Modified Digits

COMP 551: Applied Machine Learning

Edward Smith
edward.smith@mail.mcgill.ca
260570951

Navin Mordani
navin.mordani@mail.mcgill.ca
260744902

Theophile Gervet
theophile.gervet@mail.mcgill.ca
260612917

I. Introduction

The problem at hand is to find a method to accurately identify two single digit numbers from a 60 by 60 pixel image and output their sum. Our dataset consists of 100000 images created by placing two 28 by 28 pixel images of a single digit white number, taken from the well-known MNIST dataset, in a 60 by 60 pixel image filled with noise. While identifying digits of the MNIST dataset now considered relatively trivial, this problem is significantly harder. The two digits can be anywhere on the image, as opposed to centered for the MNIST dataset, and they can overlap, which prevents us from isolating the original MNIST images trivially. To solve this problem, we implemented three main machine learning techniques : logistic regression as a baseline linear method, a simple self implemented feed-forward Neural Network (NN), and different Convolutional Neural Network (CNN) architectures.

II. Related Work

CNNs have recently become the gold standard for achieving state of the art results in image classification tasks. They have consistently outperformed other techniques for the past few years (Simonyan & Zisserman [1]). Deep neural network architectures are able to learn arbitrarily complex functions of the inputs, at the price requiring a lot of data to properly train their large amount of parameters. CNNs try to compensate for a lack of data by injecting prior knowledge about the nature of images in the form of two main assumptions. Namely that pixel dependencies are local and that what is useful in one place will also be useful in other places (Krizhevsky & al. [3]).

Much has been written about how to design and train such architectures. The two main concerns are how to avoid overfitting and how to manage computation time. Regarding overfitting, Srivastava et al. [2] suggest randomly dropping units from layers in the neural network, along with their connections. This is reported to prevent neurons from co-adapting too much, and reduces overfitting better than other regularization techniques employed for CNNs. This technique has a cost in terms of convergence rate, Krizhevsky & al. [3] report that it doubles the number of iterations needed for the gradient descent to converge.

When designing CNNs, in practice a good heuristic is usually to "go as deep as your computation time allows" (A. Karpathy [4]). This brings up the computation time bottleneck, training very wide and deep networks with millions of parameters takes a long time even with parallel computing on multiple GPUs (the famous AlexNet took up to six days to train in 2012, Krizhevsky & al. [3]). Even though it would take only half an hour to train with the latest GPUs, we do not have access to this level of hardware and so computation time remains a major concern.

Using Rectified Linear Units (ReLU) in place of sigmoid or tanh activation functions for neurons has been reported to greatly improve the training time of CNNs (by a factor of six Krizhevsky et al. [3]). This boost in training time dallow the possibility of training a wider and deeper architecture better suited to large data sets.

Regarding the optimization techniques, deep neural networks, including CNNs, are usually trained using gradient descent optimization with mini batches. The two main algorithms used in practice are Nesterov's accelerated gradient, commonly referred to as Nesterov momentum [5], and the very recent Adam [6] algorithm based on the unpublished RMSprop algorithm [4].

III. Problem Representation

A. Data preprocessing methods

The images to be classified contained two white (pixel value of 255) digits on a background filled with noise. First we removed the noise by binarizing the image between the white digits and black background (pixel value of 0). We allowed pixels to remain white only if their value was 255 and there existed at least one pixel in the 8 pixel box around it whose value was greater than 245. Every pixel not satisfying these two conditions was set to zero. A example of this preprocessing can be seen in Figure 1 below :

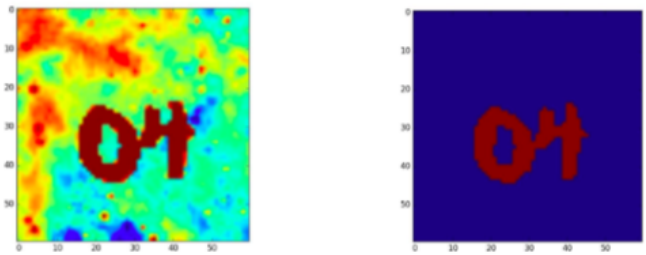

Figure 1: A sample image from our dataset before (left) and after (right) removing noise.

We experimented with two main preprocessing styles. The first, and simpler solution was be to work on the 60 by 60 images directly and classify them into one of 19 classes. The

second more complex solution was to isolate the two numbers and classify them individually.

Although it involves more work, the second scheme has many potential benefits. If we isolate two 28 by 28 images properly, we can take advantage of the MNIST dataset as additional training data. More importantly, as the MNIST dataset has received a great deal of attention, many models are known to have outstanding accuracy on it. As a final motivation for splitting the images, even if we could not take advantage of the MNIST dataset (the isolated images are not close enough to the originals) we could still used the separated digits we know to be correct to augment the dataset provided.

To separate these digits first we had to identify clusters of white pixels. This splits the problem into two clear cases. In the first case, there were two large, separate clusters. In this case separating the numbers was trivial and we will not expand on it. A sample result of this can be seen in figure 2 below:

Figure 2: Case 1 : before separation (left) and after (middle and right).

The second case, where there was only one distinct cluster, was far harder. A standard CNN was trained to identify images from the MNIST dataset prepossessed as above. Then a set of 35 predetermined cuts was applied to the unique large cluster to create 35 sets of two 28 by 28 pixel images, each representing a possible separation. Finally for each possible set, the two images were passed through a CNN trained to identify MNIST digit, and the sum of the prediction confidences was used to determine the best separation. A sample of this result can be seen in figure 3 below:

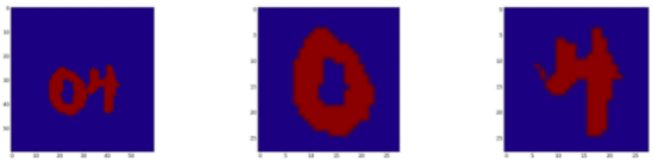

Figure 3: Case 2 : before separation (left) and after (middle and right).

B. Feature design and selection methods

For logistic regression, our main concern was to reduce the huge dimension space. In order to do so, we used the two separated digits extracted from the image, combined them into a single 56 by 28 pixel image (one digit above the other) and then reduced it to an 87 dimensional feature vector using principle component analysis (PCA).

For our implementation of a Neural Network, we used the two separated digits of size 28 by 28 pixels extracted from the original image. They were classified into one of 10 output classes and then their sum was computed. The motivation for this approach came from the fact that Neural Networks in the past have achieved high accuracy on the MNIST dataset [10]. Using this procedure quadrupled our accuracy on the validation set compared to using combined digits.

Due to the nature of Convolutional Neural Networks (invariant to translation), we used no special feature design or selection. We simply fed the original image with the noise removed as the feature vector to our model. Each of our 100000 examples had dimension (1,60,60), the first dimension being the "depth" of the image (since the image is black and white it is one, for a RGB image it would be 3).

C. Dataset Augmentation

The main concern when it came to achieve the best accuracy with our CNN was overfitting. We tried different schemes to counter this phenomenon, one of them was increasing the size of our training set. When training deep neural networks, it is standard procedure to craft additional data using label preserving transformations like image translations, horizontal reflections and patch extractions ([3], [4]). Neither reflections or patch extractions really make sense in our case as, for example, a 6 could end up looking like a a 9 or we could lose one of the two digits and then the label is not preserved anymore. But some kind of translation adapted to our problem could be performed.

In an attempt to produce more synthetic data on which to train, we used the correctly labeled separated 28 by 28 images and placed them at random on a 60 by 60 black pixel image. We did not use the MNIST data directly, as we did not know exactly how the dataset was produced in terms of the added noise, hence it could not be said with certainty that the distribution of the augmented data would match that of the original training data.

IV. Algorithm Selection and Implementation

A. Baseline : Logistic Regression

For logistic regression, we used SKLearn's library package [7]. We also used an SKlearn package for PCA. The C value (inverse of regularization strength) was set to 1, and L1 regularization was chosen over L2 regularization in an attempt to make the feature space even sparser.

B. Our Implementation : Neural Network

We implemented a fully connected feed-forward NN trained by back-propagation in Python. All hyper-parameters choices (number of hidden layers, number of nodes per layer, learning rate and regularization) were arrived at using a cross-validation set of 10,000 examples. With so many parameters to estimate, NNs can be very difficult to train. To guide our search we experimented with the architectures which were reported to achieve high accuracy on the MNIST dataset [11]. We finally settled on an architecture with two hidden layers with one hundred hidden units per layer for a total of 89,610 weights to be optimized. L2 regularization was employed to avoid overfitting.

C. Convolutional Neural Network Architectures

We started experimenting very early on the raw data (with noise removed) using a Python package called Nolearn, built on top of the Lasagne library, built on top of Theano. We started with a standard CNN architecture reported to perform well on the MNIST dataset (from blog post [8]) and adapted it to our problem of 60 by 60 pixel images. We then added one convolutional layer and one max pooling layer to improve performance. We will refer to this first architecture as architecture 1. It took the form :

[Input] → 3 * [conv 32*5*5 → maxpool 2*2] → [dense 256] → [Output (dense 19)]

All of our CNN architectures and hyper-parameter choices were tested using a 80% training 20% validation split of our original 100000 examples.

With a total of 8 layers and 188307 parameters to estimate, and without any regularization scheme we soon realized that this first model severely overfit the data. To adapt to this, we added a dropout layer before the first dense layer and this considerably increased validation accuracy, but approximately doubled the number of iterations needed for convergence.

Architecture 1 with one dropout layer took 25 epochs to converge at 10 minutes per epoch so around four hours to train in total. This is a relatively small training time for a CNN, hence we decided to follow A. Karpathy's [4] advice and go as deep as our computation time allowed.

We reasoned that two convolutional layers with filter size 3 by 3 stacked one after the other would outperform a single layer with filter size 5 by 5 (same reasoning as the authors of the VGGnet [9], also developped in A. Karpathy's notes [4]). Indeed, we get to inspect the same regions of the image (5 by 5 pixels), but we need to learn fewer parameters, which makes us less prone to overfitting. Moreover we get two non-linearities instead of one which makes the decision function more discriminative. The only downside was the increased memory cost for backpropagation and an increased computation time (not reported in the literature but significant in our case). Our new architecture, architecture 2, took the form :

[Input] → 3 * [[2 * conv 32*5*5] → maxpool 2*2] → [Dropout] → [dense 256] → [Output (dense 19)]

We saw an increase of 2% in validation accuracy by simply replacing every 5 by 5 convolutional layer by two 3 by 3 convolutional layers. But it also doubled the training time per epoch from 10 minutes to 20 minutes (about 8 hours to train in total).

To further combat overfitting we trained our model on the augmented data along side the supplied dataset. This will be expanded in the next section.

All experiments were performed with Nesterov momentum gradient descent, as it exhibited a faster convergence rate than all other method available.

V. Results

A. Logistic Regression

All hyper-parameter decisions were made using 5-fold cross-validation on a set of size 18000. The entire training set was never used as it became clear very quickly that it would never come close to results obtained with CNN architectures. When feeding our model the raw data, the accuracy was little more then always outputting the most common sum (approximately 10%). Figure 4 illustrates the accuracy obtained with various dimension reduction choices with PCA.

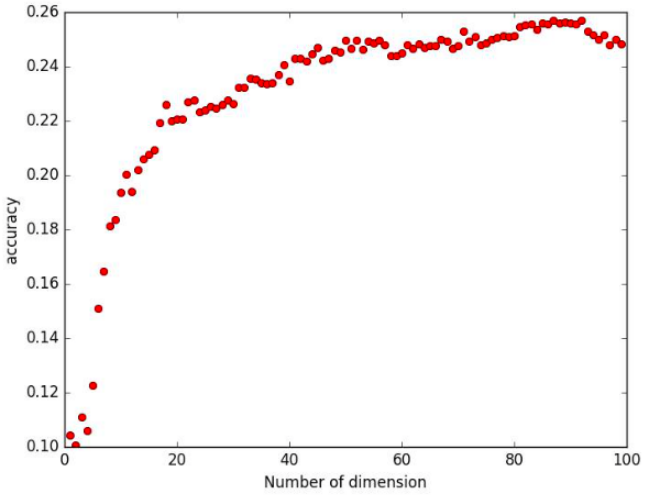

Figure 4: Accuracy as a function of the number of dimensions for PCA.

L1 regularization was chosen (25.99% accuracy), as it outperformed L2 regularization (25.86% accuracy) and no regularization at all (25.94% accuracy). The final model reached an accuracy of 25.99% with a feature vector size of 87 (as opposed to 3600 initial dimensions) and L1 regularization.

B. Neural Network

All hyper-parameter decisions were made using a separate cross-validation set of 10,000 examples. Cross-entropy was employed as the error function for for training the network and 0/1 loss was used to calculate the final accuracy. We settled on a learning rate α of 0.01 along with a momentum β of 0.9 (used in most implementations in the literature [12]). Values of α below 0.01 were too slow to train on (requiring close to 45,000 epochs). For values higher than 0.01 we were encountering oscillations in the training accuracy. The regularization parameter was set to 0.05, after testing multiple values.

The various values that we tried for the regularization parameter(λ) and their corresponding accuracy on the cross-validation set can be seen in Table I. We implemented multiple NN architectures, varying the number of hidden layers and hidden units. A list of these architectures and their corresponding accuracy can be found in Table II. As highlighted, the

architecture with 2 hidden layers of 100 units each resulted in the highest cross-validation accuracy of 40.28%.

Table I: Comparison of Different regularization parameter(λ) values

λ value	Cross-Validation set Accuracy
0.001	40.20%
0.003	40.22%
0.005	40.25%
0.007	40.25%
0.01	40.27%
0.03	40.27%
0.05	40.28%
0.07	40.26%
0.1	40.26%
0.13	40.26%
0.15	40.25%

Table II: Comparison of Different Architectures

#Hidden Layers	#Hidden Units	Cross-Validation set Accuracy
1	50	34.25%
1	100	38.24%
1	150	37.55%
2	100,50	38.01%
2	100,100	40.28%
2	100,150	39.51%
2	100,200	38.82%
3	100,100,50	36.24%
3	100,100,100	38.28%
3	100,100,150	35.82%
3	100,100,200	40.01%

We observed the model learned for roughly 1900 epochs before beginning to overfit the data. A comparison of training and cross-validation loss for one such instance of training along with the point of termination is shown in Figure 5: .

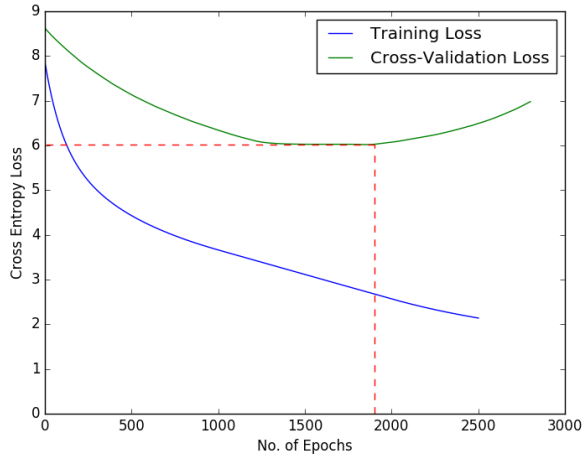

Figure 5: Cross entropy loss as a function of the number of training epochs.

C. Convolutional Neural Network Architectures

Figure 6 illustrates the results obtained with three variations of architecture 1 using Nesterov momentum. We chose to plot these three in particular as they illustrate the usual mechanics of training CNNs.

The first plot shows training with the original architecture (no dropout layer), a learning rate of 0.1 and a momentum of 0.9 (the default value in the literature). We can see training and validation error decrease very quickly (from 2.8 to 0.7 in only four epochs) which is typical of momentum based gradient descent. It can also be observed that both training and validation errors appear to plateau at 8 epochs.

The second plot was produced with the same settings but with a learning rate of 0.01 (ten times smaller). We can see that training error continues to decrease throughout the learning, while the validation error plateaus and begins to rise after 9 epochs. Here the learning rate is appropriate, but we are overfitting.

The third plot shows training with one added dropout layer. While the training takes almost twice as long to achieve a loss of 0.5, it continues to learn after this point and ultimately achieves a lower loss before beginning to overfit at around 22 epochs. This increase in training time can also be seen in figure 8 : the training time more than doubles from model A (architecture 1 without dropout layer) to model B (architecture 1 with dropout layer).

In terms of validation accuracy : the first experiment resulted in 85%, second in 88% and the third in 94%.

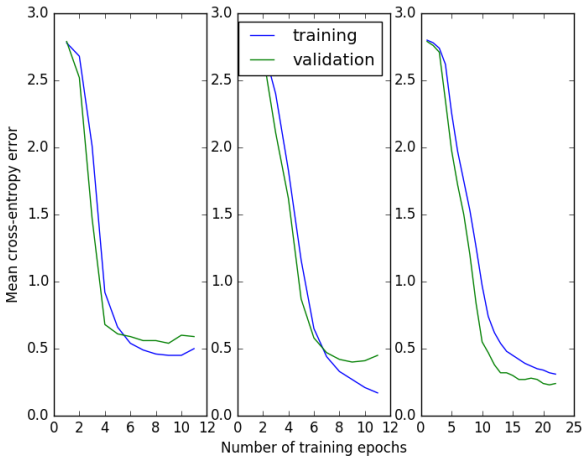

Figure 6: Cross entropy loss as a function of the number of training epochs for three different models.

Figure 7 compares the validation accuracy of architecture 1 (5 by 5 convolutional layer) trained on the original dataset to architecture 2 (two 3 by 3 convolutional layers) trained on both the original dataset and on the augmented dataset, as a function of the number of training epochs.

We can see the deeper architecture outperforms the first by 2% validation accuracy (96.1% vs 94.2%). This does not come for free as we can see in figure 8 that it takes approximately twice as long to train this deeper architecture (model C) compared to the original architecture (model B).

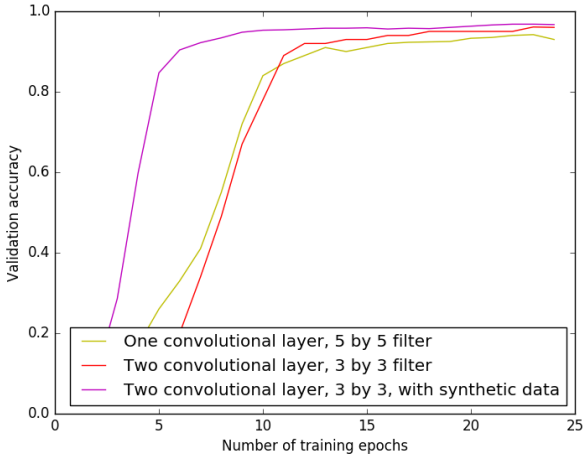

Figure 7: Validation accuracy as a function of the number of training epochs for architectures 1 and 2 described in the last section.

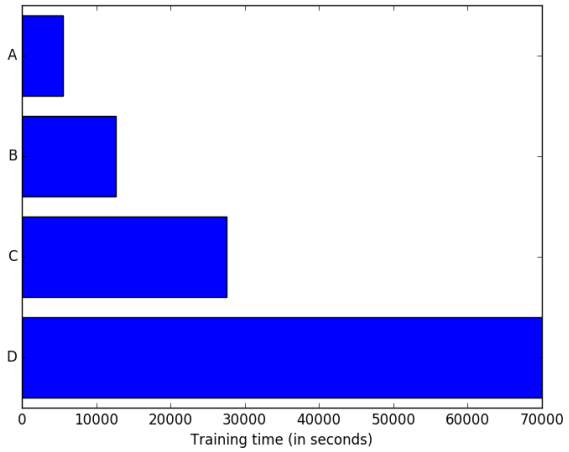

Figure 8: Training time (in seconds) for : A) architecture 1 without a dropout layer, B) architecture 1 with a dropout layer, C) architecture 2 with a dropout layer, D) architecture 2 with a dropout layer and augmented data.

Further changes in the training parameters such as altering the L2 weight decay, and adding additional dropout layers were not shown to improve performance. When the model could no longer improve performance though increased training time and could not be expanded due to constraints in training time, we turned to the idea of augmenting the data. We added 45000 synthetic examples, crafted as described before, to our training set and retrained our model from scratch. Each training epoch took about twice as long (c.f model D in figure 8 : about 70000 seconds or 20 hours to train in total). This addition to the training set allowed us to improve to 96.7% validation accuracy from the original 96.1% (also our final score on Kaggle : team name is Deewrnched).

VI. Discussion

A. Logistic Regression

Despite the relatively poor performance of 25.99%, we felt that we squeezed this technique for all the accuracy it could give. It seemed from the start that this method would not perform well at this task. The main benefit of this technique was the quick training time when compared to those of the two other techniques. The main disadvantage was its inability to learn the large degree of variation in the training set. In term of further improvements, it would have been interesting to examine how much of a performance boost increasing the 18000 images to the full training set would have had. This was not performed due to the long time separating images took.

B. Neural Network

Possibly some other gradient descent optimization techniques such as AdaGrad[12] and AdaDelta[12] can be used to slightly improve the performance. Yet it is highly unlikely that performance can be improved to match that of CNNs for this particular problem..

C. Convolutional Neural Network

Since both generating the augmented data and training the CNN took significant amounts of time (mainly because none of us had a GPU for the latter), we did not train the CNN with as much augmented data as we would have wanted to. Maybe doubling or tripling the original data would have allowed us to overfit later with our current architecture. In addition, we would have liked to train a deeper and wider network, however due to our hardware this was not possible.

VII. Statement of Contributions

A. Edward Smith

I was primarily responsible for the image preprocessing (noise removal and separation) implementation, the logistic regression implementation and analysis, and the creation of the augmented data.

B. Navin Mordani

I was primarily responsible for the design, implementation and analysis of the Neural Network and reporting the results for the same.

C. Theophile Gervet

My main responsibility was the design of the convolutional network architeture, and the design and analysis of the experiences conducted with it.

We hereby state that all the work presented in this report is that of the authors.

REFERENCES

[1] K.Simonyan, A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, ICLR, 2015.

[2] N Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, The Journal of Machine Learning Research, pages 1929âĂŞ1958, 2014.

[3] A. Krizhevsky, I. Sutskever, G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS, 2012.

[4] A. Karpathy, *Convolutional Neural Networks for Visual Recognition*, GitHub repository, http://cs231n.g ithub.io/neural-networks-3/, 2016.

[5] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton, *On the importance of initialization and momentum in deep learning*, ICML (3), pp. 1139-1147, 2013.

[6] Diederik P. Kingma, Jimmy Lei Ba, *Adam: A Method for Stochastic Optimization* ICLR, 2015.

[7] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E., *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, volume 12, pp 2825-2830, 2011.

[8] http://blog.christianperone.com/2015/08/convolutional-neural-networks-and-feature-extraction-with-python/

[9] Karen Simonyan and Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition* ICLR, 2015.

[10] LeCun,Cortes,Burges, *http://yann.lecun.com/exdb/mnist/*

[11] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, Juergen Schmidhuber, *Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition* Journal of Neural Computation, Volume 22, Number 12, December 2010.

[12] Sebastian Ruder, *An overview of gradient descent optimization algorithms* Internet: http://sebastianruder.com/optimizing-gradient descent/index.html#momentum.