
>>> model.add(Dropout(0.25))
>>> model.add(Flatten())
>>> model.add(Dense(128, activation='relu'))
>>> model.add(Dropout(0.5))
>>> model.add(Dense(num_classes, activation='softmax'))
```

Try to identify the different layers as shown in Fig. 3 and calculate the number of parameters of such a model. Try also to modify these parameters (number of epoch, number of conv/pool layers, the convolution kernel size, removing the dropout, etc) and look at the precision and prediction performances.
Compare the performances between the two models (FCN and DCNN) and try to generate overfitting by increasing the number of parameters and reducing the number of images for training.