Hyper parameters, Implementation and results

```
- Thomas Ehling A20432671
```

Abstract

With the library Keras and the programming language python, 2 differents Neural Network will be implemented on 2 specific datasets, and we will analyze the effect of the differents hyper parameters.

implementation details

The link to all the data files are in the “data.txt” file. You can also find these links here :

- cifar_10 : no link, loaded from keras
- crimebase.data : https://drive.google.com/file/d/1e78DPbXmsSpKUfVnJ86axQHZc_pSODO3/view?usp=sharing

All python files are named “as1_” + the name of the corresponding dataset.

FOR LOADING THE ATA :
Each program has his own arguments :

- **CIFAR10** : No arguments needed
-
- **Crime** : 1 argument : **Path to the file crimebase.data**
  If no data path given, an error will be generated !

Each program will produced a “result_”+name of the dataset+".txt".
This file is a log file with all the final metrics. It can be found complete in the folder data.

I. CIFAR10 Dataset

I.A Proposed solution

Data preparation

The 3 selected class are “Airplane”, “Automobile” and “Ship” (0,1,8), so the data can still be related in a way it would be in a real-life project.

The data set is loaded with a training and testing data-set. These data-sets are processed to keep only the selected classes.

Then the function *split-data* split the training data into a training set (4/5) and a validation set (1/5).

At this step, here is the size of the datasets :

- training : 12 000
- validation : 3 000
- testing : 3 000

Vectorizing of the images:

```
train_img = train_img.reshape(-1, NB_FEATURES)
test_img = test_img.reshape(-1, NB_FEATURES)

train_img = train_img.astype("float32") / 255
test_img = test_img.astype("float32") / 255
```

Encoding of the class labels :

```
train_labels = keras.utils.to_categorical(train_labels)
val_labels = keras.utils.to_categorical(val_labels)
test_labels = keras.utils.to_categorical(test_labels)
```

Shape at the end of the preprocessing :

- training images : (12000, 3072)
- training labels : (12000, 3)

- validation images : (3000, 3072)
- validation labels : (3000, 3)
- testing images : (3000, 3072)
- testing images : (3000, 3)

Model Architecture

Here is the model architecture :

```
model = models.Sequential()

model.add(
    layers.Dense(512, activation='relu', input_shape=(train_data.shape[1:]), kernel
# model.add(layers.Flatten())

model.add(layers.Dense(128, activation='relu', kernel_regularizer=dense_regularizer
# model.add(BatchNormalization())
model.add(Dropout(drop))

model.add(layers.Dense(32, activation='relu', kernel_regularizer=dense_regularizer)
# model.add(BatchNormalization())
model.add(Dropout(drop))

model.add(layers.Dense(16, activation='relu', kernel_regularizer=dense_regularizer)
# model.add(BatchNormalization())
model.add(Dropout(drop))

model.add(layers.Dense(NB_CLASS, activation='softmax'))

model.compile(loss=loss_fct,
              # model.compile(loss='sparse_categorical_crossentropy',
  optimizer=opt,
              metrics=[metric])
```

with these values set by default :

```
loss_fct = "categorical_crossentropy"
opt = "Adam"
batch_size = 12
epochs = 40
```

```
dense_regularizer = None
drop = 0
```

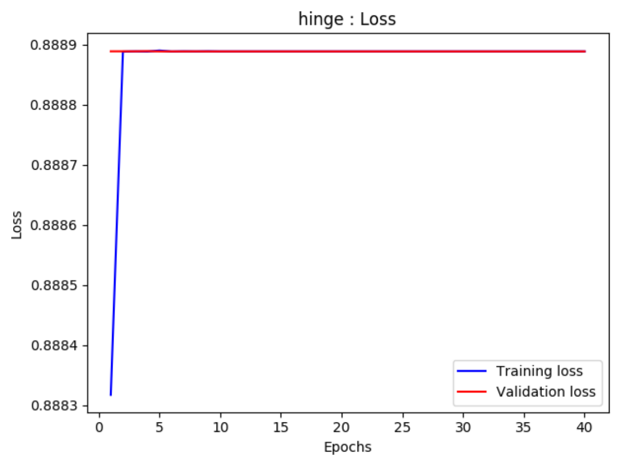
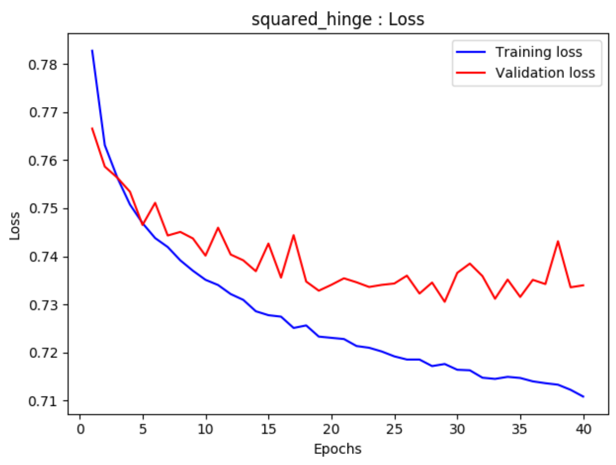
To assess the value of the final accuracy, a random function has been run and reported an accuracy of 33%.

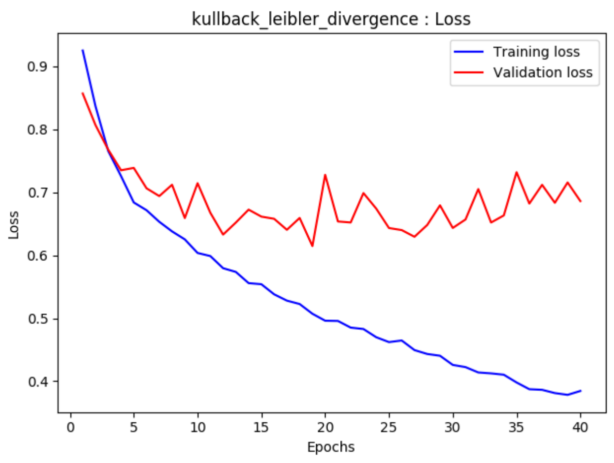
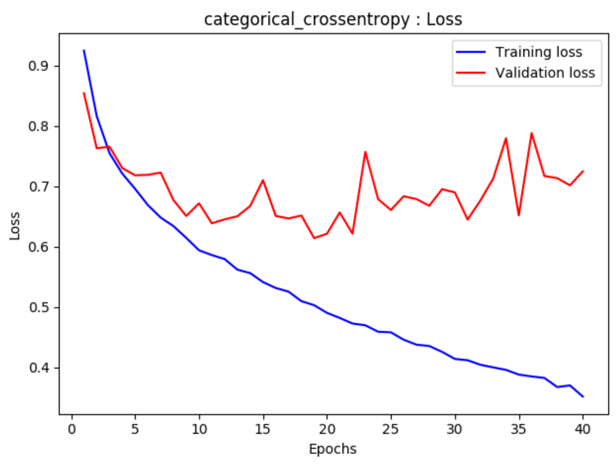
HyperParameter tuning

1. loss functions

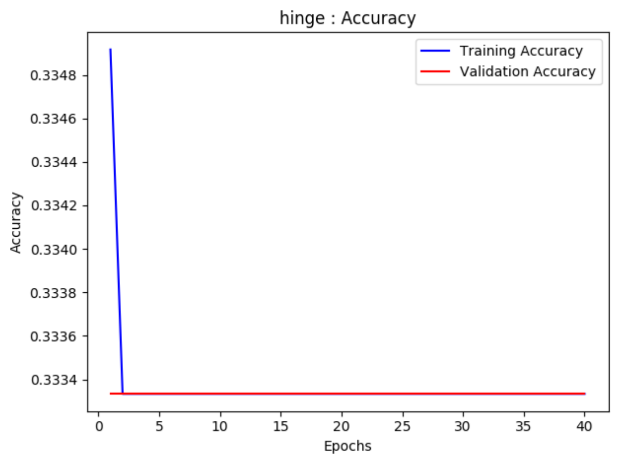
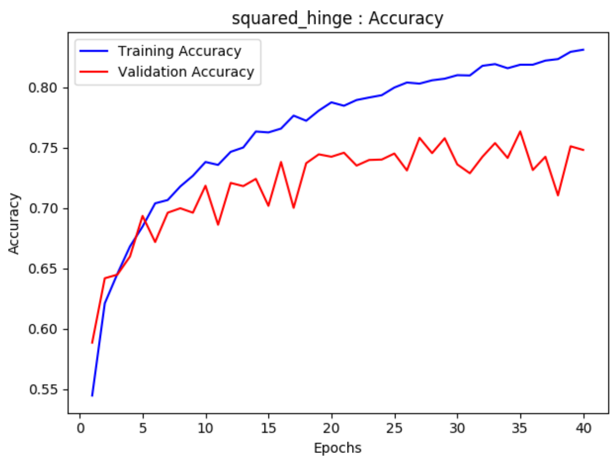
It is a model for single output regression, so the loss functions we can use are :
“hinge”
“squared_hinge”
“kullback_leibler_divergence”
“categorical_crossentropy”

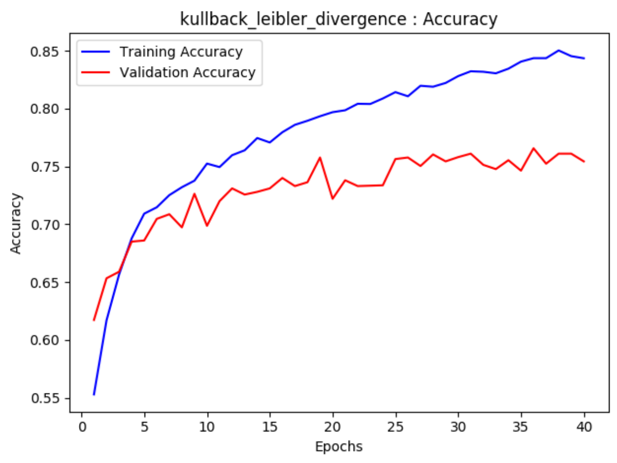
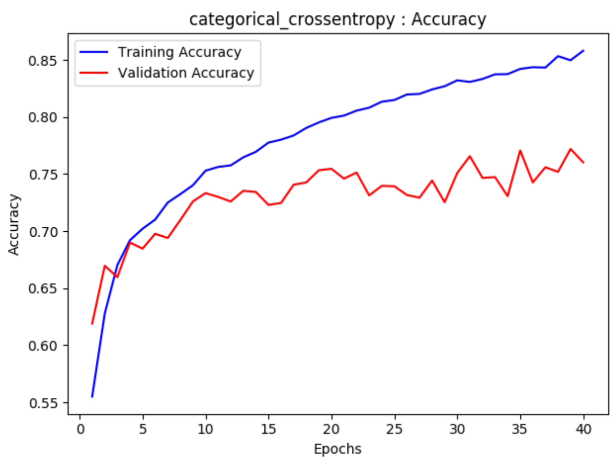
Here are the resulting loss by epochs :

hinge	squared_hinge
	
kullback_leibler_divergence	**categorical_crossentropy**

kullback_leibler_divergence	categorical_crossentropy
	

Here are the resulting accuracy by epochs :

hinge	squared_hinge
	

kullback_leibler_divergence	categorical_crossentropy
	

And the final accuracy by loss function :
“hinge” : 83.6
“squared_hinge” : 82.05
“kullback_leibler_divergence” : 66.66
“categorical_crossentropy” : 82.3

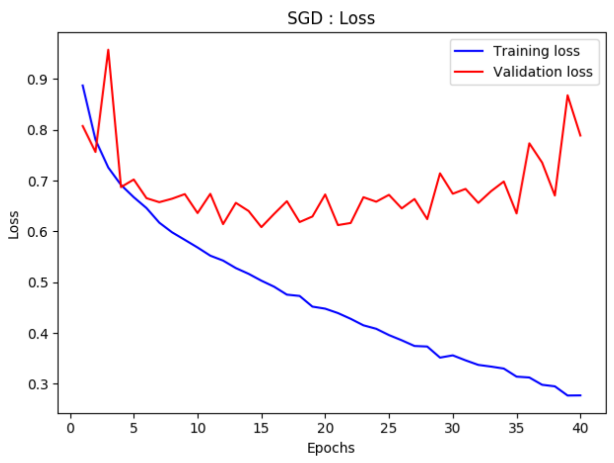
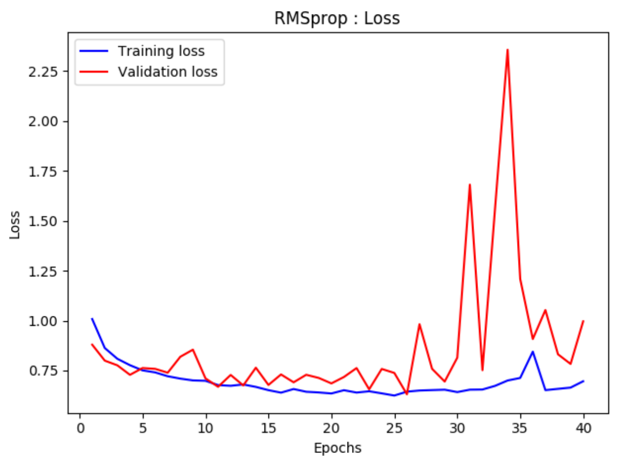
Except for the Kullback_Lieber that has a lower accuracy, the final values does not change much, but the impacts of the loss functions on the graphs really are according to the theory.

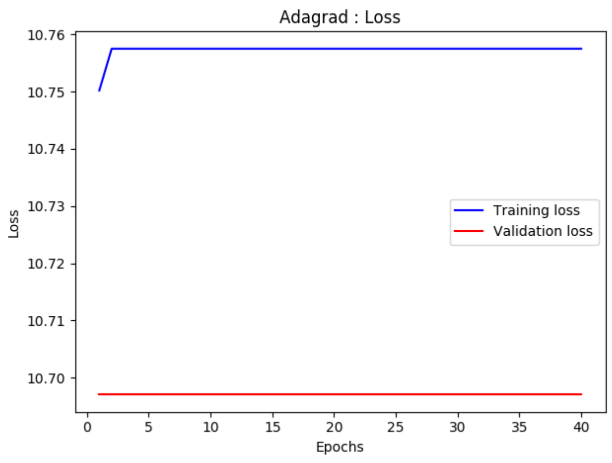
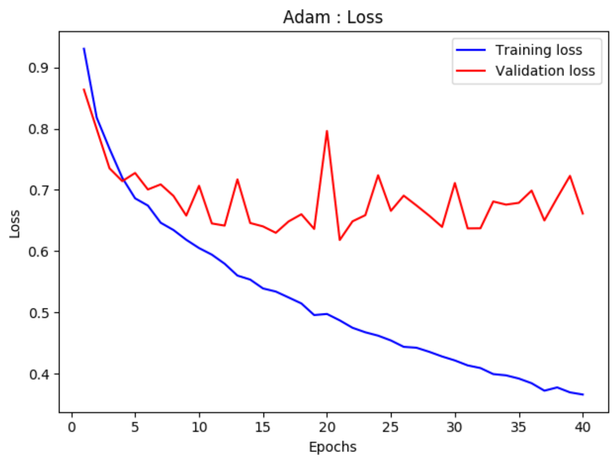
I kept categorical crossentropy for the remaining test, as the loss function on the graphs is smoother than hinge (we need that to analyze the impact of the next hyper parameters tuning) and the final accuracy is still high.

2. optimizers

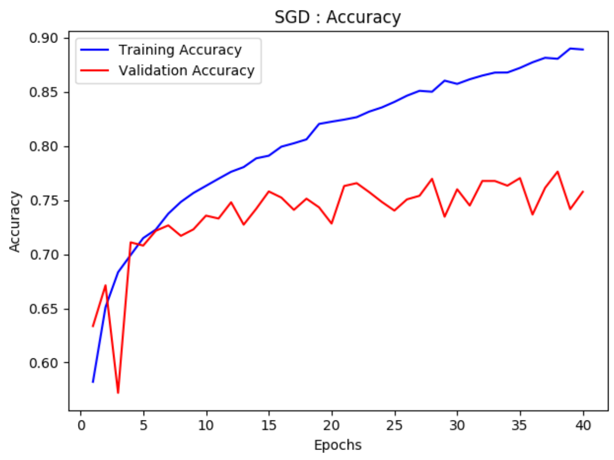
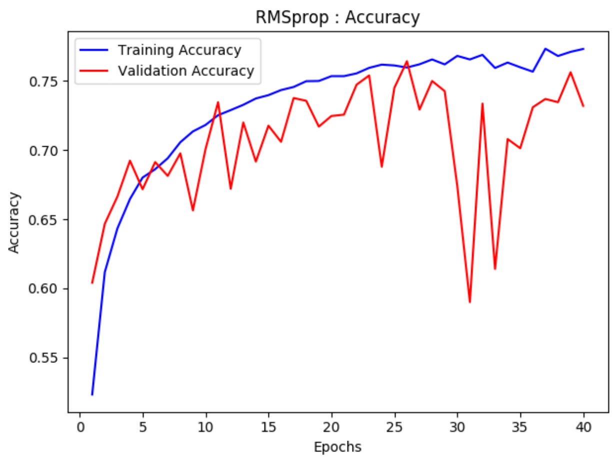
We have 4 different optimizers :
SGD
RMSProp
AdaGrad
Adam

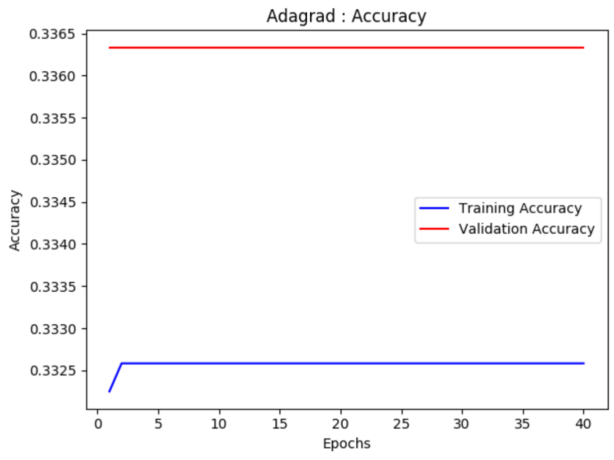
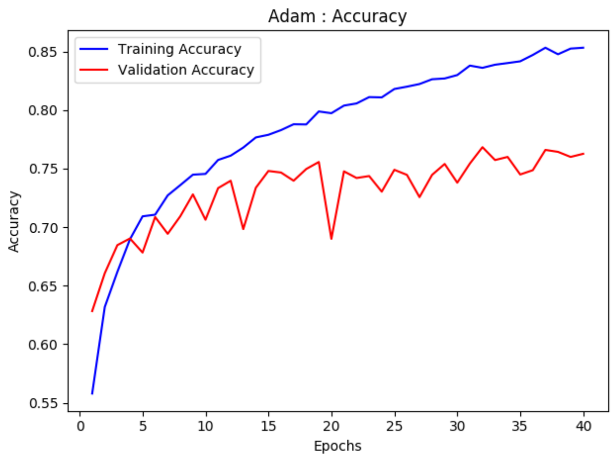
Here are the resulting loss by epoch :

SGD	**RMSProp**
	
AdaGrad	**Adam**

AdaGrad	Adam
	

Here are the resulting accuracy by epoch :

SGD	RMSProp
	

AdaGrad	Adam
	

And the final accuracy by optimizers :

SGD : 83.72

RMSProp : 82.97

AdaGrad : 83.92

Adam : 83.13

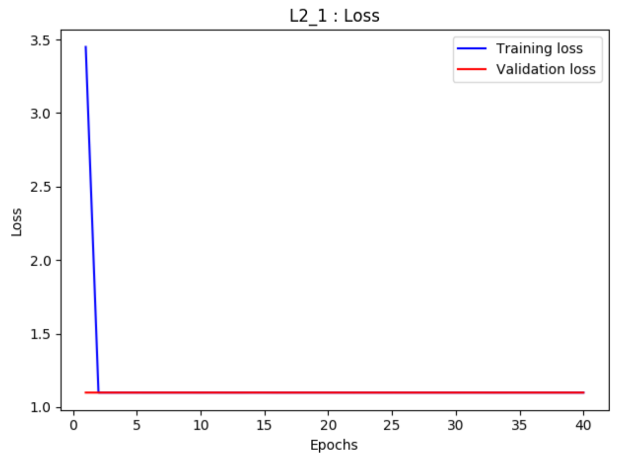
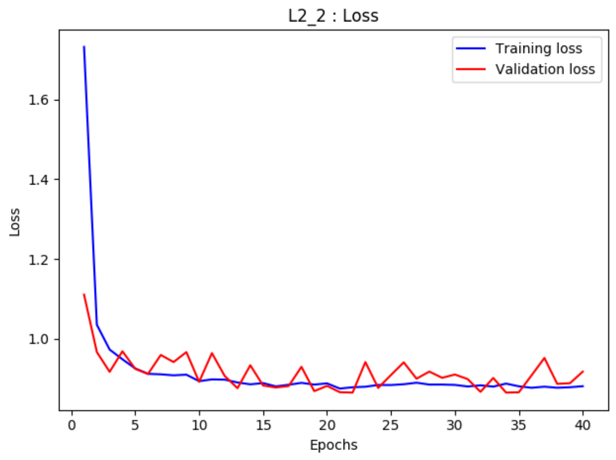
We can notice that the loss functions on the graph react as it is supposed to be : The SGD and Adam are similar and smoother, Adagrad is not noisy at all, and RMSProp is noisy.

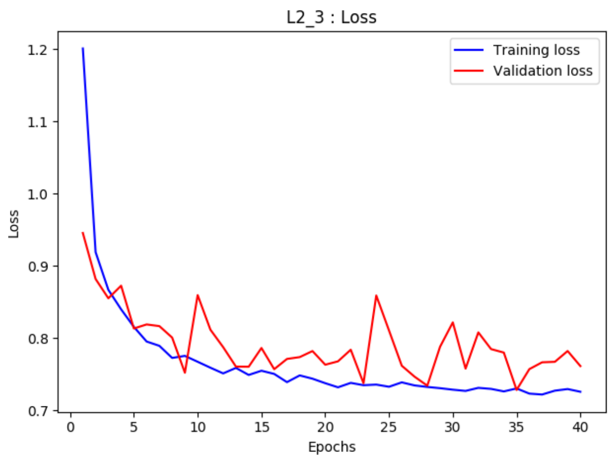
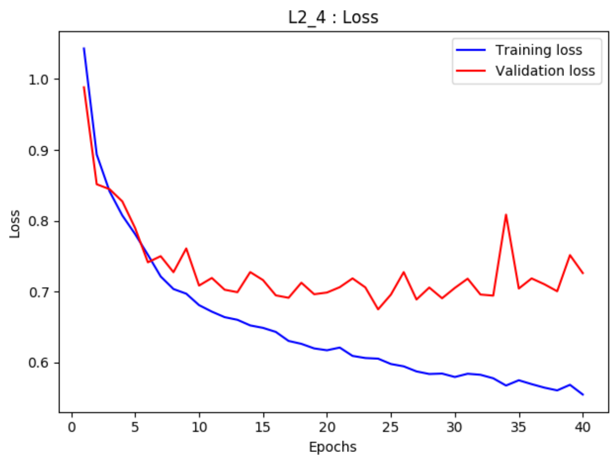
We can notice the impact of the optimizers too, as the accuracy values are higher than before.

3. Regularization l2

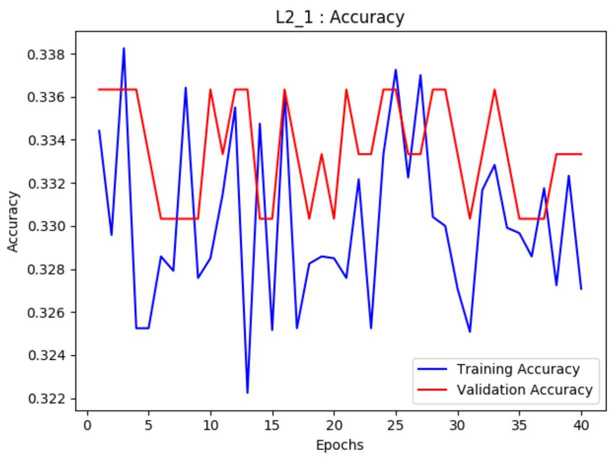
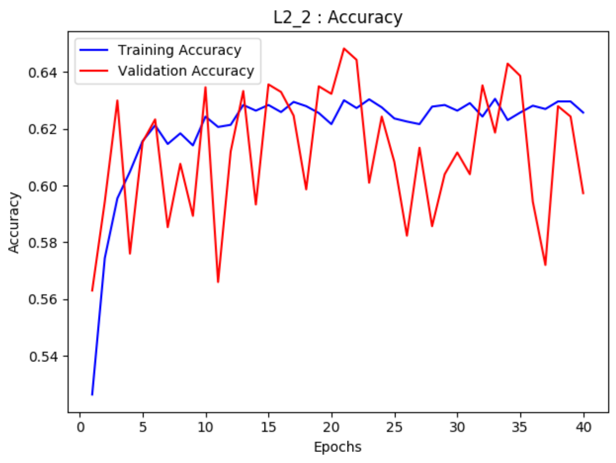
For the L2 reguralization, I tested with the values : 0.1, 0.01, 0.001, 0.0001

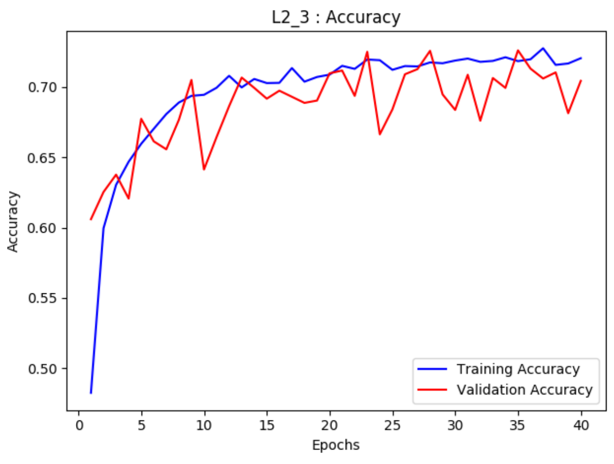
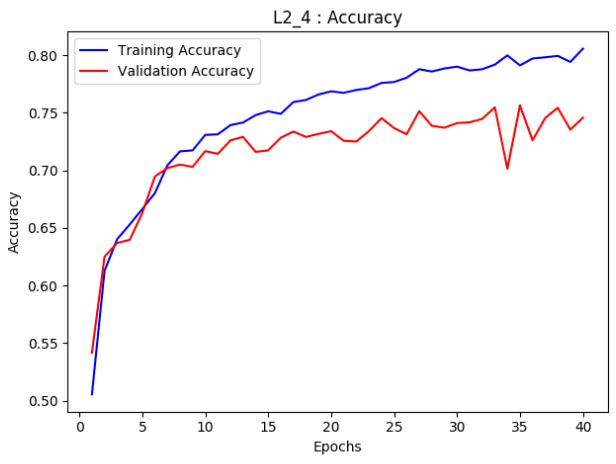
Here are the resulting loss by epochs :

L2(0.1)	**L2(0.01)**
	
L2(0.001)	**L2(0.0001)**

L2(0.001)	**L2(0.0001)**
	

Here are the resulting accuracy by epochs :

L2(0.1)	**L2(0.01)**
	

L2(0.001)	**L2(0.0001)**
	

And the final accuracy by value :

L2(0.1) : 66.66

L2(0.01) : 66.66

L2(0.001) : 79.85

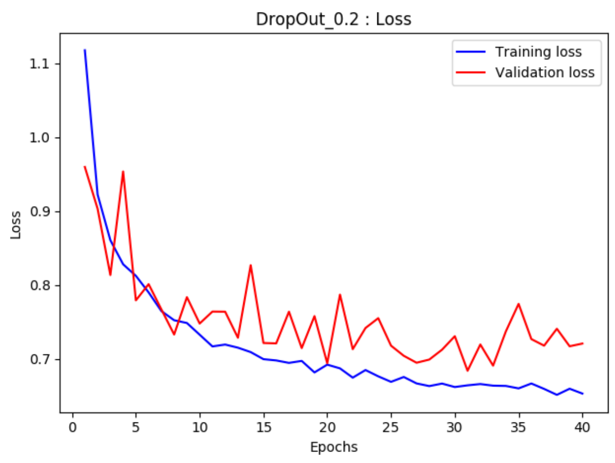
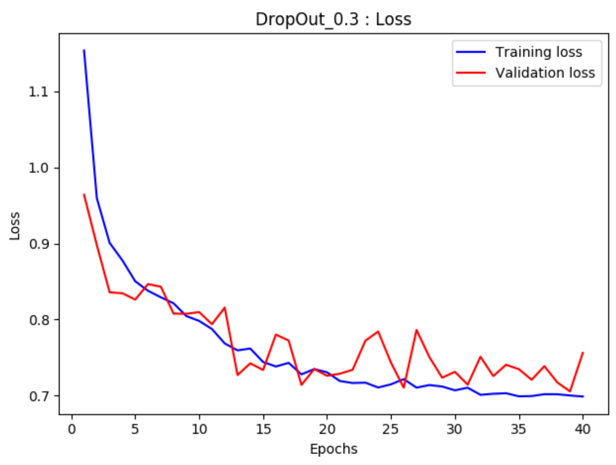
L2(0.0001) : 82.31

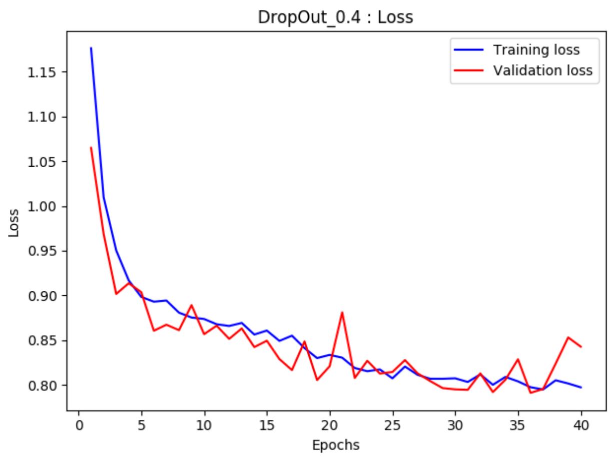
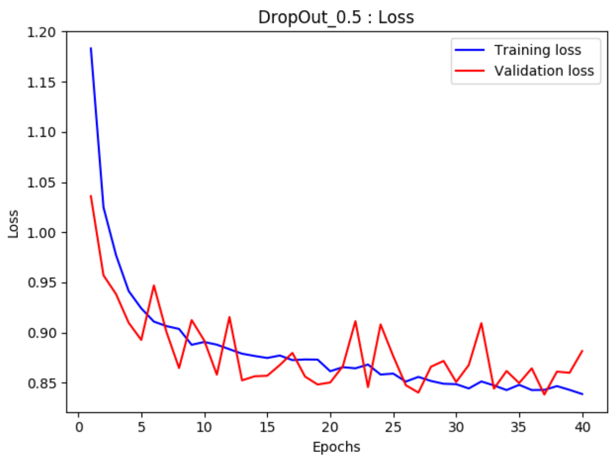
As expected, we can notice that the higher value of L2 are too high, introduce noice and lower the accuracy where the best value for L2 reguralization is 0.0001, as the curve is smooth and the final accuracy is the smallest.

3. Dropout

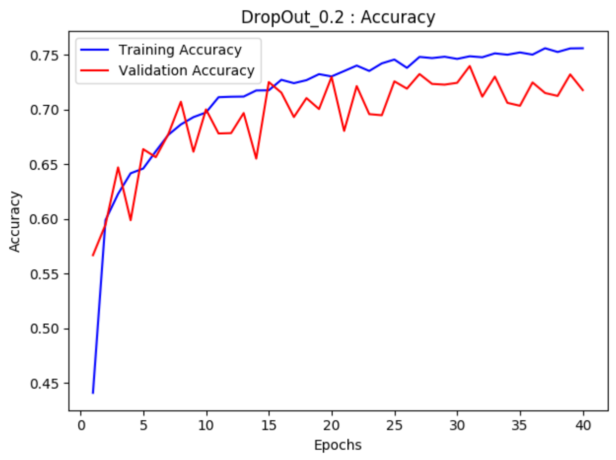
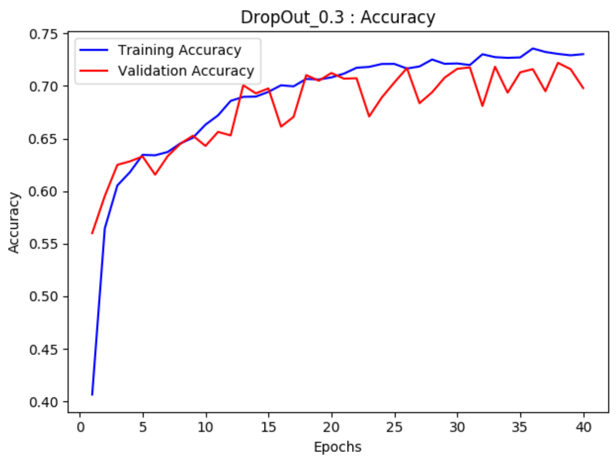
For the Dropout, I tested with the drop rate : 0.2, 0.3, 0.4, 0.5

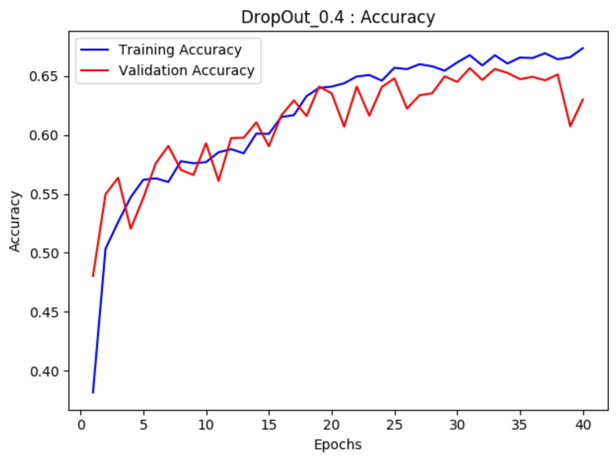
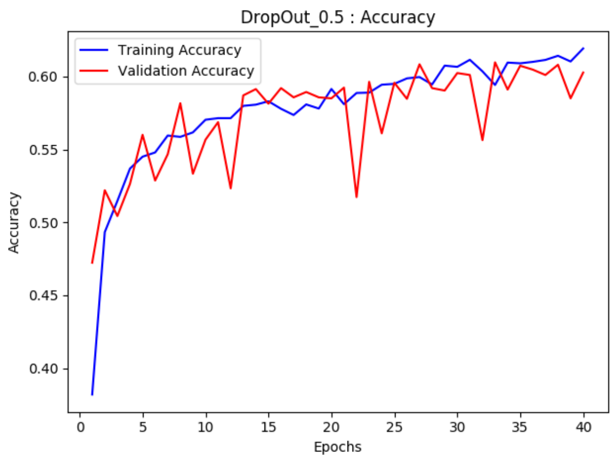
Here are the resulting loss by epochs:

Dropout(0.2)	Dropout(0.3)
	
Dropout(0.4)	**Dropout(0.5)**

Dropout(0.4)	**Dropout(0.5)**
	

Here are the resulting accuracy by epochs:

Dropout(0.2)	**Dropout(0.3)**
	

Dropout(0.4)	**Dropout(0.5)**
	

And the final accuracy by drop rate :
DropOut 0.2 : 81.62
DropOut 0.3 : 78.86
DropOut 0.4 : 78.62
DropOut 0.5 : 72.02

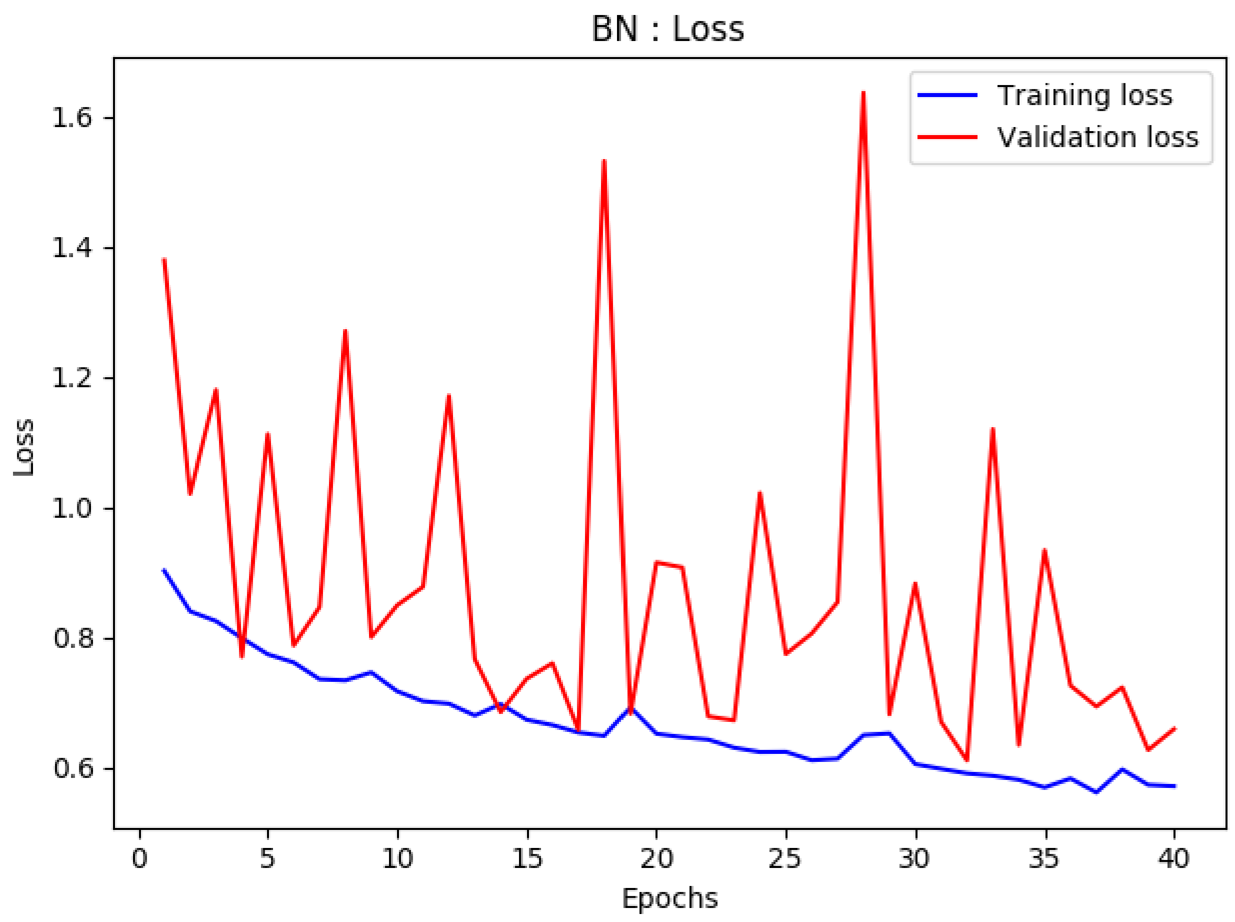
We can see that the best drop rate here is 20%. The curve is the smoothe and the final accuracy is the smallest. However we can notice that the accuracy is lower than before. That can be due to the small size of the data-set.

4. Batch Normalization

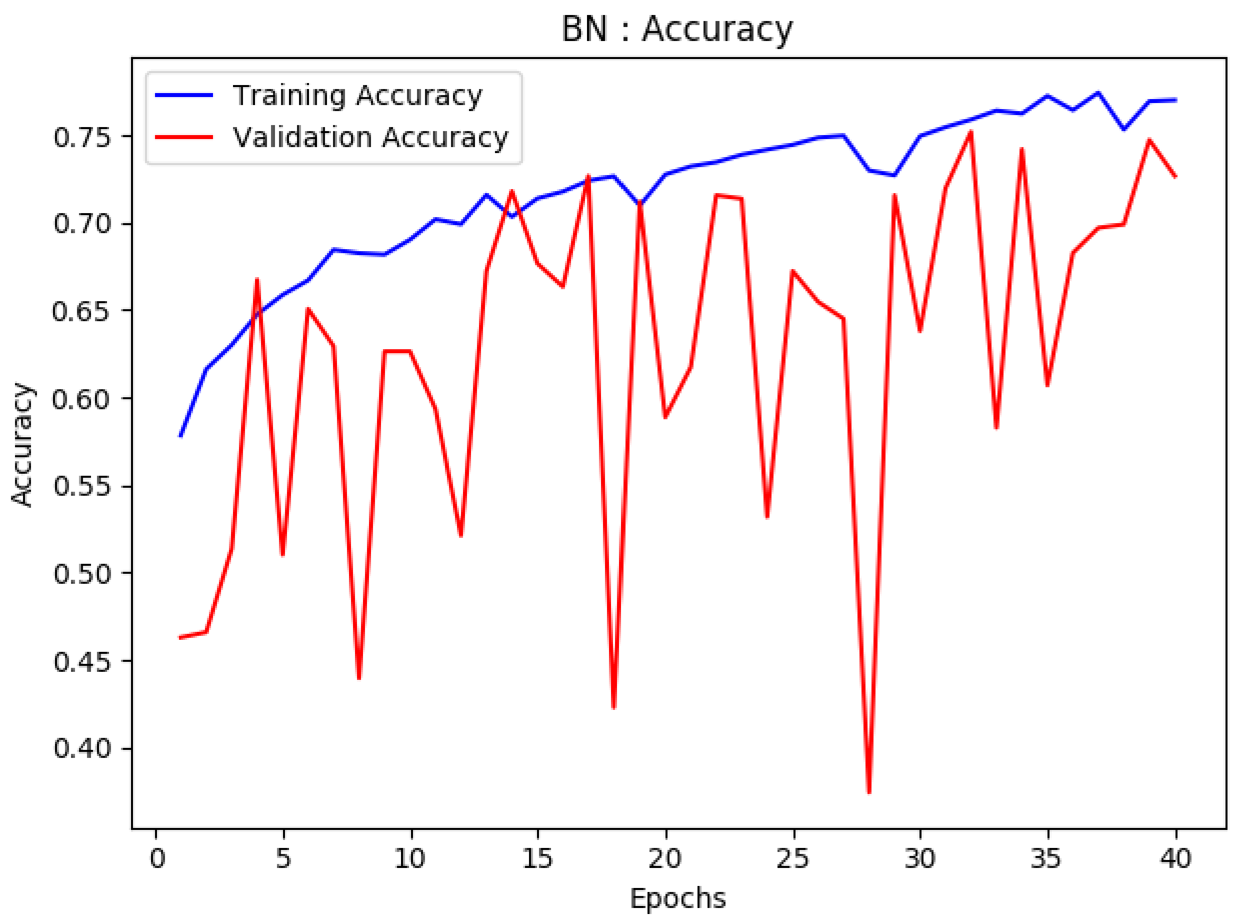
Here is a simplified version of my model architecture to show where the btach normalization is being done.

```
model = models.Sequential()
model.add(layers.Dense(512))

model.add(layers.Dense(128)
--> model.add(BatchNormalization())

model.add(layers.Dense(32))
--> model.add(BatchNormalization())

model.add(layers.Dense(16))
--> model.add(BatchNormalization())

model.add(layers.Dense(NB_CLASS, activation='softmax'))

model.compile(loss=loss_fct, optimizer=opt, metrics=[metric])
```

Resulting loss:



Resulting accuracy :



Final Accuracy : 80.63

The final value, high still, remain lower than the others. It may be cause the batches we are using a too small to implement the Batch Normalization regularization, or the dataset is too small.

II. UCI Crime Dataset

Data preparation

Here the situation is different than before, because even if there are way more attributes, they have already been normalized between 0 and 1, and according to their mean.

However, there are missing values, annotated by “?”. After trying several possibilities, I figured that the best way to process it is to replace all these missing values by “-1”. So, eventually, the model will learn to consider these values. Another way to proceed would be to assign the mean value to all missing ones, however the accuracy was lower when I tested it.

There is nothing to do for the labels, are they are float numbers between 0 and 1.

The function *split-data* first split the training data into a training set (4/5) and a testing set (1/5). Then, it split the training data again into a training set (4/5) and the validation set (1/5).

Model Architecture

The model architecture is :

```
model = models.Sequential()
model.add(
    layers.Dense(64, activation="relu", input_shape=(train_data.shape[1:]), kernel_
# model.add(BatchNormalization())
model.add(Dropout(drop))
model.add(layers.Dense(32, activation="relu", kernel_regularizer=dense_regularizer)
# model.add(BatchNormalization())
model.add(Dropout(drop))
model.add(layers.Dense(1, kernel_regularizer=dense_regularizer))
model.compile(optimizer=opt, loss=loss_fct, metrics=[metric])
```

with, by default :

```
seed = 7
loss_fct = "mse"
opt = "rmsprop"
metric = 'mae'
num_epochs = 50
batch_size = 1
dense_regularizer = None
drop = 0
```

HyperParameter tuning

1. loss functions

It is a model for single output regression, so the loss functions we can use are :

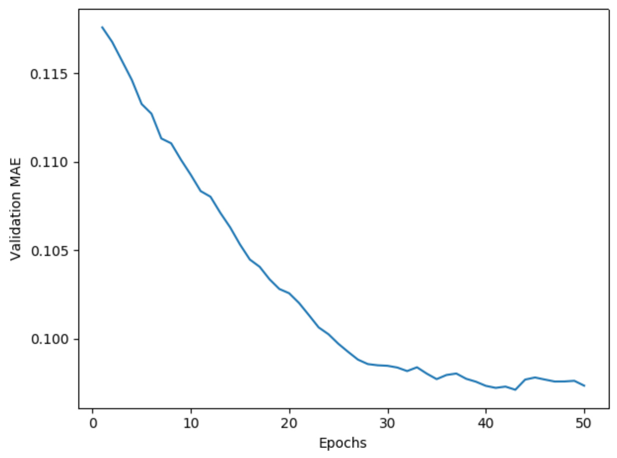
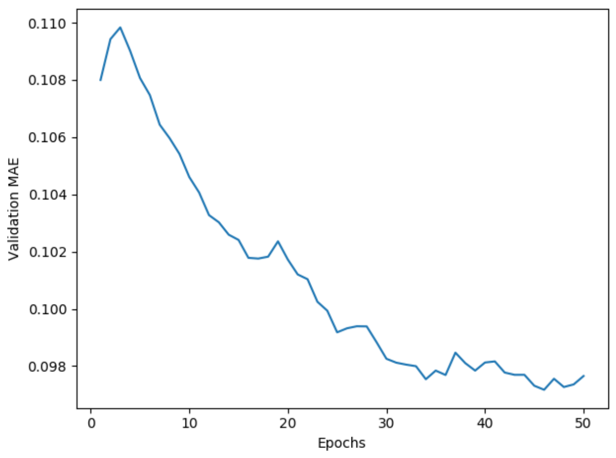
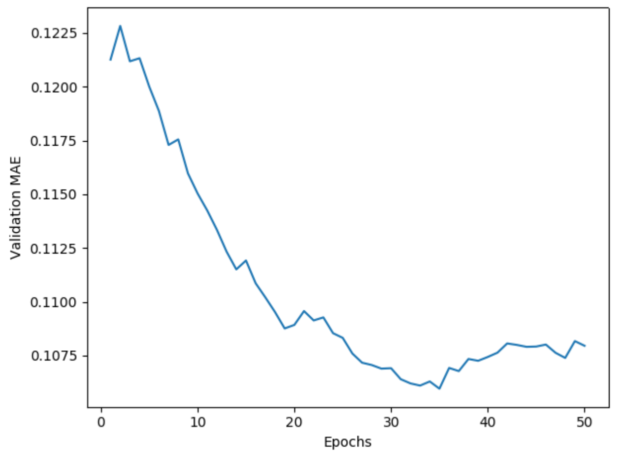
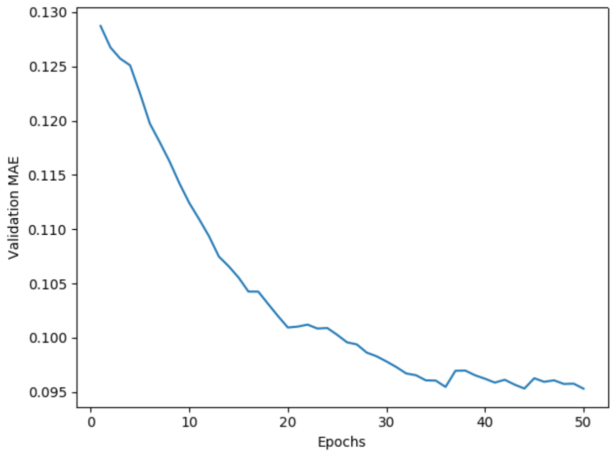
mean_absolute_error : L1

mean_squared_error : L2

logcosh : log-cosh

tf.losses.huber_loss : hubert

Here are the results :

L1	**L2**
	
log-cosh	**huber**
	

And the final mae by loss function :

mean_squared_error : 0.10046065195250474

mean_absolute_error : 0.0931244062137174

logcosh : 0.10108523902287678

huber : 0.09639699566163427

The final values does not change much, but the impact of the loss function on the graphs really is according to the theory :

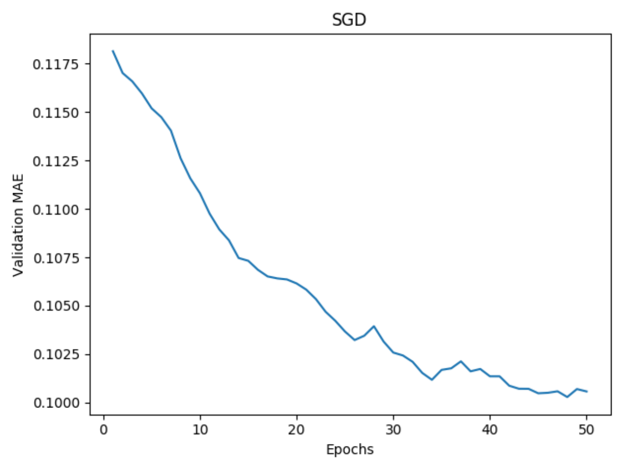
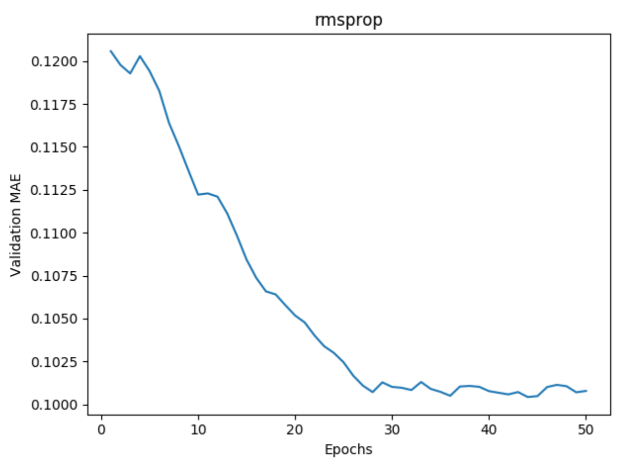
- L1 loss curve decrease smoothly
- L2 decrease more, but more abruptly
- logcosh : decrease even more but even more abruptly
- huber is the same as log-cosh but smoother

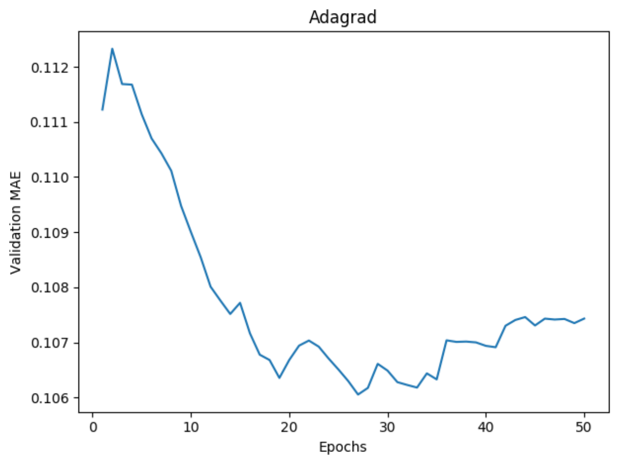
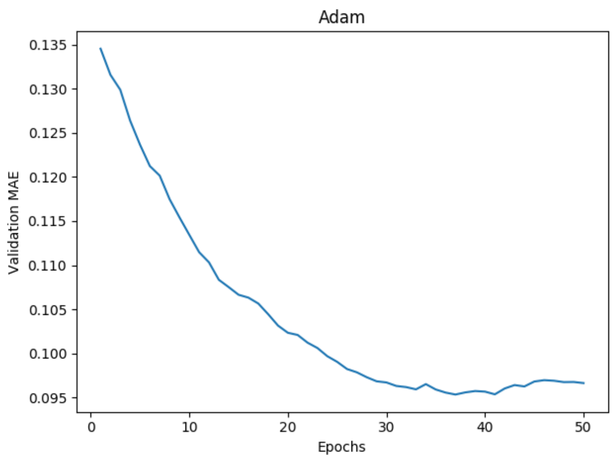
2. optimizers

I decided to kept Huber loss as the loss function for the remaining tests.

We have 4 different optimizers :
SGD
RMSProp
AdaGrad
Adam

Here are the results :

SGD	RMSProp
	
AdaGrad	**Adam**

AdaGrad	Adam
	

And the final mae by optimizers :

SGD : 0.09763676524862974

RMSProp : 0.09987740052811404

AdaGrad : 0.10475323565280922

Adam : 0.09552480519798856

We can notice :

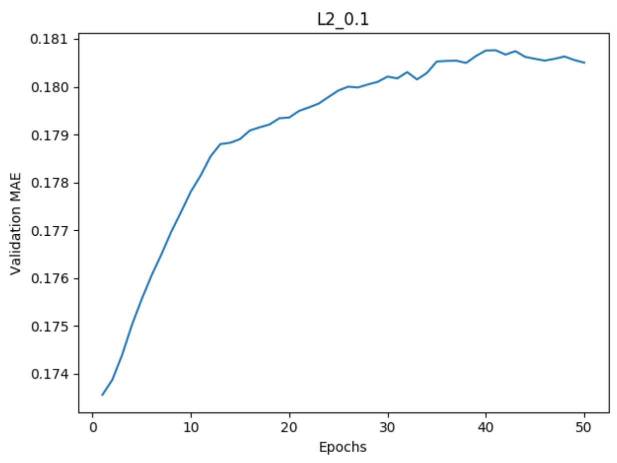
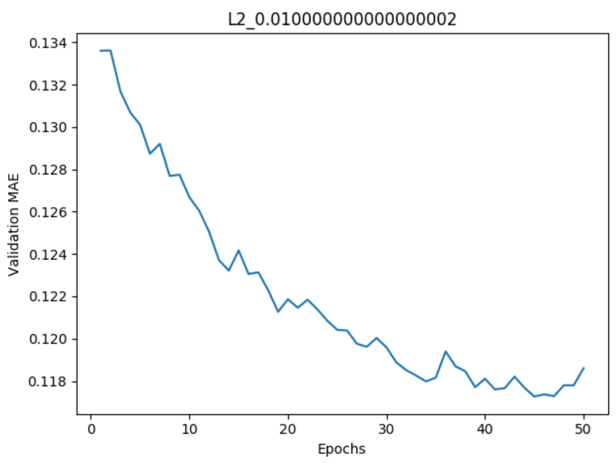
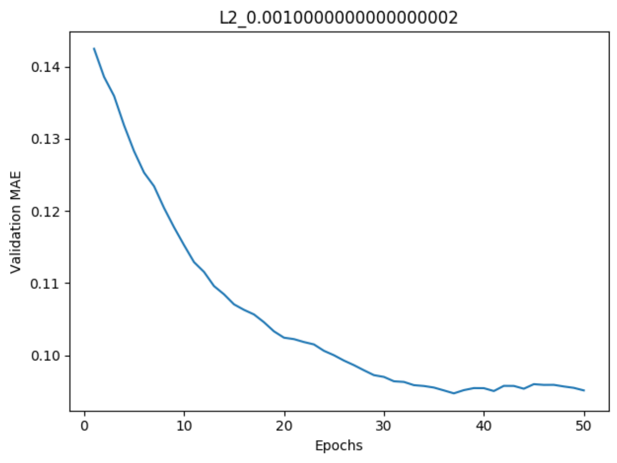
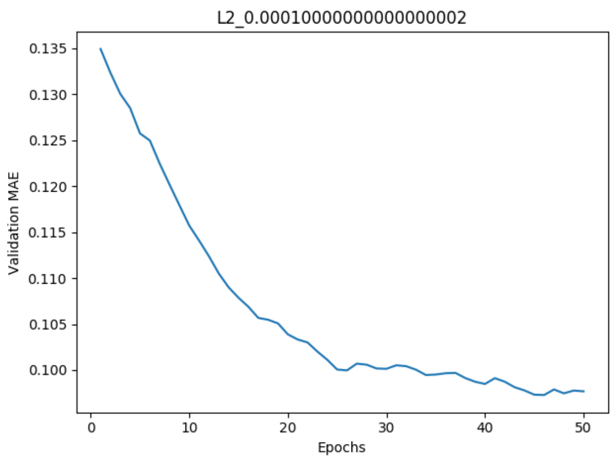
TODO

3. Regularization l2

For the L2 reguralization, I tested with the values : 0.1, 0.01, 0.001, 0.0001

Here are the results :

L2(0.1)	L2(0.01)

L2(0.1)	**L2(0.01)**
	
L2(0.001)	**L2(0.0001)**
	

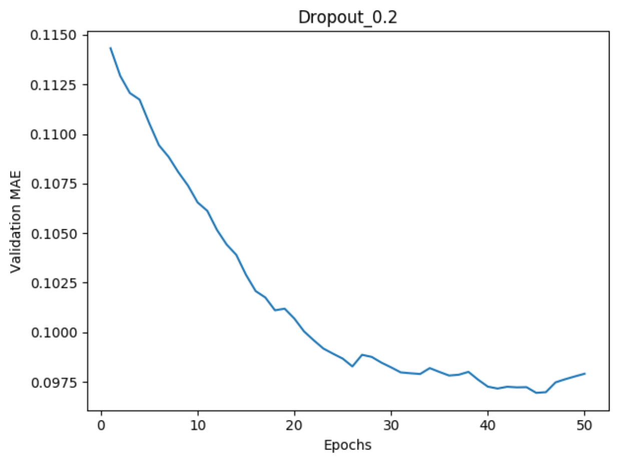
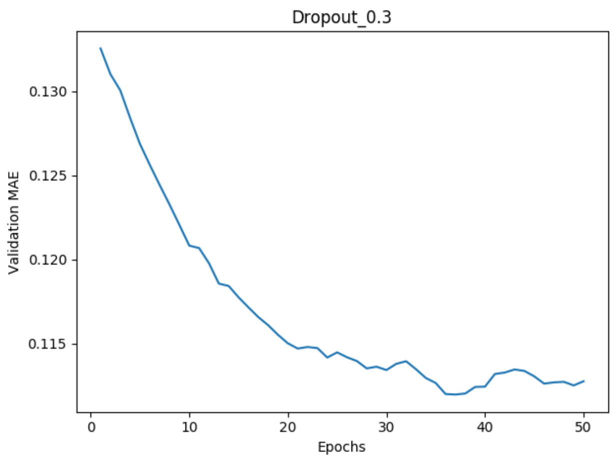
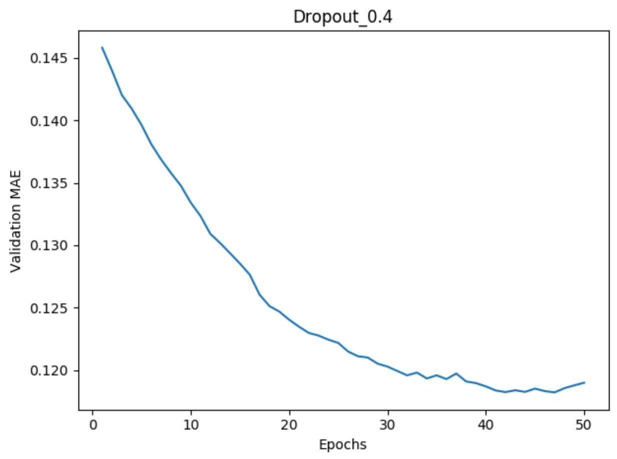
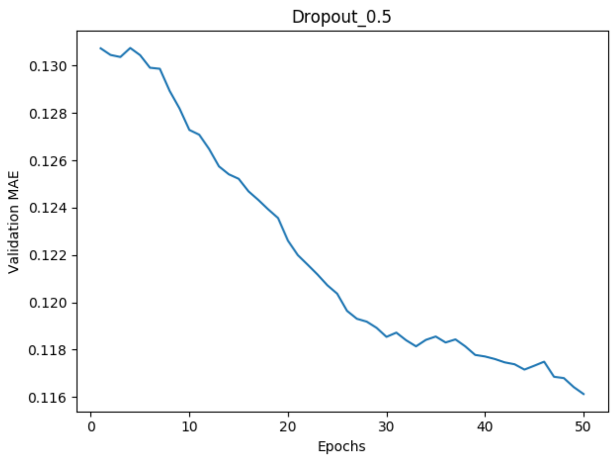
And the final mae by value :
L2(0.1) : 0.17312735126450143
L2(0.01) : 0.12496511331032437
L2(0.001) : 0.09454084980100112
L2(0.0001) : 0.09426881752756416

As expected, we can notice that the higher value of L2 are too high, introduce noice and lower the mae where the best value for L2 reguralization is 0.0001, as the curve is smooth and the final mae is the smallest. There is a small improvement from the precedent losses.

3. Dropout

For the Dropout, I tested with the drop rate : 0.2, 0.3, 0.4, 0.5

Here are the results :

Dropout(0.2)	Dropout(0.3)
	
Dropout(0.4)	**Dropout(0.5)**
	

And the final mae by drop rate :
DropOut 0.2 : 0.09813523426838802
DropOut 0.3 : 0.11199997028563652
DropOut 0.4 : 0.11976749522372099
DropOut 0.5 : 0.11228825491166974

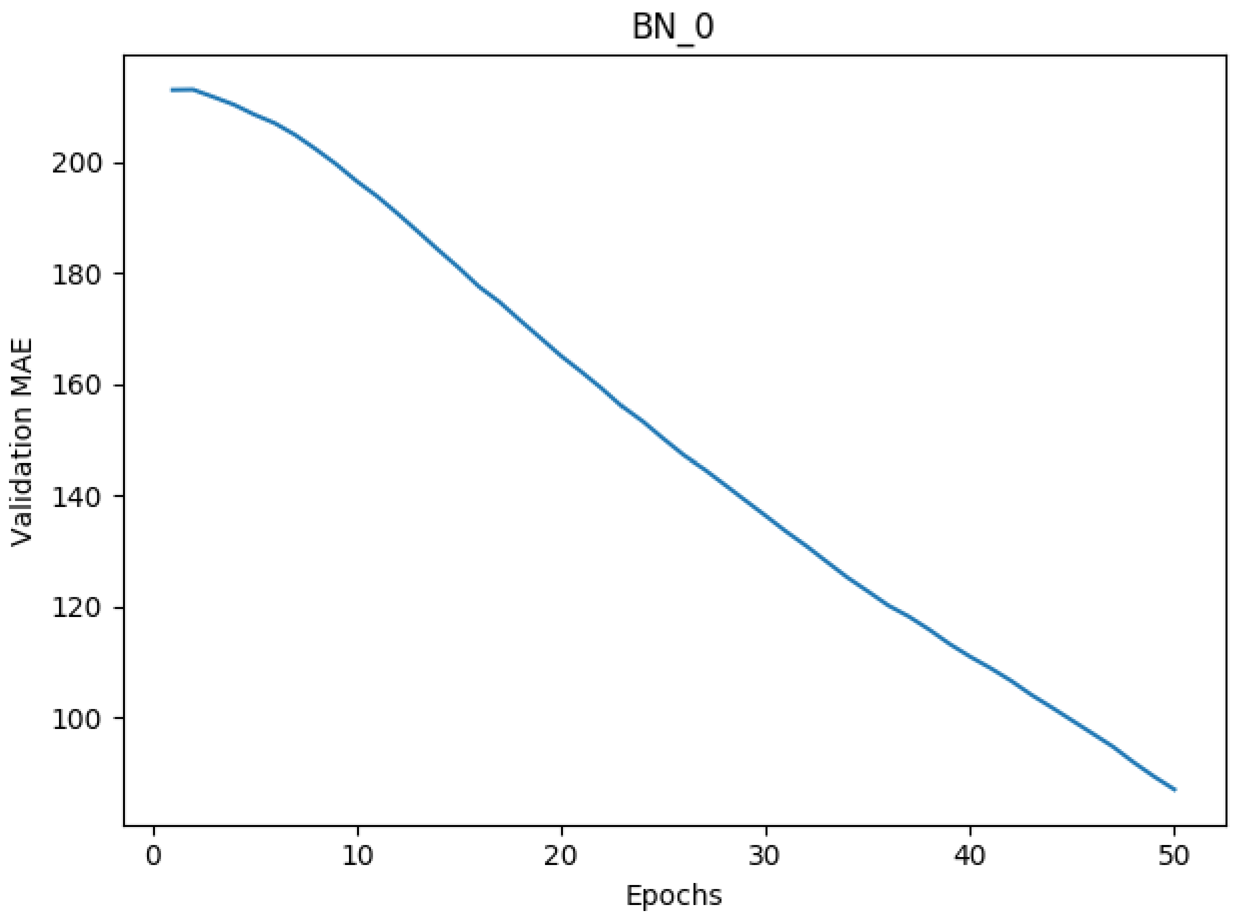
We can see that the best drop rate here is 20%. The curve is the smoothe and the final MAE is the smallest. However we can notice that the final mae is slighty superior that the one without regularization.

4. Batch Normalization

Here is a simplified version of my model architecture to show where the btach normalization is being done.

```
model = models.Sequential()

model.add(layers.Dense(64))
--> model.add(BatchNormalization())

model.add(layers.Dense(32))
--> model.add(BatchNormalization())

model.add(layers.Dense(1))
model.compile(optimizer=opt, loss=loss_fct, metrics=[metric])
```

Result :



Final Mae : 71.12431900329352

The final value is way to high. It may be cause the batches we are using a too small to implement the Batch Normalization regularization.