Digit Reco MNINST DATASET

November 14, 2020

1 Subject

The dataset for the project is distributed via private Kaggle InClass competition. You can access this via this link: https://www.kaggle.com/account/login?ReturnUrl=%2Ft%2F8e01943b0f17473098fe3b2409a02c65

The deadline is 15/11/2020 @ midnight CET.

The outcome of your work should be a report (at least 3 pages long, not counting images and code). It should reflect your work on the dataset, motivation behind method choices, and analysis of model performance. It is encouraged to present not only successful, but also failed solutions, if they are supported by analysis and prove to be helpful in the search of the final solution.

The report should be in PDF form (which can be generated from a Jupyter notebook)

2 Objectivs

Kaggle says : "In this competition you are challenged with a digits classification task. The images are similar to MNIST, however they contain the background noise."

So the goal is to build a classification model to classify the provided dataset that is a noisy version of MNIST

3 Getting the data

My idea is to use GoogleColab to be able to use its GPU if needed. To be able to work with GoogleColab I have zipped that I get from Kaggle and I have put them in my personal GoogleDrive. Since GoogleColab do not provide a permanent storage, this is a way to be able to get the dataset each time I run GoogleColab.

Connect to my personal Google Drive

```
1 Physical GPUs, 1 Logical GPUs
```

Load datas into numpy table structure - images for the train data - images_test for the test data

4 Looking at the dataset

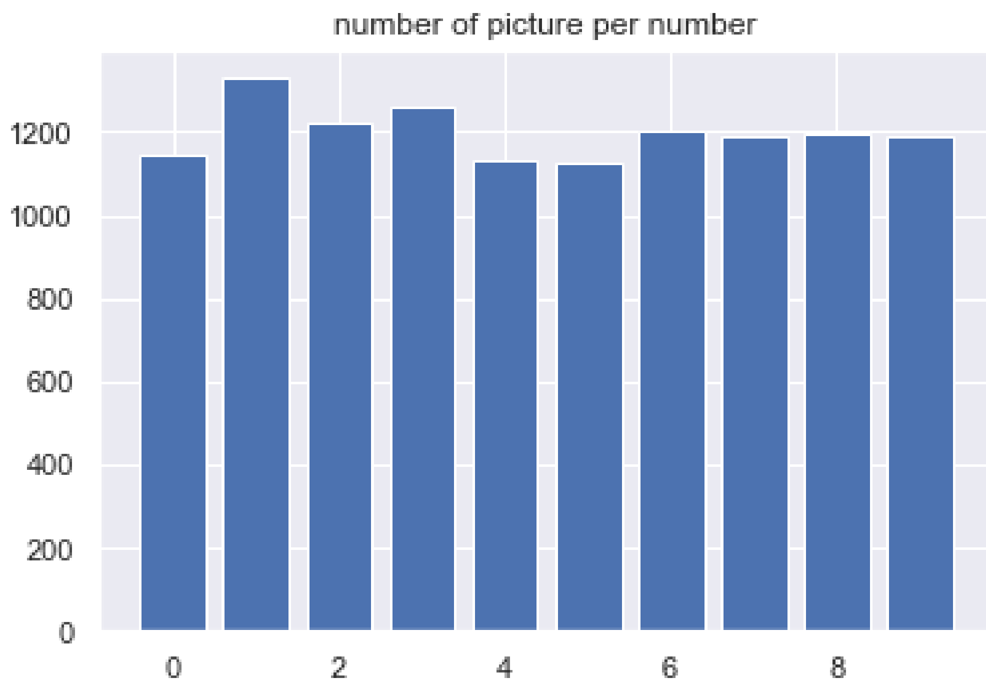
Train dataset

```
(12000, 785)
```

[7]:
```
array([[0.71393964, 0.95016594, 0.08249079, …, 0.55627015, 0.69829481,
        5.        ],
       [0.20744189, 0.15994205, 0.05264073, …, 0.20319134, 0.39100319,
        7.        ]])
```

- the train dataset has 12000 rows and 785 columns
- each row can be splitted into : the first 784 column which contain the picture information and the last row which contains the expected value of the number to be classified in the image So I split the train dataset into two parts :
- label : with only the labels
- images : with only the images

The dataset is well balanced



Test dataset

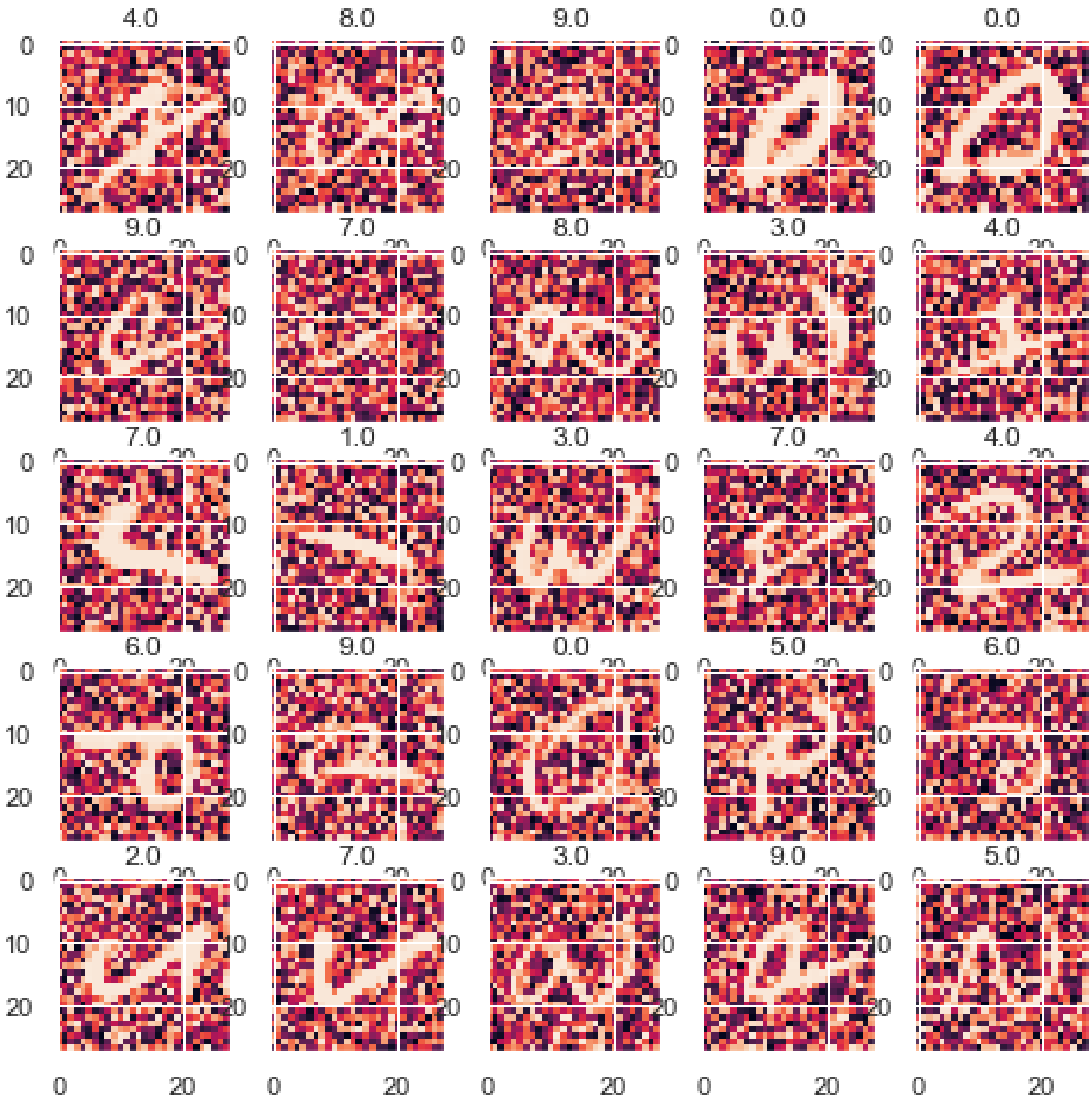
```
(50000, 784)
```

[156]:
```
array([[0.923635  , 0.84568737, 0.39288922, …, 0.91443564, 0.20895224,
        0.81808189],
       [0.00776682, 0.67074047, 0.95102143, …, 0.68060272, 0.67633705,
        0.95092055]])
```

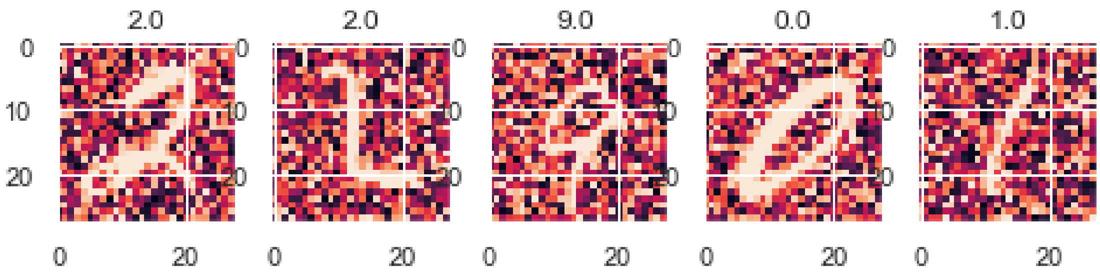
The test dataset has 50000 rows. Each rows has 784 column for the image data.

4.1 Looking at the pictures

The 784 columns are in fact an array of 28x28 pixels. We only have one array so this means we have black and white pictures with different grey levels.
I am printing 25 random pictures from the train dataset with their labels to have an idea about the pictures and the noise



We can see that : - the level of noise is quit high and some figure are not easy to see - the images are rotated and flipped.
I use a transpose on the matrix to show the images on the usual way :



5 Using SVM for classification

My first idea is to use SVM for the classification. Even if I think neural network should be able to do better it would be a good start to have a first quick reference.
Since the number of features is quite high, I choose a mix between PCA (to reduce the number of features) and SVM for classification.

And I have defined a grid search strategy to try to find best parameters for both PCA and SVM.

I am using for this grid search the whole train dataset, since this gridsearch function is doing cross validation folder by itself with this dataset.

Preparing the data for the PCA/SVM - x_grid used for the grid search - x_train to train the model when the hyperparameters are choosen - x_valid to predict and evaluate the accuracy of the model
The splitting is done in order to keep the same balance as the initial balance

```
{'svc__C': 100, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

So the best found parameters are : {svc___C=100, svc___gamma=0.01, svc___kernel=rbf}
I re-run the grid search to be a bit more precise and I add also the number of component of the PCA as a parameter to see the impact.

```
{'pca__n_components': 50, 'svc__C': 100, 'svc__gamma': 0.05}
```

After this second steps the best parameters are : {'pca___n_components': 50, 'svc___C': 100, 'svc___gamma': 0.05}

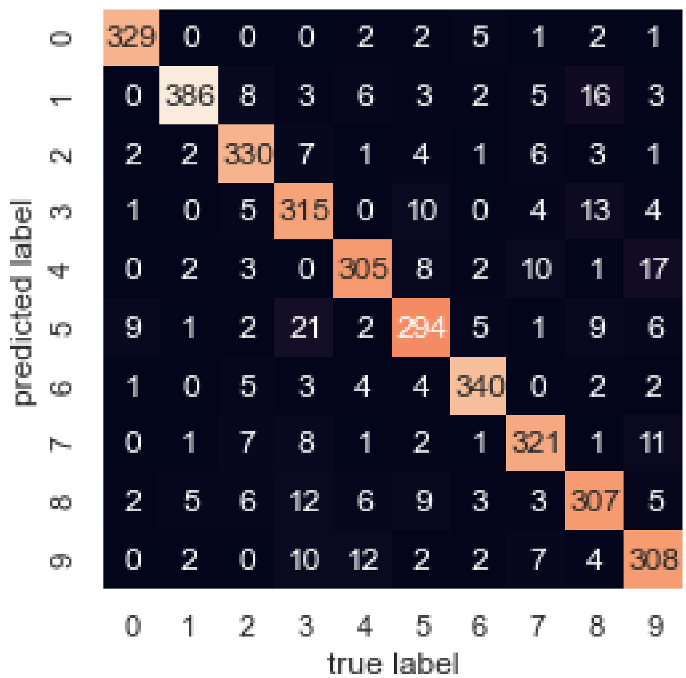
```
[125]: Pipeline(steps=[('pca',
                  PCA(n_components=50, svd_solver='randomized', whiten=True)),
                 ('svc', SVC(C=100, class_weight='balanced', gamma=0.05))])
```

I run the PCA/SVM on the training data and predict on the validation dataset in order to have an estimation of the classification accuracy.

In conclusion, with PCA/SVM the accuracy on the validation dataset is : 90%

```
              precision    recall  f1-score   support

           0       0.96      0.96      0.96       344
```

```
           1       0.89      0.97      0.93       399
           2       0.92      0.90      0.91       366
           3       0.89      0.83      0.86       379
           4       0.88      0.90      0.89       339
           5       0.84      0.87      0.85       338
           6       0.94      0.94      0.94       361
           7       0.91      0.90      0.90       358
           8       0.86      0.86      0.86       358
           9       0.89      0.86      0.87       358

    accuracy                           0.90      3600
   macro avg       0.90      0.90      0.90      3600
weighted avg       0.90      0.90      0.90      3600
```



6 CNN Neural Network - simple structure

Now I will use a classical CNN neural network usually used on the MNIST dataset.

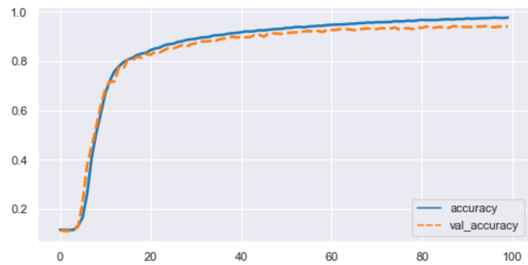
6.1 Preparing the data for the CNN

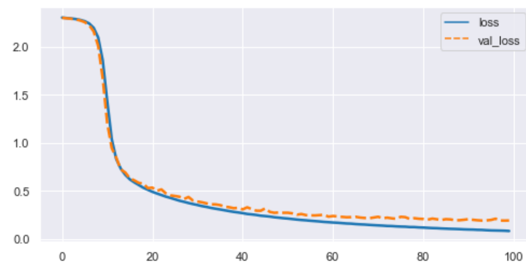
Here I reshape the datasets (x_train,x_valid,x_grid) so construct 28*28 matrix
I also transpose this matrix to make the display of the pictures more visual and add a channel dimension to fit with the CNN network

Transform numerical variables into categorical variable

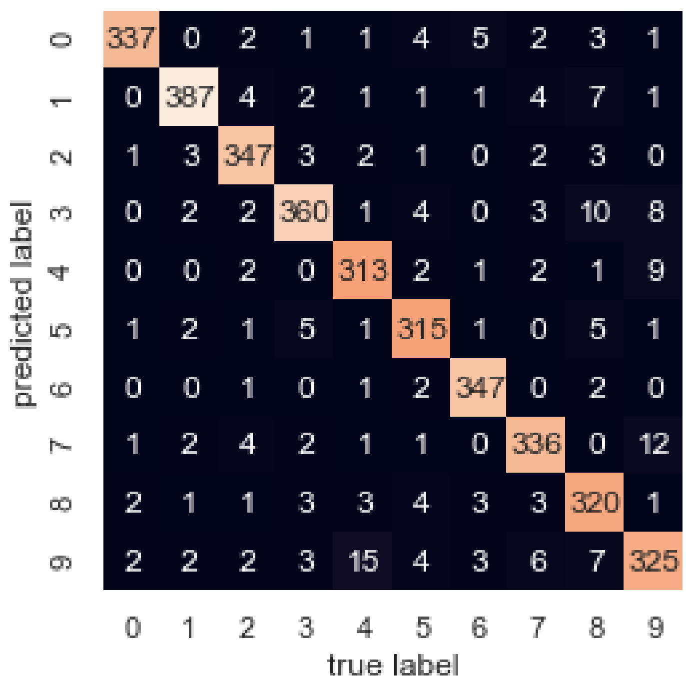
6.2 Building the CNN

```
<Figure size 432x288 with 0 Axes>
```





	precision	recall	f1-score	support
0	0.95	0.98	0.96	344
1	0.95	0.97	0.96	399
2	0.96	0.95	0.95	366
3	0.92	0.95	0.94	379
4	0.95	0.92	0.94	339
5	0.95	0.93	0.94	338
6	0.98	0.96	0.97	361
7	0.94	0.94	0.94	358
8	0.94	0.89	0.92	358
9	0.88	0.91	0.89	358
accuracy			0.94	3600
macro avg	0.94	0.94	0.94	3600
weighted avg	0.94	0.94	0.94	3600



With this "simple" CNN the result is improved to around 94% accuracy : better then the result found for SVM

Looking at the loss and accuracy curves : - after 30/40 epochs the validation loss start increasing wherease the training loss decreases - the final result in terme of accuracy is around 95% : far from what CNN can do with traditional MNIST dataset so I can hope to improve the model

So my ideas to improve this result is to tune the hyper parameters of my modele : - initialisation of the weights - optimizer - number of batch - number of epoch - kind of activation used

To do this I use the GridSearchCV function that realize both grid search on hyperparameters and cross validation with 5 folders to evaluate the best choice.

[135]:

	rank_test_score	mean_test_score	std_test_score	param_activation \
1	1	0.953000	0.008343	relu
7	2	0.952667	0.005692	relu
0	3	0.950833	0.006084	relu
2	4	0.950500	0.008368	relu
13	5	0.949333	0.004295	relu
11	6	0.949083	0.004605	relu
10	7	0.948333	0.005798	relu
8	8	0.947667	0.005837	relu
5	9	0.947417	0.003665	relu
4	10	0.946750	0.002065	relu
16	11	0.946250	0.003238	relu
3	12	0.946083	0.005908	relu

9	13	0.945667	0.005807	relu
6	14	0.944667	0.011566	relu
15	15	0.942833	0.003178	relu
17	16	0.942333	0.003551	relu
12	17	0.941750	0.003965	relu
14	18	0.941167	0.006205	relu
19	19	0.918750	0.010010	sigmoid
22	20	0.915000	0.004074	sigmoid
25	21	0.904167	0.025931	sigmoid
28	22	0.900333	0.019781	sigmoid
18	23	0.899000	0.009915	sigmoid
21	24	0.891250	0.024457	sigmoid
23	25	0.888333	0.018189	sigmoid
20	26	0.887000	0.008406	sigmoid
34	27	0.886667	0.009163	sigmoid
31	28	0.870833	0.026333	sigmoid
27	29	0.835667	0.037184	sigmoid
26	30	0.821750	0.049187	sigmoid
29	31	0.809667	0.075800	sigmoid
24	32	0.688167	0.297752	sigmoid
30	33	0.651583	0.274231	sigmoid
35	34	0.638417	0.111130	sigmoid
32	35	0.604000	0.256638	sigmoid
33	36	0.432000	0.295103	sigmoid

	param_batch_size	param_optimizer	param_init	param_epochs
1	32	Nadam	normal	10
7	64	Nadam	normal	10
0	32	Adam	normal	10
2	32	rmsprop	normal	10
13	128	Nadam	normal	10
11	64	rmsprop	uniform	10
10	64	Nadam	uniform	10
8	64	rmsprop	normal	10
5	32	rmsprop	uniform	10
4	32	Nadam	uniform	10
16	128	Nadam	uniform	10
3	32	Adam	uniform	10
9	64	Adam	uniform	10
6	64	Adam	normal	10
15	128	Adam	uniform	10
17	128	rmsprop	uniform	10
12	128	Adam	normal	10
14	128	rmsprop	normal	10
19	32	Nadam	normal	10
22	32	Nadam	uniform	10
25	64	Nadam	normal	10

28	64	Nadam	uniform	10
18	32	Adam	normal	10
21	32	Adam	uniform	10
23	32	rmsprop	uniform	10
20	32	rmsprop	normal	10
34	128	Nadam	uniform	10
31	128	Nadam	normal	10
27	64	Adam	uniform	10
26	64	rmsprop	normal	10
29	64	rmsprop	uniform	10
24	64	Adam	normal	10
30	128	Adam	normal	10
35	128	rmsprop	uniform	10
32	128	rmsprop	normal	10
33	128	Adam	uniform	10

The result of this grid search is that the best hyper parameters are : * init : normal * batch : 32 or 64 is close * optimizer : 'nadam' and 'rmsprop' are close * activation : 'relu' So I decide to run a new gridsearchCV including this time bigger epoch and to keep open the choice for optimizer and batch size.

[137]:

	rank_test_score	mean_test_score	std_test_score	param_activation \
2	1	0.961417	0.002721	relu
4	2	0.959333	0.006042	relu
0	3	0.957167	0.006069	relu
8	4	0.955250	0.004704	relu
10	5	0.954167	0.003575	relu
6	6	0.953833	0.003627	relu
7	7	0.952750	0.003860	relu
3	8	0.952750	0.006267	relu
1	9	0.951250	0.003039	relu
11	9	0.951250	0.004602	relu
9	11	0.947917	0.006374	relu
5	12	0.946667	0.001728	relu

	param_batch_size	param_optimizer	param_init	param_epochs
2	32	Nadam	normal	50
4	32	Nadam	normal	100
0	32	Nadam	normal	20
8	64	Nadam	normal	50
10	64	Nadam	normal	100
6	64	Nadam	normal	20
7	64	rmsprop	normal	20
3	32	rmsprop	normal	50
1	32	rmsprop	normal	20
11	64	rmsprop	normal	100
9	64	rmsprop	normal	50

5	32	rmsprop	normal	100

This grid search gives the better results for : * optimizer : Nadam * batch_size : 32 * epoch 100

7 Adding new layers

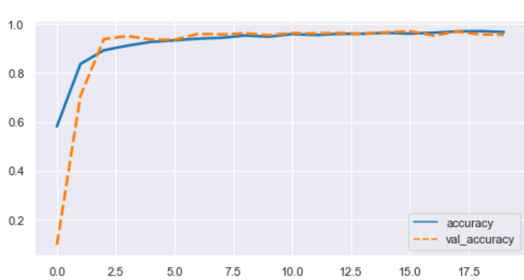
In order to try to improve the model I add more layers : * replace the convolution layer with kernel(5,5) by two layers with kernet (3,3) * replace the max pooling layer by a CNN layer with kernel(5,5) and a strides of 2 : this has mostly the same effect with the ability to have parameters * add dropout layers for a better learning * add batch normalization to try to fix the vanishing gradient problem

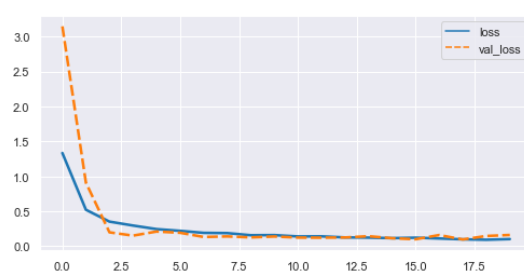
I fix the activation to 'relu' and let the inititialization to default ('glorot_uniform')

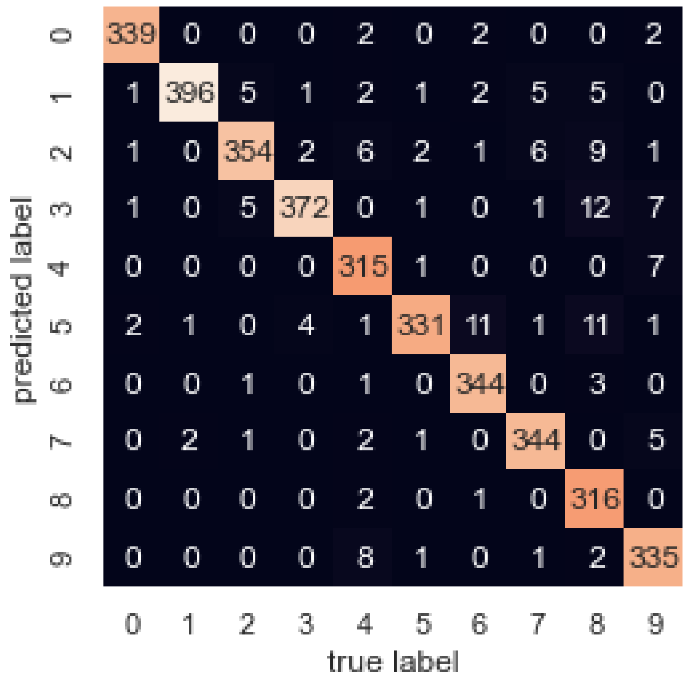
I do a first try with the hyper parameters that I have selected above and a dropout ratio of 0.5

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       344
           1       0.95      0.99      0.97       399
           2       0.93      0.97      0.95       366
           3       0.93      0.98      0.96       379
           4       0.98      0.93      0.95       339
           5       0.91      0.98      0.94       338
           6       0.99      0.95      0.97       361
           7       0.97      0.96      0.96       358
           8       0.99      0.88      0.93       358
           9       0.97      0.94      0.95       358

    accuracy                           0.96      3600
   macro avg       0.96      0.96      0.96      3600
weighted avg       0.96      0.96      0.96      3600
```

```
<Figure size 432x288 with 0 Axes>
```







The best val_accuracy is close to 97%. A clear improvment.

I do again a grid search to see if previous best parameters are still ok and to try to find best parameter for dropout rate.

For this I add call_back function : early_stop to stop search when no more improvment in order to increase the speed of the grid search

[142]:

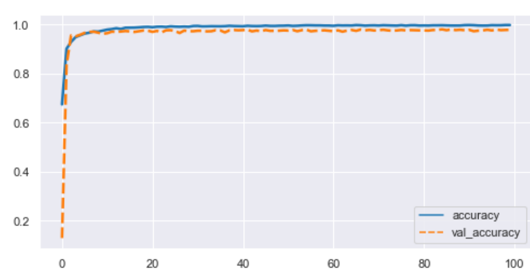
```
   rank_test_score  mean_test_score  std_test_score param_batch_size  \
3                1         0.974167        0.002541               32
9                2         0.973667        0.004883               32
2                3         0.973500        0.001818               32
1                4         0.972917        0.001826               32
7                5         0.972583        0.005918               32
5                6         0.972417        0.002392               32
6                7         0.971750        0.003903               32
0                8         0.971000        0.004351               32
4                9         0.970833        0.003819               32
8               10         0.970000        0.004617               32

  param_optimizer param_epochs param_dropout_rate
3         rmsprop          100               0.35
9         rmsprop          100                0.5
2           nadam          100               0.35
1         rmsprop          100                0.3
7         rmsprop          100               0.45
```

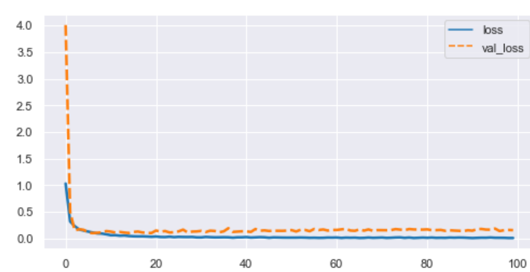
5	rmsprop	100	0.4
6	nadam	100	0.45
0	nadam	100	0.3
4	nadam	100	0.4
8	nadam	100	0.5

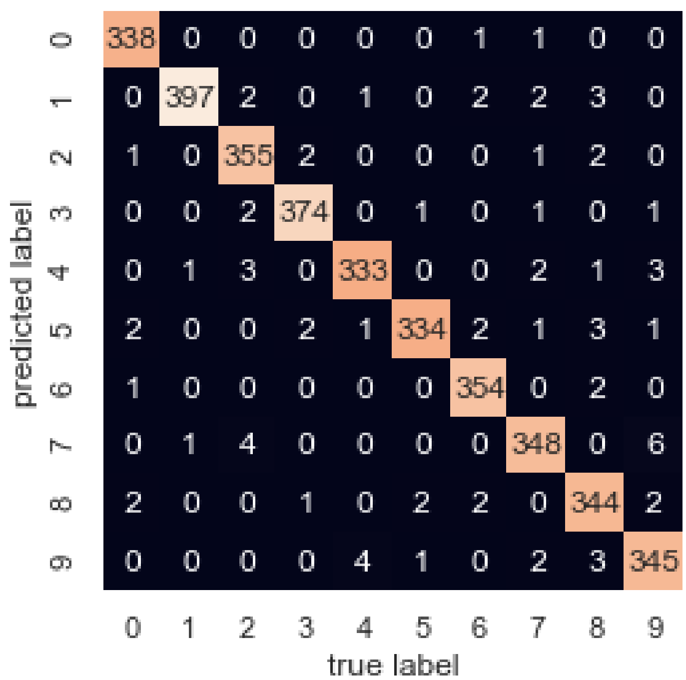
I keep the following parameters (rmsprop, drop-out=0.35) and I re-run the model. I add the callback “ModelCheckpoint” that stores the best weights in order to be able to run it again on the test dataset and submit it on Kaggle

	precision	recall	f1-score	support
0	0.99	0.98	0.99	344
1	0.98	0.99	0.99	399
2	0.98	0.97	0.98	366
3	0.99	0.99	0.99	379
4	0.97	0.98	0.98	339
5	0.97	0.99	0.98	338
6	0.99	0.98	0.99	361
7	0.97	0.97	0.97	358
8	0.97	0.96	0.97	358
9	0.97	0.96	0.97	358
accuracy			0.98	3600
macro avg	0.98	0.98	0.98	3600
weighted avg	0.98	0.98	0.98	3600

```
&lt;Figure size 432x288 with 0 Axes&gt;
```





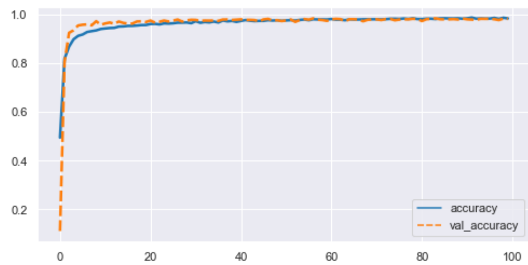


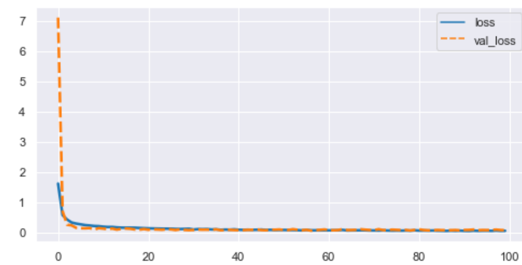
The best validation accuracy is 98,22%

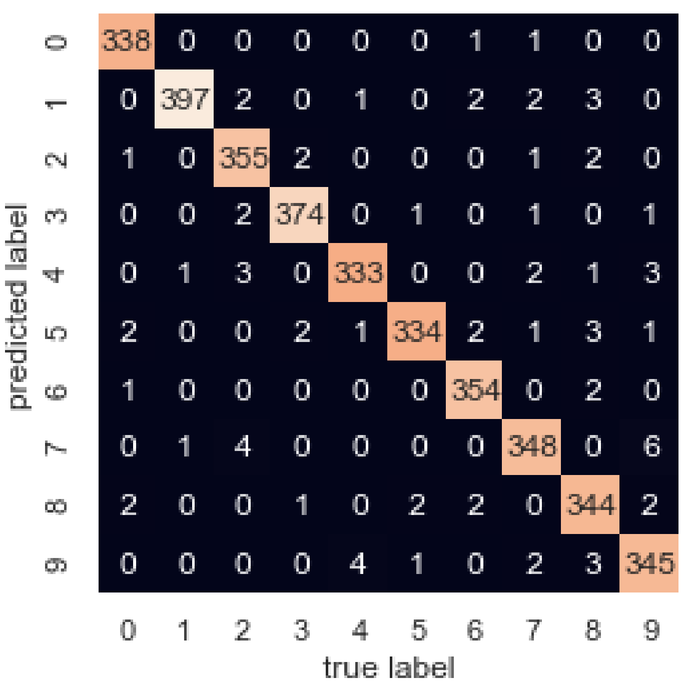
With this I obtain a score of 97.946 on Kaggle

The result on Kaggle is under what I get on training and validation dataset. I will add data augmentation to try to reduce overfitting.

```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99       344
           1       0.98      0.99      0.99       399
           2       0.98      0.97      0.98       366
           3       0.99      0.99      0.99       379
           4       0.97      0.98      0.98       339
           5       0.97      0.99      0.98       338
           6       0.99      0.98      0.99       361
           7       0.97      0.97      0.97       358
           8       0.97      0.96      0.97       358
           9       0.97      0.96      0.97       358

    accuracy                           0.98      3600
   macro avg       0.98      0.98      0.98      3600
weighted avg       0.98      0.98      0.98      3600


<Figure size 432x288 with 0 Axes>
```







With this setup I get my best result on Kaggle : 98.346

7.1 Train my best network on classical Mnist Network

Another idea is to try to do transfer learning from a pretrained model on “classical” Mnist dataset

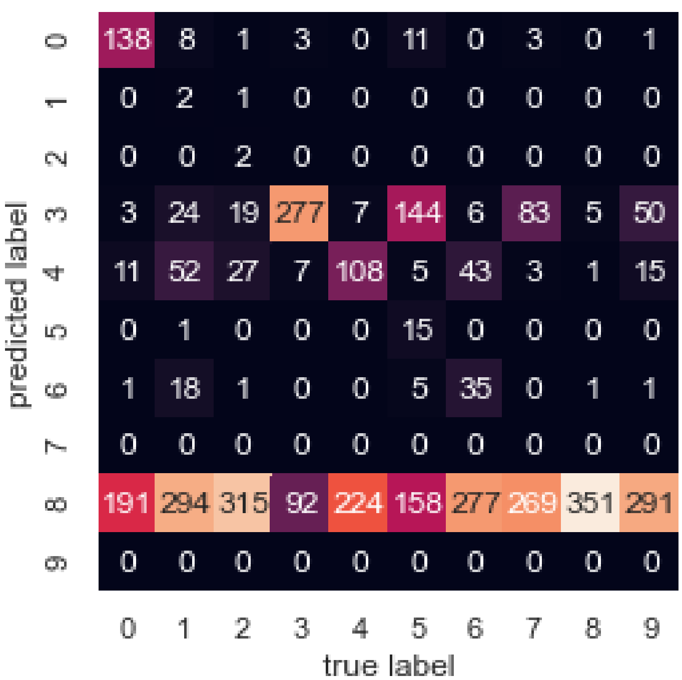
```
(60000, 28, 28, 1)
(60000,)
```

We achieve an accuracy of 99,63% and a validation accuracy of 99,63%

First test : we apply directly this model to our tain/validation data without any change.

```
C:\Users\erick\.conda\envs\digitreco\lib\site-
packages\sklearn\metrics\_classification.py:1221: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
```

```
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.84      0.40      0.54       344
           1       0.67      0.01      0.01       399
           2       1.00      0.01      0.01       366
           3       0.45      0.73      0.56       379
           4       0.40      0.32      0.35       339
           5       0.94      0.04      0.08       338
           6       0.56      0.10      0.17       361
           7       0.00      0.00      0.00       358
           8       0.14      0.98      0.25       358
           9       0.00      0.00      0.00       358

    accuracy                           0.26      3600
   macro avg       0.50      0.26      0.20      3600
weighted avg       0.50      0.26      0.20      3600
```



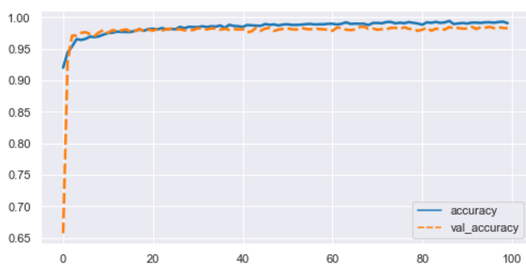
The result is very bad, so I am trying to do transfer learning from the whole model.

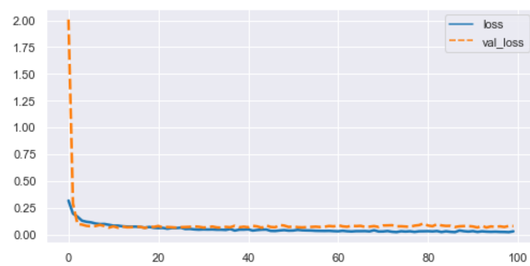
So I train the same model using the already define weights but applying my dataset pictures.

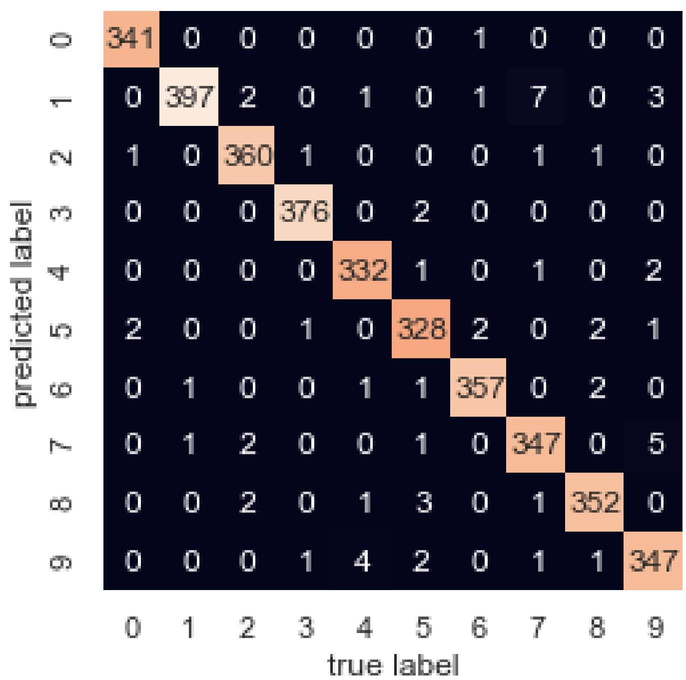
```
              precision    recall  f1-score   support
```

0	1.00	0.99	0.99	344
1	0.97	0.99	0.98	399
2	0.99	0.98	0.99	366
3	0.99	0.99	0.99	379
4	0.99	0.98	0.98	339
5	0.98	0.97	0.97	338
6	0.99	0.99	0.99	361
7	0.97	0.97	0.97	358
8	0.98	0.98	0.98	358
9	0.97	0.97	0.97	358
accuracy			0.98	3600
macro avg	0.98	0.98	0.98	3600
weighted avg	0.98	0.98	0.98	3600

```
<Figure size 432x288 with 0 Axes>
```



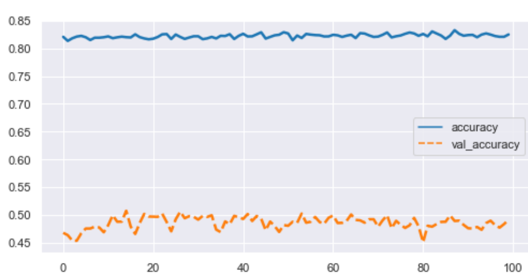


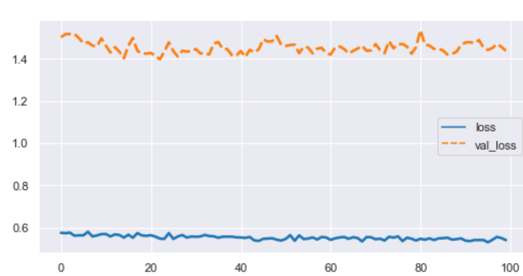


The result is good.

The best saved val_accuracy gives a result of 98.153 on Kaggle with the test dataset

Then I try to train only the last layers and to keep the CNN (feature detection part with no change from the MNist model)





The result is not good the validation accuracy do not get above 0.5. So I stop here my tests

8 Other tests

I also try other things but with no improvment of the scores.

My idea was for instance to remove the noise by another technic.

The simplest way to do it is to apply a threshold (for instance 0.95) at the picture in order to get a "Black and White" image with (0 and 1). The visual effect is qui good but the result in term of recognition is not good. I think it is because too much information is lost by this filtering.

I also try other "Callback" option available in Keras : * LearningRateScheduler and ReduceLROn-Plateau to decrease learning rate dynamically * EarlyStopping : to stop searching and consequently try to avoid overfitting

I do not find better solution when using these "Callback"

9 Conclusion

My best kaggle score : 98.346 has been reach with the following model :

```
model.add(Conv2D(32, kernel_size = 3, activation='relu', input_shape = (28, 28, 1)))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(32, kernel_size = 5, strides=2, padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(dropout_rate))

model.add(Conv2D(64, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size = 3, activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, kernel_size = 5, strides=2, padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(dropout_rate))

model.add(Conv2D(128, kernel_size = 4, activation='relu'))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dropout(dropout_rate))

model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(dropout_rate))
model.add(Dense(10, activation='softmax'))
```

and the following hyperparameters * batch size 64 * optimiser : rmsprop * droput rate 0.35

and the following data-augmentation * rotation_range=10, # randomly rotate images in the range (degrees, 0 to 180) * width_shift_range=0.1, # randomly shift images horizontally (fraction of total width) * height_shift_range=0.1, # randomly shift images vertically (fraction of total height)