Double digit recognition in noisy images with Machine Learning

Lucas Caccia
260638467
lucas.page-caccia@mcgill.ca

Théo Szymkowiak
260619310
theo.szymkowiak@mail.mcgill.ca

Arnaud Massenet
260608613
arnaud.massenet@mcgill.ca

Abstract – This report summarizes different approaches to recognizing double digits in noisy environments.

I. INTRODUCTION

In this report, we explore different supervised learning approaches for preprocessing and labelling grayscale images. Particularly, the task at hand is to evaluate the sum of 2 digits on a single image. We analyze three different models: Logistic Regression, fully connected and convolutional neural networks.

II. PROBLEM REPRESENTATION & RELATED WORK

For the past decade, the convolutional neural network approach has established itself to be most effective algorithm for computer vision problems. This was shown by the AlexNet [1] developed in 2012 for the ImageNet challenge. For the more specific tasks of multi-digit recognition, 2 approaches are commonly used. The first one is to separate the digit segmentation and digit recognition steps [2]. A more computationally is to unify these 2 steps by training a very deep convolutional network. Google's address recognition [3] demonstrated the power of such model with over 98 % accuracy.

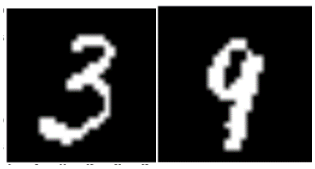
III. METHODOLOGY

Data Preprocessing:

In all the attempts discussed below, the main data preprocessing step was to remove the background by setting all the pixels smaller than a specific threshold to zero.

For some models, we attempted to separate the digits in an image. We used spectral clustering (sk-learn) to aggregate the pixels into 2 clusters. If, however, 1 cluster contained more than 80 % of the pixels, we used k-means with k = 2 to get a more balanced clustering.
We then created two images by extracting two 28x28 patches of the original image, centered at each cluster's centroid. Here are 2 results, one considered as a "success" and the other one as a "failure".

Figures 1., 2., 3.
Images from clustering process steps



Figures 4., 5.
Results from the clustering process

Figures 6., 7., 8.
Images from the clustering process steps

Figures 9., 10.
Results from the clustering process

A. Logistic regression

For logistic regression, we used the data given on the Kaggle website to train our classifier. The data consisted of 100 000 images of size 60x60. We decided to encode the images as 100 000 arrays of size 3600. One-hot encodings were used for labels.
We also tried training on the original MNIST data, and using the digit separator algorithm described above, estimate the sum by looking at each digit individually. In that case, each MNIST example was unraveled into a vector of size 28*28 = 784 instead of 3600. This part of work was done using the scikit-learn library functions.

The scikit-learn library offers different types of "solvers" which is use one of the following :

- "Non-linear conjugate gradient descent" (newton-cg)
- "Stochastic average gradient descent" (sag)
- "The limited memory-BFGS" (lbfgs)
- "Coordinate descent" (liblinear)

We decided to use all four methods and compare their results.
We presented a small summary below describing the execution of each method.
The Stochastic average gradient descent can be calculated as follows:

$$\theta'_t = \begin{cases} \theta_t & if\ t \le t_0 \\ \frac{1}{t - t_0 + 1} \sum_{k=t_0}^{t} \theta_k & t \ge t_0 \end{cases}$$

Where θ_t is:

$$\theta_t = \theta_{t-1} - \frac{n_0}{(1+\lambda n_0 t)^{\alpha}} \left(\frac{dL_{t+1}(\theta_t)}{d\theta} + \lambda\theta_t\right)$$

The Non-linear conjugate gradient descent can be can be done as follows:

1. Get the steepest direction:
   $\Delta x_n = -\nabla_x f(x_n)$
2. And get the conjugate direction:
   $s_n = \Delta x_n + \beta_n s_{n-1}$
3. We optimize α_n :
   $\alpha_n = argmin\, f(x_n + \alpha s_n)$
4. Then we update:
   $x_{n+1} = x_n + \alpha_n s_n$

Note that β_n can found using some formulas such as “Fletcher-Reeves”, “Polak-Ribière”.

The limited memory Broyden-Fletcher-Goldfarb-Shanno method is based on Newton’s method and Hessian matrices.

The coordinate descent updates x_n as follows: $x_{k+1,i} = argmin\, f(x_{k+1,1}, \dots, x_{k+1,i-1}, y, x_{k,i+1}, \dots, x_{k,n})$

As we will see later, using the raw Kaggle training data yields bad predictions. This is why we decided to use another method that involved clustering. The idea is that we can easily train our classifiers to predict the value on single digit images. Thus, we can use the clustering algorithm described above to cluster the two digits on the Kaggle data images and then use logistic regression.

B. Neural Network

The objective of this section is to explain the implementation of a Neural Network. The MNIST data was encoded as for logistic regression.

The Neural Net was trained using back-propagation with a sigmoid activation function for the input, hidden and output layer. The sigmoid activation function is as follow:

$$h(t) = \frac{1}{1 + e^t}$$

In order to do back-propagation and compute the gradient with respect to the parameters we also need its derivative:

$$h'(x) = x * (1 - x)$$

Where x is just the following pre-activation function:

$$a(x) = W^T x + b$$

Then, we can do back-propagation by computing the delta (or error) and back-propagate it onto the previous layers (except the input layer). We call this SGD (Stochastic Gradient Descent).

The strategy used to recognize the double digits in the images is as follow: The network does a patch-scan with patches of 28*28 pixels every 2 pixels. For a 60*60 image, that is 256 scan per image. Due to the high cardinality of the input vector (728) we could only train our neural net on ~200 training examples.

In order to improve the efficiency of the network, 2 techniques were used:

1) Dropout[4]. Dropout has established itself as a highly efficient regularization method, by decreasing the model's variance. The principle is simple: the gradient on a hidden unit can be set to 0 at any backpropagation iteration with probability p (usually 0.5)

2) Momentum. The main idea is to add some of the previous delta on a synapse to preserve the momentum of the back-propagation.

$$W_n = W_{n-1} + \alpha\Delta_n + m\Delta_{n-1}$$

C. Neural Network with Convolution layers

Before doing any hyper-parameter tuning on our CNNs, we used a basic architecture as a baseline to see what data encoding worked best. This architecture was composed 3 layers: an input, convolution(hidden) and output layer. It is referenced as the baseline model below.

Encodings

Our first attempt(1) at a CNN was trained directly on the MNIST dataset. The output vector was encoded as described in the previous sections. Digit extraction from the original image was done as described above.

We then tried modifying the training set label into a 10 dimensional probabilistic vector of the possible digits involved: For example, if the sum was 2 (assuming a uniform distribution of digits), the possible combinations were 0,2 and 1,1, giving us the vector:

$$[0.25\ 0.5\ 0.25\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$

Using the basic architecture, a model(2) was trained on a training set composed of 80% of the data provided to us in 'train_x.bin'. The rest was used as a test set.

Data creation

Inspired by *Google's address recognition CNN [3]*, we decided to unify the digit localization and recognition steps. But to do so, we needed more examples, with better labels.

Using the original MNIST dataset, we generated 3 million 60 x 60 images, each containing 2 digit. No backgrounds were added. We randomly selected 2 points on the grid, and then inserted the digits. We enforced that the center of 2 images could not be closer than a fixed distance to give the same overlapping as in the training set. By visual inspection, we found that setting a distance of 15 pixels mimicked best the data given to us.

We also generated another 2 million examples with background, where each background came from a 60x60 patch of a handpicked collection of 20 backgrounds. The motivation behind this process is explained in the next section.

Our Y vector for this data was a 55 dimensional-vector, using 1-hot encoding. 55 is the number of possible unordered combinations of 2 digits. We split the data provided to us into sets of 50k examples, serving and cross-validation and test sets.

After training with the baseline model, and finding this encoding to be more successful,

We began optimizing the CNNs hyper-parameters.

All of the hyper-parameters were found through cross-validation. This includes the choice of optimizer, weight initialization, kernel/filter sizes. For the number of layers, the layer ordering, the size of each layer and the pooling size, we followed basic axioms for CNN architecture [4]. From there cross-validation was performed, limiting the search space to values/configuration that could fit in our machines and that had a respectable training time.

Another attempt was to relabel the training set provided to us. We took the correctly predicted images from our model and attached a new label to it. This created a dataset containing 96 410 examples with 55 dimensional vectors as output. We trained a new model on this data without removing the background.

IV. RESULTS

A. Logistic regression

We extracted the data using the methodology
presented in the previous section. We used the four method to make prediction on our data and we calculated different metrics to evaluate them. For the training of the classifiers we selected 20 000 images and their corresponding labels from the data given on Kaggle. Then we made predictions on the train data and evaluated the results.

	Precision	*Recall*	*F1-Score*	*accuracy*
"newton-cg"	0.215	0.187	0.192	0.187
"liblinear"	0.206	0.187	0.190	0.187
"lbfgs"	0.158	0.157	0.155	0.157
"sag"	0.172	0.165	0.161	0.165

By looking at the metrics both "newton-cg" and "liblinear" scored the highest. However, even these "best" classifiers performed very poorly. The main cause for such poor results comes from the fact that we used the data from Kaggle to train our classifiers. The issue comes from the labeling of the data. The main cause for such poor results comes from the poor data labelling. Indeed, by using the sum as our label, we are giving the model 2 tasks: identifying the digits and adding them together. While it may be able to do both things separately, it fails to do both simultaneously. The model tries to extract similar features between digits 2,3 and 5,0 because they have the same label. However, it is obvious to a human that they share no common features. This is why we decided to train our classifiers with data from MNIST and then make predictions on the 2 extracted digits from each image. We trained our classifiers on 20 000 entries of the MNIST data and tested on 10 000 examples of the provided training set. The results are as follows:

	Precision	*Recall*	*F1-Score*	*accuracy*
"liblinear"	0.888	0.888	0.8877	0.888
"lbfgs"	0.902	0.902	0.901	0.902

Using the previously trained classifiers, we applied our clustering algorithm to make prediction on the train data from Kaggle. We then measured the performance of each classifier:

	Precision	*Recall*	*F1-Score*	*accuracy*
"newton-cg"	0.5	0.4	0.4	0.4
"liblinear"	0.5	0.4	0.4	0.4
"lbfgs"	0.4	0.2	0.267	0.2
"sag"	0.4	0.2	0.267	0.2

Again "newton-cg" and "liblinear" scored the best overall but the results are lower than what we could have expected. This will be discussed in the next section.

B. Neural Network

Using 200 rows from the training data from Mnist, we were able to get a 60% accuracy on a Mnist validation set after 292 epochs. The weights of the neural net are initialized randomly uniformly from 0 to 1 included. The neural net was made of 2 layers with 401 and 20 nodes respectively. Using 603 epoch, we achieved 66.5% accuracy on the Mnist validation set (however it took more than 40 minutes to train).

Training with Mnist raised several challenges:

1) The neural net is composed of fully connected forward layer (not convolutional layers). Therefore, we need to perform patch scan on the input data (scan 28 * 28 squares on the 60 * 60 image) and find the 2 numbers with the highest confidence which takes time and is tricky when the two digits are the same.

2) Mnist data was processed to eliminate noise. The training data that we were given was purposely merged with background noise. Without removing the background noise, the neural net accuracy was ~10%. After removing the background noise using a threshold of 240, the accuracy was a little bit higher (~20%) but still not great.

Following those observations, we chose a better training flow. Each epoch will only train on 10 different example (one for each digit), and for each digit, the image will be randomly picked from the bank of images (loaded from the Mnist data).

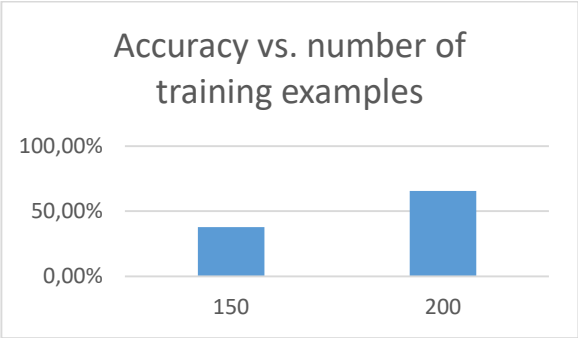
When talking about error we refer to the norm of the difference vector between the input and the output summed over the 10 training examples at each epoch.

$$e = \sum_{i=0}^{10} ||in_i - out_i||_2 = \frac{1}{2} \sum_{i=0}^{10} \sum_{k=0}^{10} (in_{i,k} - out_{i,k})^2$$

Following this error definition, it took around 3000 epochs on a training set of 150 different images to reach a 0.005

average error (over last 25 errors) and a 50% accuracy over Mnist validation set.

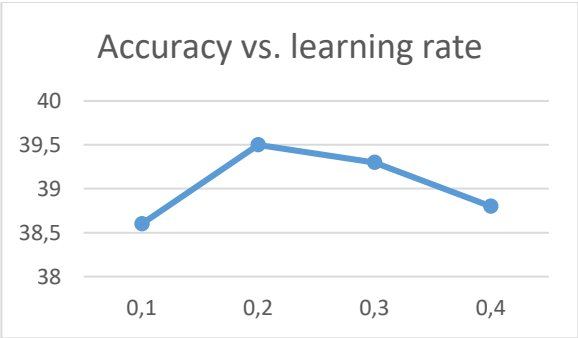
The new architecture consists of 2 layers: one with 718 hidden nodes and one with 200 hidden nodes. This architecture forces the network to "extract" features from the 784 node input layer twice. The following graph summarizes the accuracy on the Mnist validation set.



Accuracy of Neural net with size of training set

We could not do many mores test cases because training the network on more examples takes a long period of time. Of course, the capacity of the network had to be augmented as the number of training example rose. For 200 training examples, the layers were: 718 nodes, 400 nodes, 50 nodes.

In order to compare the different value of hyper parameters, we are going to use a training set of 200 different images to make the net converge faster. The momentum is set a 0.9. The back propagation stops when the error is below 0.005. All those training sessions were limited to 1000 epochs.



Accuracy of Neural Net with learning rate

We can see that the best accuracy is obtained with a learning rate of 0.2. From other tests we deduce that minimizing the momentum would slow down the algorithm. A momentum of 0.4 actually stuck the back-propagation in some local minima with 0.03 error.

The accuracy on the data from the kaggle competition was 10%. This is accuracy is not great and can explained from a variety of factures. See discussions.

C. Neural Network with Convolution layers

	Model	*accuracy*
(1)	Basic model trained on Mnist data, combined with digit separation	77.32
(2)	Basic Model with probabilistic encoding	21.20
(3)	Basic Model trained on "no-background" generated images	85.29

After trying 3 different encoding approaches, we concluded that the last encoding, using a 55 dimensional vector and training on generated images was worth pursuing.

Our most successful net was composed of 3 sequences of alternating convolutional and pooling layers[5] (See Appendix). Each convolutional layer doubled the depth of our input, while each max pooling layer halved the other 2 dimensions. We then flattened the input, and added 2 fully connected layers. All layers were interspersed with dropout. From there, all other models trained replicated this architecture.

	Model	*accuracy*
(4)	2-seq CNN	94.30
(5)	3-seq CNN	95.53

As we were experimenting with possible CNN architectures, we found that always having a non-decreasing volume (e.g. the product of all the dimensions of your input) throughout the convolutional layers gave much better results. We also found the best dropout probability to be around 0.2, which disagrees with the general rule of setting it to 0.5. Our hypothesis is that in this scenario, it is much harder to overfit the data since the digits in our generated training set are also contained in the test set.

Upon visual inspection of our misclassified data, it was clear that mislabeling was the result of a faulty background

removal. Indeed, if the background was too bright, it would not be completely removed. On the other hand, raising the threshold value could result in the complete erasure of a digit. Several attempts were made to resolve this problem. This was the motivation for creating images with backgrounds, and relabeling the original data, as described in the previous section.

Figure 11.
Faulty Background removal

We also tried removing the background prior to training. The motivation behind this is that our background removal process tends to sharpen the edges of the digits. We thought training on data more similar to the ones we are making predictions would yield better results. Note that test set for this model was 10% of the 96 410 relabeled examples in the provided dataset. We kept the hyper-parameters fixed, therefore there was no need for a cross-validation set.

Figure 12.
Image generated without background

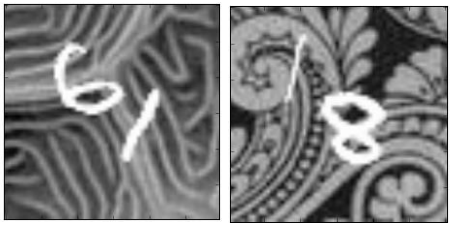


Figure 13., 14.
Image generated with background

Figure 15.
Image generated with background after background-removal

	Model	*accuracy*
(5)	Final model trained on generated images with background	92.3
(6)	Final model trained on the relabeled training set	91.2
(7)	Final model trained on generated images with bg, but bg removed prior to training	96.9

Our last attempt to get better accuracy was to combine the results from several models. We used two bagging techniques. The first one was to extract the most confident answer out of all models. The second one was to output the average result of each model's prediction. The latter was quickly dismissed, as it did not improve the results. We also tried different background removal thresholds. We then proceeded to bag the predictions of all models on all different thresholds.

	Model	*accuracy*
(8)	Most confident of model 4, with multiple bg removal thresholds	96,5
(9)	Most confident of 8,5	96,7
(10)	Most confident of 8,5,6	96,9
(11)	Most confident of 8,5,6,7	97,5

The CNNs were built using the Keras library, a high-level library that used Theano in the backend

V. DISCUSSIONS

A. Linear regression

As said above, the results we got using the clustering algorithm are not on par with what we got with the MNIST test data. This is the results of 2 main factors. Firstly, we can understand that our clustering algorithm was not perfect and could be further improved. Another reason why we did not perform as good as on the MNIST test data comes from the fact that for each entry we need to correctly predict 2 digits instead of one. Thus, if we are able to predict half of all digits correctly but only one per "pair" of digits then our accuracy will be under 25%. The combination of these 2 facts explain the results of our predictions.

B. Neural Network

It is quite difficult to cross-validate and optimize the parameters of a Neural Network because the training takes an enormous amount of time. We used a cloud server running 24h/24h which allowed me to test multiple parameters, but unfortunately we could not test every possible combinations of hyper-parameters (e.g. training data size, validation data size, etc.). The more training example, the more time (epochs) the neural net takes to converge using back-propagation. For example, training with 500 examples takes ~10000 epochs.

Neural Nets with forward layer works quite well in digit recognition but implementing Convolution layer combined with Max Pooling layers would have made the predictions much better, as shown by our CNN results.

Regarding the accuracy on the Kaggle dataset, the low accuracy can be attributed to a difficulty to train the network on a bigger dataset.

C. Neural Network with Convolution layers

We found a net increase in performance of our last model, trained on data which previously had a background. This suggests that our loss of accuracy emerged from the background removal process. It would have been interesting to work on a better background removal algorithm. Knowing that the best CNN architecture has less than .25%[6] error on the MNIST data, having a perfect background removal and digit segmentation algorithm could have increased our prediction by more than 2 %.

VI. CONTRIBUTIONS

We hereby state that all the work presented in this report is that of the authors. Théo implemented the Neural Network, Arnaud implemented the linear regression and Lucas did the CNN work. The whole team worked on the report.

VII. REFERENCES

[1] ImageNet Classification with Deep Convolutional Neural Networks, A. Krizhevsky, I.Sutskever, G. E. Hinton. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] Multi-digit Recognition using Convolutional Neural Network on Mobile, X. Yang, J. Pu http://web.stanford.edu/class/cs231m/projects/final-report-yang-pu.pdf

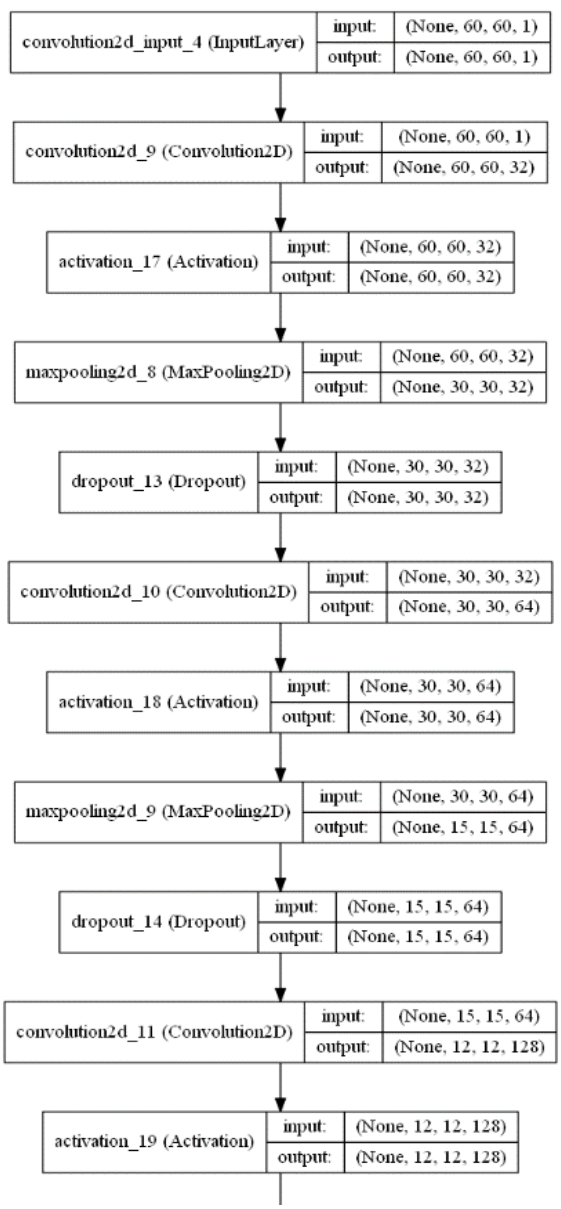
[3] Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks, Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, Vinay Shet https://arxiv.org/abs/1312.6082

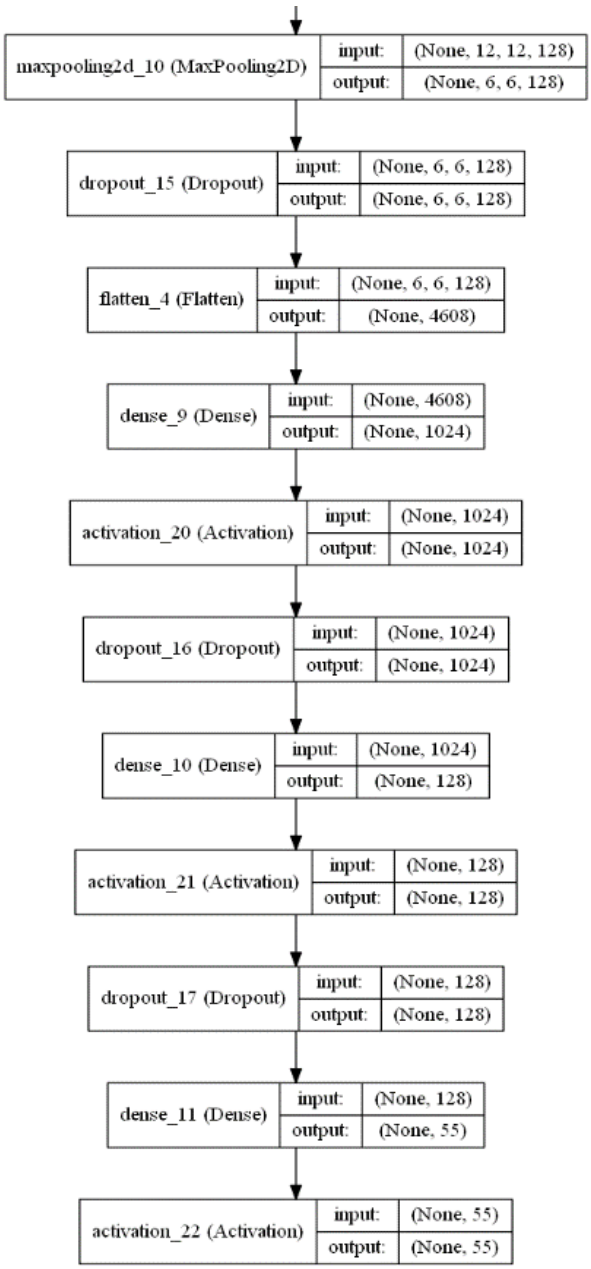
[4] Dropout: A Simple Way to Prevent Neural Networks from Overfitting, N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salahutdinov https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[5] Quora answers about:" What is your approach to convolutional neural network design?" https://www.quora.com/What-is-your-approach-to-convolutional-neural-network-design

[6] List of MNIST accuracy: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

APPENDIX :





Here is a visual representation of our most successful CNN's architecture.