University of Bergen

Introduction to machine learning
INF264

Project 2: Digit recognizer

Student :
Arthur Gautier

October 15, 2023

Contents

1 Dataset

We start by downloading the two provided data files. "emnist hex images.npy" contains 111,399 images, each of size 20 × 20, and the file "emnist hex labels.npy" contains the corresponding labels. We load the files as instructed in the guidelines.

2 Models

2.1 Neural network

2.1.1 Importations

In order to be able to build our model, we need to import all the required modules.

- **Numpy** to be able to compute matrix
- **Matplotlib** to print figures
- **Keras** to build our neural network

Our imports then take the form:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Dense, Flatten
from keras.models import Sequential
from keras.utils import to_categorical
from keras.datasets import mnist
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

Then, we need to split our dataset into a training and a test part in order to build the network. The train and test data contain respectively 70 and 30% of the initial data.

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, test_size=0.30, random_state=42)
```

In the code above, X is the image and y the label of the image.

2.1.2 Data pre-processing

All the labels are represented as a single number or it's not what is expected for a Neural Network : we need to turn each label into one-hot representation to be able to train our model. To do that we'll use *to_categorial()* function from Keras module. This function is going to create an 1D array fill with 0 with the number of label length except at the index of the label. For example :

```
to_categorical(3, num_classes=17)
```

will create an array of length 17 fill with 0 except on index 3. This representation is called the one-hot encoding. Thus, we have to do this on every label of our train and test data.

```
# Convert y_train into one-hot format
temp = []
for i in range(len(y_train)):
    temp.append(to_categorical(y_train[i], num_classes=17))
y_train = np.array(temp)
# Convert y_test into one-hot format
temp = []
for i in range(len(y_test)):
    temp.append(to_categorical(y_test[i], num_classes=17))
y_test = np.array(temp)
```

2.1.3 Building the model

Now we can start to create the Neural Network with Keras. We start by initialize a sequential model. It is probably the easiest way to build a model in Keras as it enables building a model layer by layer.

Then, we add a dense layer with 10 neurons with sigmoid ativation function. Defined as below :

$$\forall x \in \mathbb{R}, f(x) = \frac{1}{1 + e^{-x}}$$

We use this function because it's derivable and its codomain is $[0, 1]$ which provide values similar to probability.

To finish, we add another layer which acts as our output. We use the Softmax activation function here. Softmax makes the output sum up to 1 so the output can be interpreted as probabilities. We use 10 neurons in this last layer because our classification task have 10 classes.

```
# Create simple Neural Network model
model = Sequential()
# Dense layer with 10 neurons and an activation function (sigmoid)
model.add(Dense(10, activation='sigmoid'))
# Output
model.add(Dense(17, activation='softmax'))
```

We can now observe the architecture we created with the *model.summary()* function.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 10)                4010

 dense_1 (Dense)             (None, 17)                187

=================================================================
Total params: 4,197
Trainable params: 4,197
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: Model summary

2.1.4 Compiling the model

Now we just need to compile our model :

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

The loss function used here is *categorical_ crossentropy* it's one of the best in multiclass classification problem. Same thing with Adam for the optimizer ; the optimizer controls the learning rate and adjusts it throughout training. Lastly, we have accuracy to be passed in mertrics argument in order to measure the performance of our classifier. Now that our model is built, we can now train it.

2.1.5 Training the model

```
model.fit(X_train, y_train, epochs=5, batch_size=16,
      validation_data=(X_test,y_test))
```

```
Epoch 1/50
2359/2359 [==============================] - 5s 2ms/step - loss: 1.6782 - acc: 0.6003 - val_loss: 1.0318 - val_acc: 0.7581
Epoch 2/50
2359/2359 [==============================] - 3s 1ms/step - loss: 0.8388 - acc: 0.7912 - val_loss: 0.7228 - val_acc: 0.8076
Epoch 3/50
2359/2359 [==============================] - 3s 1ms/step - loss: 0.6589 - acc: 0.8197 - val_loss: 0.6258 - val_acc: 0.8230
Epoch 4/50
2359/2359 [==============================] - 4s 2ms/step - loss: 0.5890 - acc: 0.8342 - val_loss: 0.5795 - val_acc: 0.8325
Epoch 5/50
2359/2359 [==============================] - 4s 2ms/step - loss: 0.5528 - acc: 0.8419 - val_loss: 0.5543 - val_acc: 0.8392
```

Figure 2: Training

As we can see, after each epoch the model gets more and more precise. After 5 epochs with a final result of 84% of accuracy on the training dataset (The epoch is a hyperparameter of gradient descent that controls the number of complete passes through the training dataset.). We can also play with the batch size in our parameters : it's a hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters are updated. Let's draw the accuracy and loss curves during the training

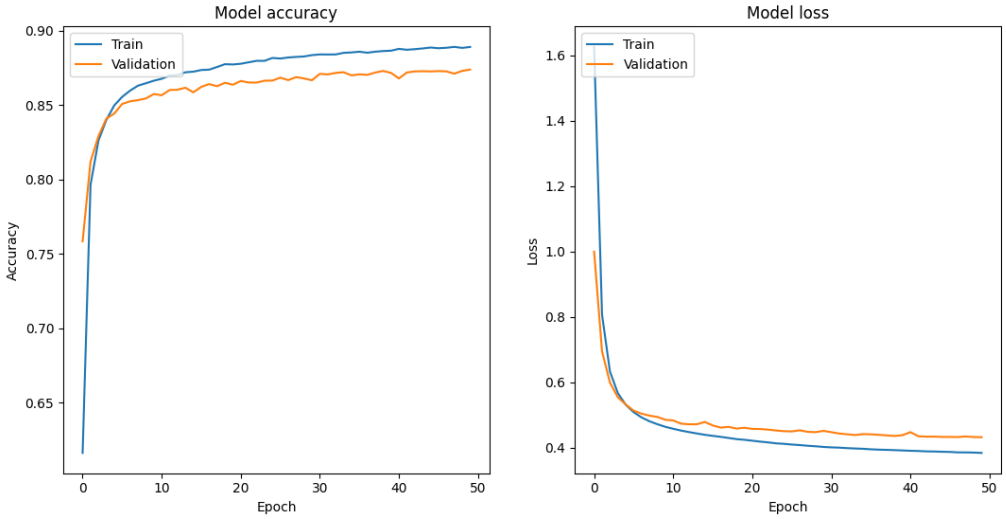

Figure 3: Accuracy and Loss curves for 50 epochs

2.1.6 Using our model to make predictions

To have a deeper view of the quality of our model we can import the module *confusion_matrix* from *sklearn.metrics* and use the function as follow on the test dataset:

```
predictions = model.predict(X_test)
predictions = np.argmax(predictions, axis=1)

# Obtain the confusion matrix
cm = confusion_matrix(y_test.argmax(axis=1), predictions)
print(cm)
```

We get this matrix as a result :

```
[[1982    0    2    1    9    5   19    4    7    3   16   14   49  106    0    1    0]
 [   2 2159   11   15   13   10    7    3   16    6    3    0    6    0    0   25    2]
 [  14   18 2007   14    9    2   17   12   17    0   23    7   12   27    6   15    1]
 [   4    9   44 1939    1   74    2   28   20   14    8   54    2   17    0    1    0]
 [   6    9    6    1 1996    1   27    0   11   81   75    0   13    1    0   34    2]
 [   8    2   16   80    5 1850   31    6   19    7   11   15   73    7   34   47    2]
 [   5    1    5    1    3    6 2095    0    2    0    9    2   37    4   10    3    0]
 [   2   14    7   10    7    5    0 2030    4   91    4    0    0    2    0   29    3]
 [   3   34   22   28   18   44   15   12 1711   31   11  138   24    1   25   35    0]
 [   3   10    0   32   81    4    0   48   13 1958    6    0    1    0    0   31    1]
 [   2    9   20    1   76    4   10    1    9   16 1574   14    1   13    4   21    0]
 [  41    0   16   60    2    7   19    0  103    3   44  787   18   29   25    7    0]
 [  49    2    4    1   14   30   38    3    9    5    6   15 1705    1   12    9    0]
 [ 194    6   33   26    1    5   25    2    4    2   19   25   11  999    1    6    0]
 [   2    0    6    0    1   28   39    0   14    0    2   19   41    2  237   10    0]
 [   4   51   44    1   41   41    6    8   17   37   37    0   19    6    6 1600    4]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    1 1698]]
```

Figure 4: Confusion matrix

We reach 88% of accuracy. In the confusion matrix, each row represents the true class of the data samples, while each column represents the predicted class by the model. The diagonal elements of the matrix correspond to the correctly predicted samples, showing high values in this case due to the strong diagonal pattern. This indicates that the model's predictions align closely with the actual ground truth, demonstrating good accuracy across different classes.

2.2 Deep learning

Now that we understand how a neural network works and how to build it, let's look at deep learning networks. To experiment deep learning we are going to build a convolutions neural network (CNN) in order to classify images.

2.2.1 Importations

As previously, in order to be able to build our model, we need to import all the required modules.

- **Numpy** to be able to compute matrix
- **Matplotlib** to print figures
- **Keras** to build our neural network
- **Sklearn.metrics** to have the confusion matrix of our training model

We will use the same dataset as before and thus apply the same data processing.

2.2.2 Building our model

We will use again the Sequential model type. Our first two layers are convolutional layers. They are going to deal with our input images seens as 2-dimensional matrices. The 64 and 32 refers to the number cof nodes in each layer. The Karnel size refers to the dimension of

our convolution matrix for the filtering. So here convolution matrix is dimension 3. The activation function used in these two first layers is the ReLU function, defined as below :

$$\forall x \in \mathbb{R}, f(x) = max(0, x)$$

In order to connect our convolution layers to our dense, output layer, we need to add a flatten layer between these two. The final dense layer needs 17 nodes as before, one for each possible outcome - we continue to use the softmax function.

```
#create model
model = Sequential()
#add model layers
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(20, 20, 1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(17, activation='softmax'))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 18, 18, 64)        640

 conv2d_1 (Conv2D)           (None, 16, 16, 32)        18464

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 17)                139281

=================================================================
Total params: 158,385
Trainable params: 158,385
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: Model summary

2.2.3 Compiling the model

Now we just need to compile our model. We use the same parameters as before. Check the previous section for more explanations.

```
#Compile model using accuracy to measure model performance
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

2.2.4 Training the model

We have now our model ready to be trained. We processed as before with 3 epochs and a batch size of 16.

```
#Train the model
history = model.fit(X_train, y_train, validation_data=(X_test, y_test),
                    epochs=3,
                    batch_size=16)
```

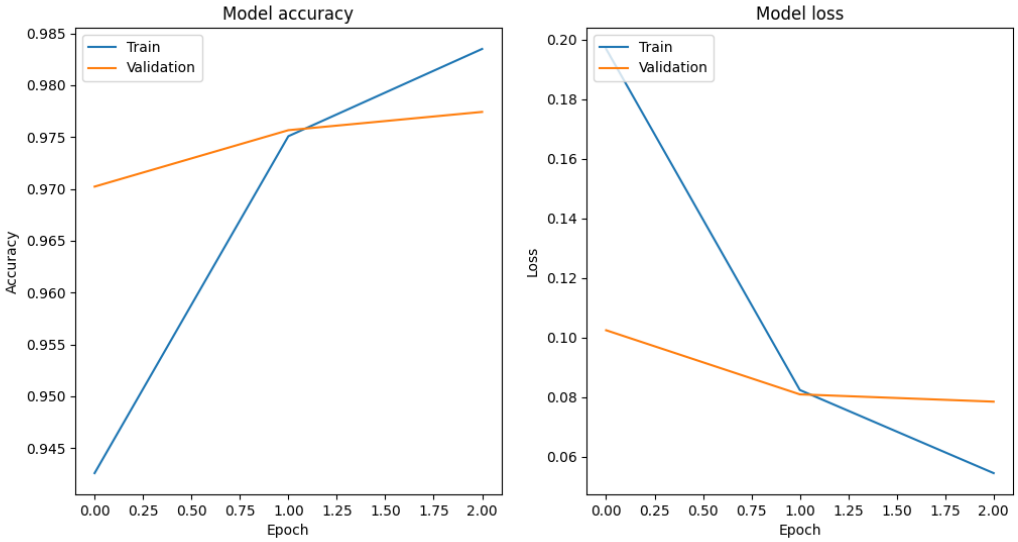

Figure 6: Accuracy and loss curves for 3 epochs

As we can see we reach a good accuracy with only 3 epochs, an accuracy around 98%. I also tried to train the model with a more epochs, we reach quickly the over-fitting with this model : the loss curve based on validation set increase after each epochs.

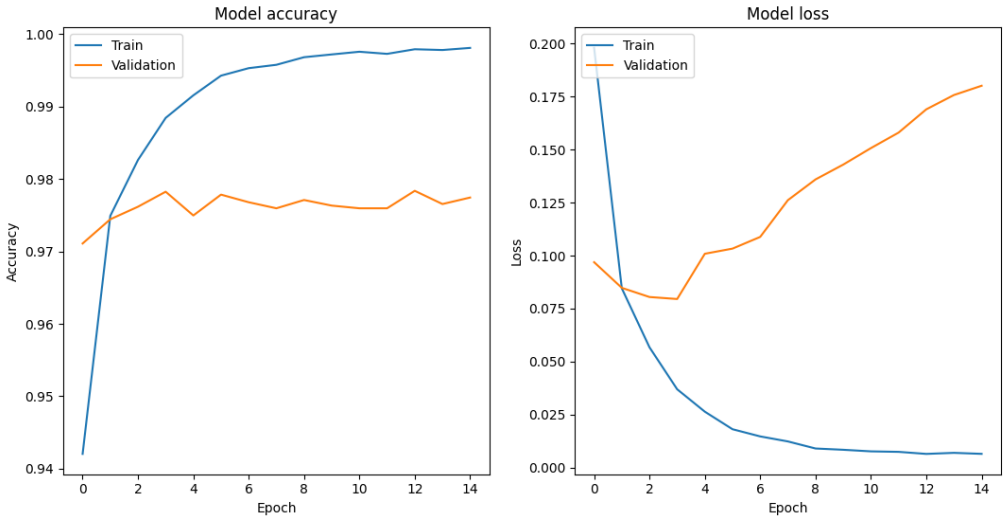

Figure 7: Accuracy and loss curves for 15 epochs

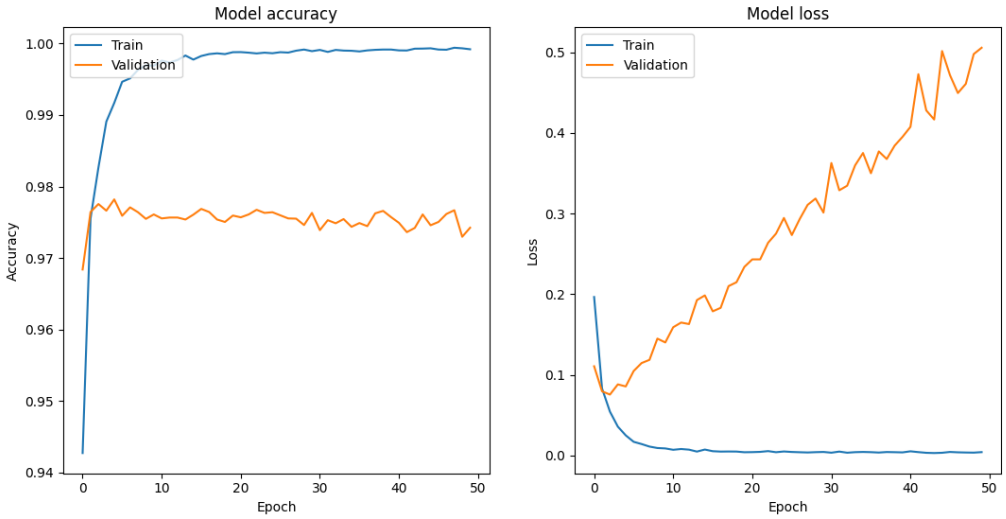

Figure 8: Accuracy and loss curves for 50 epochs

2.2.5 Using our model to make predictions

We can now look at the confusion matrix below, based on the test set.

```
[[2107    0    0    0    0    2    6    0    2    4    0    5   14   78    0    0    0]
 [   0 2258    3    2    1    0    1    3    0    1    0    1    4    0    0    3    1]
 [   1    2 2170    2    2    1    0    2    3    3    0    2    2    9    0    2    0]
 [   1    0    7 2190    0    2    0    2    1    2    0   10    0    2    0    0    0]
 [   1    1    2    2 2191    0    2    1    3   24   14    0    3    1    0   18    0]
 [   0    1    2   17    0 2157    7    1    4    6    0    5    3    0    4    6    0]
 [   2    1    0    0    2    3 2158    0    1    0    0    1   13    0    2    0    0]
 [   1    1   15   10    5    1    0 2146    2   15    1    1    0    4    0    6    0]
 [   3    5    0    4    4    6    4    0 2065    8    2   30    2    0    6   13    0]
 [   2    0    0    7    7    1    0    4    3 2160    0    2    0    1    0    1    0]
 [   1    1    1    1    6    0    0    0    0    3 1746    4    0    2    0   10    0]
 [   1    1    0   11    0    0    3    0   10    2    3 1125    0    4    1    0    0]
 [   5    2    0    0    2    3    3    0    0    0    0    4 1872    0    9    3    0]
 [  50    0    3    2    0    1    2    0    0    0    3    8    0 1288    0    2    0]
 [   0    0    0    0    0    0    0    0    1    0    0    0    1    0  391    8    0]
 [   2    4    3    0    2    5    0    1    3    0    4    0    2    4    3 1888    1]
 [   0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0 1699]]
```

Figure 9: Confusion matrix

We reach 98% of accuracy on the test dataset. It can therefore be seen that deep networks are much more efficient than simple networks. They require less epochs and give better results. They are nevertheless more complex in their architecture and integrate many more parameters than conventional neural networks.

2.3 Support Vector Machine

2.3.1 Importations

- **Numpy** to be able to compute matrix
- **Sklearn** in order to get the useful tools for machine learning and with specifics modules of sklearn, to train and evaluate the model.

We will use the same dataset as before and thus apply the same data processing. The only difference is that we limit the amount of data to 10000 samples for X and y : otherwise the model will be too long to compute.

2.3.2 Building our model

We start by defining the parameters of the grid for *GridSearchCV*. *GridSearchCV* will perform a search for the best hyper-parameters for our model using cross-validation.

```
1  # Defining the parameters grid for GridSearchCV
2  param_grid = {'C': [0.1, 1, 10, 100],
3                'gamma': [0.0001, 0.001, 0.1, 1],
4                'kernel': ['rbf', 'poly']}
```

We define this dictionary with various hyperparametrs for the SVM classifier.

- **C** penalty parameter :

- **gamma** kernel coefficient :
- **kernel** kernel type :

We then create a SVM classifier using the *svm.SVC* class from *sklearn*.

```
svc = svm.SVC(probability=True)
```

the parameters *probability* allow the model to provide probability estimates.
Now we can use *GridSearchCV* as mentioned previously. We provide it the SVC models and the parameters that we defined before.

2.3.3 Training the model

We train the model using the training data (X_train and y_train) by calling the fit method on the model.

```
model.fit(X_train,y_train)
```

2.3.4 Using our model to make predictions

We can now look at the confusion matrix below, based on the test set.

```
model with 10000 images in the dataset

The model is 91.76666666666667% accurate
[[188   0   0   1   0   0   3   0   0   0   2   0   1   8   0   0   0]
 [  0 205   0   1   0   0   0   0   0   0   0   0   0   0   0   0   1]
 [  2   3 194   1   0   0   0   3   0   0   1   0   1   4   0   0   0]
 [  0   0   1 181   0   4   0   2   3   1   0   2   0   0   0   0   0]
 [  0   0   7   0 192   0   0   0   0   3   4   0   1   0   0   4   0]
 [  1   0   1   6   0 173   0   0   1   0   0   0   3   0   0   3   1]
 [  1   1   0   0   0   0 200   0   0   0   0   2   2   0   0   2   0]
 [  1   0   3   0   0   0   0 224   0   5   1   0   0   0   0   0   0]
 [  0   0   0   8   3   3   1   1 171   2   1   8   0   0   1   6   0]
 [  0   1   0   1   2   1   0   5   0 206   0   0   0   0   0   0   0]
 [  0   1   1   0   2   1   0   0   0   2 162   0   0   1   0   2   0]
 [  4   0   2   6   2   1   1   0   5   0   1  74   2   4   1   0   0]
 [  2   1   0   0   1   3   1   0   1   0   0   0 166   0   0   1   0]
 [ 11   1   0   2   0   0   6   0   0   0   0   2   0  96   0   0   0]
 [  0   0   0   0   1   1   2   0   1   0   2   1   7   0  28   1   0]
 [  0   3   2   1   3   1   0   0   2   1   2   0   3   0   0 145   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 148]]
```

Figure 10: Confusion matrix

As we can see, the results are great with a strong diagonal in our confusion matrix. We reached 92% of accuracy with only 10000 images of the imported data. I also tried to

run the model with only 100 images, we reach 45% of accuracy. If the SVM defends itself well in terms of accuracy, it is very slow to compile compared to classic neural network and CNN.

3 Conclusion

While classical neural networks are less suited for image recognition, CNNs are specifically designed for this purpose, capable of extracting complex features from images. We can see that with the high accuracy reached in our tests. The amount of data available plays a significant role, with classical neural networks needing substantial amounts, while CNNs and SVMs can be effective with smaller datasets. Training times vary, with classical neural networks often taking longer, while CNNs are faster. SVMs have longer training times with high numbers of images in the dataset. In summary, the choice depends on the specific task's complexity, data volume, time constraints and desired performance, with CNNs typically preferred for image recognition due to their specialized architecture and outstanding results. I would say that SVMs are valuable for simpler tasks. For all those reasons, I'd choose a CNN model for the Santa's Workshop.