©2016 Society for Imaging Science and Technology

Improving a deep convolutional neural network architecture for character recognition

Bogdan-Ionuţ CIRSTEA, Laurence LIKFORMAN-SULEM, Institut Mines-Télécom / Télécom ParisTech, Université Paris-Saclay, Paris, France

Abstract

Deep architectures based on convolutional neural networks have obtained state-of-the-art results for several recognition tasks. These architectures rely on a cascade of convolutional layers and activation functions. Beyond the set-up of the number of layers and the number of neurons in each layer, the choice of activation functions, training optimization algorithm and regularization procedure are of great importance. In this work we start from a deep convolutional architecture and we describe the effect of recent activation functions, optimization algorithms and regularization procedures when applied to the recognition of handwritten digits from the MNIST dataset. The network achieves a 0.38 % error rate, matching and slightly improving the best known performance of a single model trained without data augmentation at the time the experiments were performed.

Introduction

Recently, deep learning models have obtained state of the art results in tasks such as handwriting recognition [1] [2], speech recognition [3], object recognition [4] and machine translation [5]. In contrast to classical approaches, deep learning approaches learn features in an automatic fashion, avoiding the time-consuming task of hand-crafting features.

In this paper, we apply a deep learning approach to the recognition of isolated handwritten digits. The MNIST dataset [6] is popular for this task and a variety of approaches have been compared using it [6]. We will only focus here on those approaches which take into account and exploit information about the spatial structure of images and which use deep learning, while discarding the permutation-invariant MNIST task (in the permutation-invariant task, no prior information about the spatial arrangement of the input pixels is available). Many of the most recent and most successful proposals also use deep convolutional neural networks, as we do [7] [8] [9] [10] [11] [12]. Other approaches use recurrent neural network variants, such as the Multi-dimensional long short-term memory (MDLSTM) [13] or ReNet [14]. While these approaches encode information about the spatial structure of the images mainly through the architecture design, other approaches do so mainly through data augmentation, by using image pixels shift, scaling, or elastic distortion, in order to help the trained systems become more invariant to these transforms and to artificially augment the training set. The most notable of these approaches is [15], where the authors train big simple feedforward neural networks (also known as multilayer perceptrons - MLPs) using elastic distortions. As is true for many other tasks, training ensembles of classifiers often results in better performance than training a single classifier [12] [15] [16].

We focus in this paper on a single classifier, with no data augmentation. Thus recognition results can be more easily compared with others, since they do not depend on the amount and the type of augmented data.

As the number of approaches is relatively large and the number of deep learning architectures for this task is potentially huge, we will focus on an efficient convolutional architecture which uses recently-proposed activation functions (Rectified Linear Units - ReLU and its variants Leaky ReLU and Parameterized ReLU). We then train the proposed architecture with recently introduced initialization and optimization procedures. The initialization procedure is designed to avoid reducing or magnifying the amplitudes of the input signals to each activation function in each layer exponentially, as well as the amplitudes of the gradients during the training procedure. The optimization algorithm is a variant of Adam [17] which is trained on mini-batches and uses annealed learning rates. It is designed to stabilize the learning of stochastic non-stationary objective (loss) functions and to better handle sparse and unstable (vanishing or exploding) gradients [18]. We also use several regularization methods, which allow us to build bigger CNNs without overfitting (dropout [19]) and accelerate the convergence of the optimization algorithm by normalizing the inputs to the CNN's different layers and improving the gradient flow through the network (batch normalization [20]). We apply our architecture and training process to the recognition of isolated handwritten digits. To the best of our knowledge, we have obtained the best recorded performance of a single system without data augmentation on the MNIST dataset [6] at the time the experiments were performed.

Compared to other approaches used in the past, our system performs all of its feature extraction in an automatic fashion, and the feature extractors are learned. We argue that this result, in addition to other results obtained using deep learning systems, suggests that automatically learned features may perform better than handcrafted ones. Using a deep learning approach can also allow handwriting researchers to import innovations from other fields (object recognition, speech recognition) and also potentially contribute to the advancement of those fields, since deep learning approaches working well in one domain can often perform well in other domains, too.

We present in Section Architecture the proposed architecture based on a cascade of convolutional layers. We then describe in Section Optimization the optimization steps used for minimizing the loss function in order to train the system based on the previously mentioned architecture. Regularization approaches such as dropout and batch normalization are presented in Section Regularization. We apply this approach to the recognition of the MNIST digit dataset in Section Experiments and present our results with the proposed architecture.

Architecture

In this section, we describe the main architectural choices we have made and their motivation, as well as the neuron activation functions we have used.

Architectural choices

A convolutional neural networks (CNN) is a neural network architecture which incorporates knowledge about the invariances of 2D shapes by using local connections patterns and tied weights [21].

In CNNs, neurons are organized in planes, with each unit in a plane sharing a set of weights and performing the exact same operation on different parts of the image. Each such plane is called a feature (activation) map. Each unit (neuron) in a feature map is connected to a certain area of its input ; this area is called a receptive field and the trained weights are called convolutional filters.

Input size	**Convolutional Layer 1**
1 x 28 x 28	conv : 3 x 3 full, stride 1, 32 feature maps
32 x 30 x 30	batch normalization
32 x 30 x 30	PReLU
32 x 30 x 30	2 x 2 max pooling, stride 2
32 x 15 x 15	0.5 dropout
	Convolutional Layer 2
32 x 15 x 15	conv : 3 x 3, stride 1, 64 feature maps
64 x 15 x 15	batch normalization
64 x 15 x 15	PReLU
64 x 15 x 15	2 x 2 max pooling, stride 2
64 x 7 x 7	0.5 dropout
	Convolutional Layer 3
64 x 7 x 7	conv : 3 x 3, stride 1, 128 feature maps
128 x 7 x 7	batch normalization
128 x 7 x 7	PReLU
128 x 7 x 7	2 x 2 max pooling, stride 2
128 x 3 x 3	flatten
128 x 3 x 3	0.5 dropout
	Fully Connected Layer
128 x 3 x 3	fully connected layer matrix multiplication
625	batch normalization
625	PReLU
625	0.5 dropout
	Softmax Layer
10	softmax

Table 1. Proposed CNN architecture

The convolutional architecture we use follows many of the guidelines from [22], which has obtained the best single-model performance in the ILSVRC 2014 object classification competition [23]. Each of the convolutional layers has filters of size 3x3, with stride 1 (the stride is the distance between the centers of the receptive fields of neighboring neurons in an activation map). The small window size allows us to capture fine details ; note also that 3x3 is the minimum filter size so that concepts such as left, right, bottom, up can still be captured. 'Full' mode convolution is only performed in the first layer. It adds surrounding zero-valued pixels to the image, so that we can apply the convolutional filters on the points at the border of the input image (before padding). Thus input images of size 28x28 are zero-padded to reach size 30x30 (this is necessary because we use 3x3 convolution filters) and the resulting output images are also of size 30x30.

The subsampling layers are always 2x2 max-pooling with stride 2, which results in subsampling both image height and width by 2. 2x2 max-pooling with stride 2 is a very popular pooling method, because it is fast, it quickly reduces the size of the hidden layers (benefiting computational efficiency) and it promotes some invariance with respect to translations and elastic distortions [9].

When choosing the number of neurons in the convolutional layers, we are once more inspired by the guidelines outlined in [22] : we double the number of neurons between each consecutive layer, from 32 in the first convolutional layer to 64 then 128 in the last one. The intuition behind this way of building the architecture is that we are trying to compensate for some of the information loss resulting from the subsampling performed by the max-pooling layers by using more units (activation maps) in the upper hidden layers. As a result of both the max-pooling subsampling and succeeding convolution operations, units in the higher layers have a bigger implicit receptive field with regard to the input image. A 3x3 receptive field on a subsampled image corresponds to a bigger receptive field applied to the unmodified input image.

Activation functions

Activation functions have been an important part of the success of supervised deep learning. One of the most successful activation functions is the Rectified linear unit (ReLU) [24] :

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Leaky rectified linear unit (LReLU) [25] have been found to either match or surpass ReLU performance [25] [26] :

$$\text{LReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ ax, & \text{otherwise.} \end{cases} \tag{2}$$

with a the scaling factor (real value). In [25] the authors suggest choosing a small a value, such as $1/100$. The authors of [26], on the other hand, have obtained better empirical results using large a values, such as $1/5.5$.

[26] argues for the use of Randomized leaky Rectified Linear Units (RReLU) to further improve the performance of LReLU, based on empirical evidence :

$$\text{RReLU}(x) = \begin{cases} x, & \text{if } x > 0. \\ \frac{1}{a}x, & \text{otherwise with } \quad a \sim \text{U}(l,u). \end{cases} \tag{3}$$

Here, during training, a is sampled from a uniform distribution, with parameters l and u (the authors suggest setting $l = 3$, $u = 8$, based on empirical evidence). At validation / test time, the value of a is fixed, with value $\frac{l+u}{2}$ (5.5 for the example above). Notice that l and u are hyper-parameters here (their gradients are not used to modify their values during training).

Parametrized Rectified Linear Units (PReLU) [27] is another rectified activation function, with the same equation as for LReLU. The difference, though, is that a is learned using its gradient. A different a can be chosen for each neuron or the a values can be 'tied' so that several neurons share a same value. This helps reduce the number of trainable parameters and, thus, can prevent overfitting. We have obtained the best results by allowing each neuron to have its own a value in the convolutional part of the network. For the fully connected layers, we could use a single a value for each fully connected layer or distinct a parameters for each neuron in each fully connected layer ; in our best-performing system, separate a values are used for each neuron.

Optimization

In this section we describe the loss function we optimize, as well as the optimization algorithm which modifies our system's trainable parameters in order to minimize it.

Loss function

The loss function we minimize is the average negative log-likelihood of the conditional distribution of the correct label given the input $\log(p(y|x))$ across training examples (x, y) from dataset S. This is equivalent to maximizing the multinomial logistic regression objective. We have also used weight decay (L2 regularization), the corresponding term in the equation below being $\alpha * ||w||^2$. Further details about the optimal value for the α hyper-parameter are provided in the Weight decay subsection.

The equation of the loss function is thus :

$$O(S) = \sum_{(x,y)\in S} [-\log(p(y|x) + \alpha * ||w||^2] \quad (4)$$

Mini-batch stochastic gradient descent(SGD)

Gradient descent is a general method for optimizing differentiable functions using the gradients of the loss function with regard to the function's parameters ; it can be efficiently applied to neural networks using the backpropagation algorithm [28].

When training with backpropagation using the training dataset, one can either use all the training samples for each parameter update (full batch gradient descent) or a single sample (stochastic gradient descent - SGD). A compromise between the two is mini-batch SGD, which consists in performing parameter updates after seeing a fixed number of training samples. The samples in each mini-batch are chosen in a stochastic manner (by e.g. shuffling the dataset).

One motivation for using mini-batches is that, as the mini-batch size increases, the estimate of the gradient improves. Furthermore, computation over mini-batches can be much more efficient than over single samples, especially when using modern computational architectures and even more so for GPUs than for CPUs.

Initialization

Initialization is important because bad initialization can lead to instability in the gradients (vanishing or exploding gradients) and consequently poor model performance after training. In contrast, good initialization can accelerate convergence speed.

We have obtained the best results using the initialization method described in [27]. For simplicity, we initialize the PReLU a parameters with zero values (right after this initialization, PReLUs behave just like ReLUs). We initialize the batch normalization γ parameters to 1, and the β parameters to 0 (see Batch normalization). The neuron weight matrices are initialized using values drawn from Gaussian distributions with mean 0 and standard deviation $\sqrt{\frac{2}{n_i}}$, where n_i is the number of inputs to the neuron, following the procedure in [27], and the biases are initialized to 0.

This procedure helps avoid both the exponential vanishing and exploding of gradients, as well as of inputs to each layer during the feedforward phase. We refer you to [27] for the full mathematical treatment.

Adam variant

Algorithm 1 Adam (variant)

Legend :
t = timestep
θ_t = neural network parameter values at timestep t
lr = learning rate
$decay$ = learning rate decay
$init$ = initial learning rate value
ε = hyper-parameter to make the algorithm numerically stable

Algorithm :
$lr = init$
while θ_t not converged do :
  g_t = gradient of loss function with regard to θ_t
  m_t = moving average of g_t
  v_t = moving average of g_t^2
  $\theta_t = \theta_{t-1} - lr * \frac{m_t}{\sqrt{v_t}+\varepsilon}$
  $lr = lr * decay$

Adam [17] is an optimization algorithm which has been developed for improving the performance of stochastic gradient descent (SGD) on stochastic non-stationary loss functions and sparse gradients.

In the case of training convolutional neural networks, the stochasticity comes from mini-batch training and batch normalization, from dropout, as well as from the inherent noise in the function to be learned (which takes as inputs images and outputs labels). Adam is also claimed [17] to allow for less hyper-parameter tuning. When using simpler optimization procedures such as SGD, different learning rates have to be chosen for different CNN layers, leading to more hyper-parameters which need to be set ; [17] claims that this is not necessary when using Adam. On the other hand, the algorithm introduces a few additional hyper-parameters (ε and a few other parameters described in detail in [17]). We briefly describe the algorithm as we have used it in Adam (variant). A detailed description of Adam is beyond the scope of this paper ; we refer the reader to the relevant publication [17].

One aspect we have found empirically is that the learning schedule (the way the learning rate is adjusted during the optimization procedure) is critical in determining both the speed of learning (the number of epochs required), as well as the final performance (as measured by the validation accuracy). In spite of the fact that the authors of [17] claim that this optimization method is quite robust to step size, we have found that adapting the learning rate during learning still helps. We thus propose a variant of Adam which consists in using annealing (multiplying the learning rate by a constant after each epoch of training). This corresponds to the last equation in Adam (variant).

The value of the learning rates obtained using the annealed learning rate procedure are quite high (see subsection Methodology). This is in line with other empirical observations for choosing the learning rate when using dropout and batch normalization [19] [20]. When using dropout, each sub-sampled architecture is trained for only one step, but all the possible sub-sampled architectures share parameters, so each update must have a large effect for the training procedure to behave as if it is training an ensemble rather than a single model [10]. Using higher learning rates during the optimization procedure corresponds to the updates having larger effects.

An intuitive explanation for why annealing the learning rate works well is that, as the learning rate decays, the optimization takes shorter steps, doing less exploration, so that it eventually settles into a local minimum [19].

Regularization

Regularization is an important component of many successful machine learning systems. Deep learning systems routinely use several different regularization methods, such as dropout [19], L2 regularization (weight decay) and early stopping. Besides these now somewhat standard regularizers, we have also used batch normalization, a method which has been proposed only recently but has been very successful in training deep convolutional neural networks [20].

Dropout

Dropout [19] is a regularization method which allows for much bigger networks to be trained without overfitting. The basic idea is to randomly drop neurons from the neural network during training with probability p sampled from a Bernoulli distribution. The result of this procedure is that this training approximates training an ensemble of much thinner neural networks. During validation or testing, an approximation of using the ensemble of thinned networks can be obtained by scaling the activations by the same probability p.

Batch normalization

Batch normalization [20] is a regularization method which allows for much faster training of neural networks because it allows for slightly higher learning rates and converges in fewer epochs. Batch normalization addresses the so-called problem of internal covariate shift [20], which can be defined as the changing (shifting) of the input distribution when training a classifier. A deep neural network can be seen as a composition of stacked classifiers, each taking as input the output of previous layers. During training, as we modify the parameters of the lower layers, the distribution of the input to the higher layers changes. Batch nor-

Algorithm 2 Dropout

Legend :
l = layer index
x^l = input of layer l
y^l = output of layer l
p = dropout probability
Equations without dropout during training :
$y^l = W^l * x^l + b^l$
$x^{l+1} = f(y^l)$
Equations with dropout during training :
$r^l_p \sim Bernoulli(p)$
$\tilde{x}^l = r^l \cdot x^l$
$y^l = W^l * \tilde{x}^l + b^l$
$x^{l+1} = f(y^l)$
Equations without dropout during validation / test :
$y^l = W^l * x^l + b^l$
$x^{l+1} = f(y^l)$
Equations with dropout during validation / test :
$y^l = p * (W^l * x^l + b^l)$
$x^{l+1} = f(y^l)$

Algorithm 3 Batch normalization

Legend :
n = number of examples in the mini-batch
i = index of the example in the mini-batch
μ = mean value of the examples in the mini-batch
σ^2 = variance of the examples in the mini-batch
ε = hyper-parameter to make the algorithm numerically stable
β, γ = learned parameters to restore the layer's representational power
Algorithm :
$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$
$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$
$\tilde{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
$BN_{\gamma,\beta}(x_i) = \gamma \cdot \tilde{x}_i + \beta$

malization works by standardizing the inputs between each layer of a neural network (to mean 0 and variance 1) during training, while also adding a few trainable parameters so that the original inputs can be recovered and the trained system does not lose any representational power. In doing so, it also improves the gradient flow through the network, by reducing the dependence of gradients on the scale of the parameters or of their initial values [20].

We apply batch normalization to every training mini-batch and we shuffle the training set, as recommended in [20]. Training with batch normalization leads to each example being seen in conjunction with other examples in the mini-batch, so that the training procedure no longer produces deterministic values for a given training example. This can be seen as having a regularizing effect similar to using dropout. Applied naively, batch normalization only adds two extra parameters (γ and β) for each neuron, so the risk of over-fitting as a result of an increase in the number of parameters is low. As a result of adding the batch normalization β parameters, the neuron bias parameters are no longer necessary, so we no longer use them in the layers in which we use batch normalization ; therefore, the effective number of parameters only increases by 1 per neuron. When using the validation set, we no longer split it into mini-batches, but use it in its entirety. This has the advantage of removing the randomization introduced by batch normalization (when using mini-batches) when estimating the system's accuracy on the validation set. The same procedure is applied when testing the system's final performance (on the test set).

Weight decay

We have tried L2 regularization (also called weight decay when applied to neural network weights), but found that it did not improve our results ; therefore, the α hyper-parameter was set to 0. An explanation of why we don't need L2 regularization could be that dropout has actually been found to be an improved form of L2 regularization [29] and also leads to sparser hidden unit activities, similarly to the effect of L1 regularization. Another hint comes from the fact that in [20] it has been found empirically that the L2 regularization can be diminished when using batch normalization.

Early stopping

We have used early stopping when performing optimization : we run the optimization algorithm for a maximum 300 epochs (maximum 300 passes through the entire dataset using mini-batches) and stop after 30 epochs without improvement on the validation set. Another method we have tried, but which has worked worse empirically, was to go through a fixed set of learning rates (e.g 0.1, 0.01, 0.001) in decreasing order, by decreasing the learning rate when the validation accuracy doesn't improve after a given number of epochs (e.g. 30). Note that we do not retrain the model using both the training and validation samples.

Experiments

Dataset

MNIST [6] contains 28x28 images of isolated handwritten digits, the task being to correctly classify these images. The data is split in 3 sets : a training set of 50000 images, a validation set of 10000 images and a test set of 10000 examples.

Preprocessing

As is standard in the literature ([7], [11]), the only preprocessing we perform is scaling the inputs to [0, 1] values. We have also tried standardizing the inputs using global contrast normalization, but haven't seen improved results.

Methodology

Our experiments start from the architecture proposed in [30], using the theano framework [31] [32]. During the experiments, we have kept the deep CNN architecture relatively unchanged but have modified activation functions, optimization algorithms and regularization methods. The specific architecture, as well as the activation functions and specific optimization and regularization procedures are presented in Table 1 1.

We have also experimented with using an architecture variant in which a second fully connected layer of the same size (625 neurons) was added, while keeping the rest of the initial architecture the same ; however, we have not observed improved results.

For the optimization procedure, we have used Adam, keeping all its hyper-parameters to their default values in [17], except for the learning rate, which was annealed during training. We obtain our best system using an initial learning rate of 0.005 and a learning rate decay of 0.98.

As regularization, in the best system we have used dropout [19] with probability of removing a neuron $p = 0.5$, in both the convolutional and the fully connected layers (we don't apply dropout to the input images). Though one might expect that overfitting is not a problem for convolutional layers, since they have few parameters (as a result of the local connection patterns and the tied weights), dropout can still help by providing noisy inputs to the higher fully connected layers, preventing them from overfitting [19]. We have also tried values of $p = 0$ (corresponding to no dropout), $p = 0.2$ and $p = 0.8$.

Notice that we haven't performed a systematic, extensive hyper-parameter search, due to limitations in access to computation (GPUs). Using random search [33] or Bayesian optimization [34] might lead to improved results, as has been observed empirically in [34].

Model	Test error (%)
Ours	0.38
Deeply-Supervised Nets [7]	0.39
Fractional max-pooling [9]	0.44
Maxout Networks [10]	0.45
Network in Network [11]	0.47

Table 2. Comparison with state-of-the-art results on MNIST test set (single system, no data augmentation)

Results

The error rate obtained with the system described in Section is shown in Table 2 2. The same table also shows results of other deep learning approaches which obtain state-of-the-art results on

MNIST without using data augmentation and model ensembles. The misclassified digits are shown in Fig. 1 1.

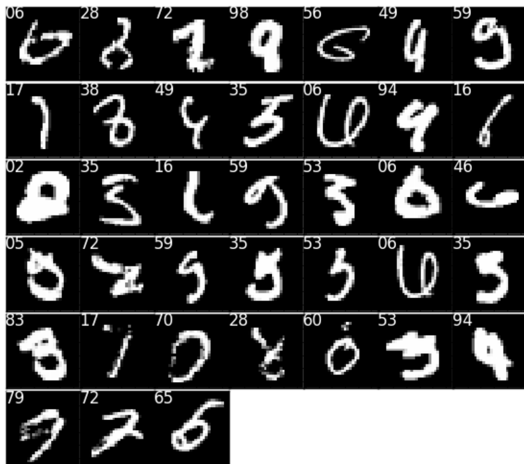

FIGURE 1. *All misclassified samples of the MNIST test set. The first number is the estimated label, the second one is the ground truth.*

This result matches and slightly improves, to the best of our awareness, upon the best previous result for a single model trained without data augmentation at the time the experiments were performed [7] [8].

We have computed the Wald confidence interval with 95 % confidence ; rounded to two decimals, we obtain an upper bound of the error rate of 0.5 % and a lower bound of 0.26 %. We have also performed 12 new experiments using different random seeds. The best test set error rate we obtain is 0.38 %, while the worst is 0.49 %. The mean error rate over these random sampling experiments is 0.43 % with 0.029 % standard deviation. Notice that while the 0.38 % error rate estimate provided above seems rather optimistic, we have managed to reproduce that result using a different random seed (the random seed affects every part of the system which involves stochasticity : the parameter initializations, dropout, as well as batch normalization with mini-batches).

We present below a more thorough comparison between our deep CNN and other deep CNN architectures which obtain state of the art results on MNIST without using data augmentation and model ensembles.

In [11], the authors use a modified CNN architecture where the convolutional layers contain micro neural networks instead of simple linear filters followed by nonlinearities. This enhances the discriminability for local patches within the receptive fields and allows them to replace the fully connected layers with global average pooling. Dropout and SGD with momentum are used.

Authors in [10] use a Maxout activation function, which is similar to ReLU, but designed specifically to work well with dropout. This activation function is used in both the convolutional and the fully connected layers.

[7] use companion loss functions for training the hidden layers of their deep convolutional neural networks, in order to better deal with vanishing gradients [18]. The CNN they train isn't as deep ; they use bigger filter sizes for the convolutional layers, the ReLU activation function and different loss functions for both the main and the companion objectives (the hinge loss). They also use 0.5 dropout for regularization and SGD with fixed momentum for optimization.

[9] introduce a stochastic form of spatial pooling, which allows them to reduce the spatial size of the activation maps more gradually. They use a modified, spatially-sparse CNN [35], with LReLU and dropout.

Notice that the use of micro neural networks and companion objective functions for the hidden layers are complementary to our approach and could be used to further enhance performance.

Conclusions

In this paper we have argued for engineering systems which can perform automatic (learned) feature extraction, rather than using hand-engineered features. Using a deep learning approach also allows for deep learning innovations from other applications domains to be imported and applied with ease and the deep learning contributions can also be used in other application domains.

One possible extension of this work is training the system on more extensive handwriting datasets, such as NIST SD-19, as well as using data augmentation and training ensembles. The robustness of the results we have obtained could be improved by choosing the hyper-parameters in a more automatic fashion, using random search [33] or Bayesian optimization [34].

Another possible direction for future research is integrating convolutional neural networks into text-line handwriting recognition systems. Though, historically, convolutional neural networks have played an important role in recognizing digit or character sequences [21], more recently, they have been replaced with Multidimensional Long Short-term (MDLSTM) neural networks [1] [2].

Acknowledgments

This research was supported by a DGA-MRIS scholarship. We wish to thank the Paris-Saclay Center for Data Science for access to the GPU platform which was used for this work, platform funded by the P2IO LabEx (ANR-10-LABX-0038) in the framework « Investissements d'Avenir » (ANR-11-IDEX-0003-01) managed by the French National Research Agency (ANR). We also thank NVIDIA Corporation for a GPU donation.

Références

[1] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 545–552, 2009.

[2] Theodore Bluche, Hermann Ney, and Christopher Kermorvant. The limsi handwriting recognition system for the htrts 2014 contest. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 86–90. IEEE, 8 2015.

[3] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE, 5 2013.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

©2016 Society for Imaging Science and Technology

[5] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.

[6] Christopher J.C. Burges Yann LeCun, Corinna Cortes. THE MNIST DATABASE of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Online ; accessed 30-November-2015].

[7] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. 9 2014.

[8] Julien Mairal, Piotr Koniusz, Zaïd Harchaoui, and Cordelia Schmid. Convolutional kernel networks. In *NIPS*, pages 2627–2635, 2014.

[9] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6, 2014.

[10] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. pages 1319–1327, 2 2013.

[11] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. page 10, 12 2013.

[12] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649. IEEE, 6 2012.

[13] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, 2012.

[14] Francesco Visin, Kyle Kastner, Kyunghyun Cho, Matteo Matteucci, Aaron C. Courville, and Yoshua Bengio. Renet : A recurrent neural network based alternative to convolutional networks. *CoRR*, abs/1505.0, 2015.

[15] Dan C. Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Handwritten digit recognition with a committee of deep neural nets on gpus. *CoRR*, abs/1103.4, 2011.

[16] Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *ICML (3)*, pages 1058–1066, 2013.

[17] Diederik Kingma and Jimmy Ba. Adam : A method for stochastic optimization. 12 2014.

[18] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318, 2013.

[19] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1) :1929–1958, 2014.

[20] Sergey Ioffe and Christian Szegedy. Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, pages 448–456, 2015.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11) :2278–2324, 0 1998.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 9 2014.

[23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3) :211–252, 4 2015.

[24] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[25] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. *Proc. ICML*, 30, 2013.

[26] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. 5 2015.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers : Surpassing human-level performance on imagenet classification. 2 2015.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. pages 318–362, 1 1986.

[29] Stefan Wager, Sida I. Wang, and Percy Liang. Dropout training as adaptive regularization. In *NIPS*, pages 351–359, 2013.

[30] `https://github.com/Newmu/Theano-Tutorials/blob/master/5_convolutional_net.py`. [Online ; accessed 30-November-2015].

[31] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano : a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[32] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano : new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[33] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[34] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[35] Benjamin Graham. Spatially-sparse convolutional neural networks. *CoRR*, abs/1409.6, 2014.