Convolutional Deep Rectifier Neural Nets for Phone Recognition

László Tóth

Research Group on Artificial Intelligence
Hungarian Academy of Sciences and University of Szeged, Hungary
tothl@inf.u-szeged.hu

Abstract

Rectifier neurons differ from standard ones only in that the sigmoid activation function is replaced by the rectifier function, $max(0, x)$. Several recent studies suggest that rectifier units may be more suitable building units for deep nets. For example, we found that with deep rectifier networks one can attain a similar speech recognition performance than that with sigmoid nets, but without the need for the time-consuming pre-training procedure. Here, we extend the previous results by modifying the rectifier network so that it has a convolutional structure. As convolutional networks are inherently deep, rectifier neurons seem to be an ideal choice as their building units. Indeed, on the TIMIT phone recognition task we report a 6% relative error reduction compared to our earlier results, giving an 18.6% error rate on the core test set. Then, with the application of the recently proposed 'dropout' training method we reduce the error rate further to 17.8%, which, to our knowledge, is the best result to date on this database.

Index Terms: Deep neural networks, sparse rectifier neural networks, phone recognition

1. Introduction

Recently there has been a renewed interest in neural networks for speech recognition, thanks to the invention of deep neural nets [1]. This technology brought significant improvements over the standard shallow network structures applied earlier, and deep nets are now being applied even for large vocabulary tasks [2, 3, 4]. The main weakness of deep networks is that they require a time-consuming pre-training for optimal performance. Some methods have already been proposed to circumvent this. For example, Seide et al. constructed a layer-wise backpropagation training strategy [4], while Plahl et al. proposed a different pre-training method based on sparse encoding symmetric machines [5]. Our team recommended yet another solution based on rectifier neurons [6]. Compared to conventional neural nets, rectifier neural networks differ only in the type of activation function used. However, this slight modification seems to enable them to learn deep structures more efficiently than standard neural nets. Using deep rectifier networks, we were able to achieve similar results on the TIMIT database than those attained with the pre-trained nets of Mohamed et al. [1], but without the need for any pre-training [6]. Here we go one step further, and refine the structure of the rectifier network so that is has convolutional layers in its lower part. The concept of convolutional neural networks is quite popular in image processing [7], but so far only sporadic efforts have been made to apply them to speech recognition. The most sophisticated of these is the construct by Abdel-Hamid et al., who apply tricks like limited weight sharing and max pooling [8]. The network studied here is simpler in structure, and is more analogous to the work of Veselý et al. [9], but it is built out of rectifier neurons instead of sigmoid neurons. As a convolutional network must inherently consist of many layers, and rectifier units proved to be more suitable for such deep structures, we expect a superior performance from building the convolutional model out of rectifier units. Here, we test this assumption on the classic TIMIT phone recognition task.

This publication was supported by the European Union and co-funded by the European Social Fund. Project title "Telemedicine-focused research activities in the fields of mathematics, informatics and medical sciences", project number: TÁMOP-4.2.2.A-11/1/KONV-2012-0073.

2. Convolutional Deep Rectifier Neural Nets

Rectifier neural nets differ from conventional neural nets in only one fundamental respect: the type of the activation function used. Instead of the usual sigmoid activation function, they apply the rectifier function $max(0, x)$ for all hidden neurons. The effect of this change on the behavior of the network was analyzed by Glorot et al., and they also gave nice experimental results for image recognition and NLP tasks [10]. Further motivation for the use of rectifier units was provided by Nair et al., who successfully applied them to improve restricted Boltzmann machines [11]. We experimented with phone recognition on the TIMIT database, and found that deep rectifier neural nets could achieve the same recognition accuracy as the deep sigmoid nets presented in [1], but their training was simpler and faster [6].

Existing neural network code can be easily modified so that it can handle rectifier units. Besides the replacement of the sigmoid activations by the rectifier function, the activation derivatives also have to be modified. Also, we found that the application of weight decay or weight normalization is recommended in order to keep the weights within reasonable limits [6]. These modifications are, however, very simple, and in all other respects a rectifier net operates exactly the same way as a standard perceptron-based network. In particular, it can be trained using standard backpropagation training. The target function can be the usual cross-entropy error, but the results may be improved further by augmenting it with a penalty term that enforces sparsity. For this purpose we found the function $\sum log(1 + a_j^2)$ proposed by Sivaram [12] to work nicely, where the sum is over all rectifier unit outputs a_j.

In this paper, we use the same backpropagation training procedure as that described in [6], and refine only the architecture of the network. Fig. 1 shows the schematics of the network we are going to apply here. Let us first consider the network composed of the blocks of neurons denoted by shaded boxes. If the network consisted of only these components, then we would have a standard non-convolutional deep network with four neural layers and an input layer consisting of seven consecutive

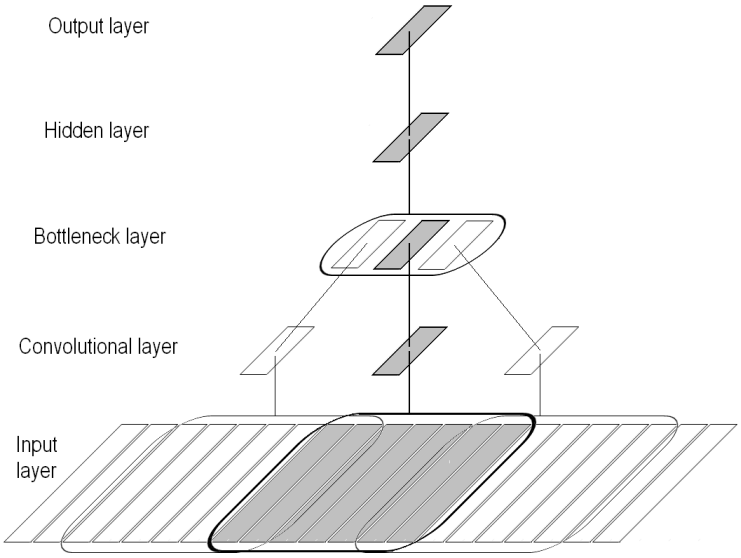

Figure 1: *Schematic diagram of the convolutional network applied here. The blocks of neurons denoted by blank boxes use the same weights as the shaded layer does in the same row. For clarity, full connections between layers are denoted by a single line only.*

feature vectors. What makes the network convolutional is the additional blocks of units denoted by blank boxes. These are left blank so as to emphasize the fact that they do not have their own weights, but they use the same weights as the corresponding shaded layer does in the same row. Practically speaking, this means that the convolutional layer gets evaluated on several blocks of input. This way the input context of the network can be extended considerably without significantly increasing the number of weights. For example, in Fig. 1 the convolutional layer processes three blocks of local input, altogether covering a span of 17 feature vectors. A further important property is that these blocks may be positioned several frames away, so the convolutional part of the network inherently performs subsampling as well. Here we will refer to the local block of input vectors as the 'local context', while the full span of the input to the network will be called the 'convolutional context'.

The layer simply called the 'hidden layer' in the figure operates on several blocks of output from the preceding layer. This enables the network to fuse the pieces of information coming from the local context windows. The price for this is that for this layer the number of inputs is much larger than that for the rest of the layers; and then the number of weights required is also bigger. If we want to keep the number of weights similar to those of the other layers, then we have to reduce the input size. Because of this reduced size, the input to this layer is called the 'bottleneck layer' in the figure. For example, if the subsequent layer uses five blocks of output from the bottleneck layer, then the bottleneck layer should be a fifth of the size of the other layers. With this restriction the convolutional net does not require more parameters than a conventional net, but it is able to process a much bigger span of input vectors.

The convolutional network is trained using semi-batch backpropagation, exactly the same way as its non-convolutional counterpart. The only step that might need explanation is how the error is propagated through the bottleneck. When propagating the error backwards from the hidden layer, we get error values for both the shaded and the blank blocks of neurons in the bottleneck layer. However, the blank blocks do not have independent weights, but are just replicas of the units denoted by the shaded box. Hence, the errors attributed to these replica units should all be propagated to the units of the shaded box, which we solve here by taking the average of the errors. Note that other strategies are possible as well, for example Abdel-Hamid et al. use max pooling layers to combine the results of convolutional layers [8]. Of course, this requires modifications to the error backpropagation scheme as well.

Finally, it should be mentioned that the convolutional structure shown in Fig. 1 is a relatively 'shallow' one. We will experiment with networks that have more layers, and hope that with the use of rectifier units we will be able to efficiently train these deeper structures as well. In these experiments we will simply refer to the layers up to the bottleneck layer as the 'lower part', and the layers above it as the 'upper part' of the network.

3. Experimental Settings

The results reported are phone recognition error rates on the well-known TIMIT database. The training set consisted of the standard 3696 'si' and 'sx' sentences, while testing was performed on the core test set (192 sentences). A random 10% of the training set was held out for validation purposes, and for tuning the meta-parameters. We will refer to this block of the data as the 'development set'. All the experiments used a phone bigram language model estimated from the training data. To get frame-level labels for the training data, a conventional HMM was trained (using HTK), and then a forced alignment was performed with it. We worked with context-dependent (CD) phone models, which were obtained with the decision tree-based state clustering tool of HTK, and resulted in 858 tied states. These were derived from the 61 phone labels, and they were mapped to the usual set of 39 labels only for evaluation; that is, *after* decoding. During decoding no effort was made to fine-tune the language model weight and the phone insertion penalty parameters; they were just set to 1.0 and 0.0, respectively.

For preprocessing we used the mel filter bank outputs di-

rectly, following Mohamed et al. [1]. We had the opportunity to work with exactly the same features as they did in [1], as they kindly gave us the corresponding HTK config file. This preprocesing method extracted the output of 40 mel-scaled filters and the overall energy, along with their Δ and $\Delta\Delta$ values, altogether yielding 123 features per frame.

The weights of the neural net were initialized based on the formula proposed by Glorot et al. [13]. The net was trained using semi-batch backpropagation, with the batch size being 100. The initial learn rate was set to 0.001 and held fixed while the error on the development set kept decreasing. Afterwards it was halved after each iteration, and the training was stopped when the improvement in the error was smaller than 0.1% in two subsequent iterations. This way, the training took only 13-15 iterations on average. All the neurons of the networks were rectifier neurons, apart from the softmax output layer.

The training target function to be optimized was the standard frame-level cross-entropy cost. In the sparsifying experiments it was augmented with the the sparsity penalty term $\lambda \sum log(1 + a_j^2)$, following Sivaram [12], where the sum goes over all rectifier activations a_j, and λ was set to 0.001. Earlier we found that it is harmful to apply the sparsity penalty in the early phase of training [6]. Hence, in the experiments the sparsity penalty was switched on only after 9 training iterations.

4. Results and Discussion

4.1. Choosing the size of the input and the bottleneck layer

In the first set of experiments we looked for the optimal size of the input of the network. For this purpose we used a relatively 'shallow' network with one convolutional layer of 2000 neurons in the lower part, and two hidden layers of 2000-2000 neurons in the upper part. The bottleneck was configured to combine five blocks of outputs from the convolutional layer; hence the size of the bottleneck layer was set to $2000/5 = 400$ units. The intention of the experiments was to determine the best value for the size of the local context and the step size (i.e. the downsampling rate) of this local input window for obtaining the convolutional context. For the local context, three sizes were used: 9, 13 and 17 frames. As regards the downsampling rate, the values 3, 4 and 5 were tried out. These altogether gave nine combinations, for which the phone recognition error rates on the development set are shown in Table 1. For instance, a local input size of 9 and convolutional input of $0, \pm 3, \pm 6$ means that input blocks of 9 frames of data are used, and five such blocks are processed, positioned at 0, ± 3 and ± 6 frames away from the central frame. As can be seen from the results, context size of 9 and sub-sampling rate of 5 yielded the best results. Some other settings gave very similar scores, and we chose this combination partly because it had the lowest frame-level error rate as well (not shown here).

Next, we wondered whether the narrower bottleneck layer causes any performance degradation. To test this, the experiment with the best parameter values was repeated, but with an increased bottleneck layer size of 1000 neurons (which meant that the input size of the subsequent hidden layer was 5000). As shown in the last row of Table 1, we did not find any increase in the recognition accuracy, so in all subsequent experiments the bottleneck size was set to 400.

4.2. Adding more hidden layers and enforcing sparsity

Having fixed the size of the input and the bottleneck layer, we considered two other ways to improve the model and hence the recognition results. One was to refine the model by introducing more hidden layers. For this purpose, the lower part of the network was modified so as to contain two or three hidden layers instead of just one. The other was to extend the error function with the sparsity term described in Section 3. The recognition results obtained with these refinements are shown in Table 2. In contrast with non-convolutional nets where we saw a consistent improvement both with increasing network depth and with the enforcement of sparsity [6], here the scores are not fully convincing. Checking the scores on the development set, the introduction of additional hidden layers does not seem to improve the results, and the sparsity penalty does not have a significant effect either, apart from the case of three convolutional layers[1]. After analyzing the results got with the sparsity penalty, we found that while the frame-level error rate (which we minimize during training) decreased *in every case*, the phone error rate (which we measure during evaluation) frequently stagnated or even increased. This clearly shows that we definitely need a modification of our training algorithm that would optimize the error at the sequence level rather than at the frame level, like the methods described by Kingsbury [14] and Mohamed et al. [15].

Sizes of hidden layers	Context		Error on dev. set
	local	conv.	
2000-400-2·2000	9 frames	$0, \pm 3, \pm 6$	16.82%
2000-400-2·2000	9 frames	$0, \pm 4, \pm 8$	16.41%
2000-400-2·2000	9 frames	$0, \pm 5, \pm 10$	**16.35%**
2000-400-2·2000	13 frames	$0, \pm 3, \pm 6$	16.86%
2000-400-2·2000	13 frames	$0, \pm 4, \pm 8$	16.66%
2000-400-2·2000	13 frames	$0, \pm 5, \pm 10$	17.22%
2000-400-2·2000	17 frames	$0, \pm 3, \pm 6$	16.36%
2000-400-2·2000	17 frames	$0, \pm 4, \pm 8$	16.86%
2000-400-2·2000	17 frames	$0, \pm 5, \pm 10$	17.03%
2000-1000-2·2000	9 frames	$0, \pm 5, \pm 10$	16.53%

Table 1: Phone error rates on the development set for various sizes of local and convolutional context. The minimum is typeset in bold.

4.3. Training the lower and upper parts separately

The convolutional structure we used here is closely related to the 'hierarchical' or '2-stage' scheme often applied with conventional neural nets [16, 17, 18]. In this approach a neural net is trained on a block of consecutive feature vectors, then a second network is trained on the output of the first net (again using several consecutive vectors as input). Compared to our convolutional scheme, we see two main differences. First, these hierarchical models usually do not sub-sample the output of the lower net, though there are exceptions [19]. Second, the two nets are trained separately, while in our network the error is propagated down from the upper to the lower part, through the bottleneck layer. On one hand, one would expect the joint training to be more optimal than the separate one. On the other, backpropagation has certain difficulties when training deep structures like our seven-layer construct here [13]. For example, Seide et al. found that a layer-wise training of a deep network gives a much better performance than training the whole structure in one go [4]. Although that result is for sigmoid nets and rectifier units

[1] Although in the table we gave the results on the core test set as well, and these show more convincing trends, we are of course not allowed to select any meta-parameter based on the test results.

Network size	devel. set	core test set
2000-400-2·2000	16.35%	20.02%
+sparsity	16.32%	20.13%
2·2000-400-2·2000	16.63%	19.98%
+sparsity	16.62%	18.71%
3·2000-400-2·2000	16.40%	19.32%
+sparsity	**16.05%**	**18.64%**

Table 2: Phone error rates on the development set and on the core test set for various number of hidden layers, trained without and with sparsity penalty.

2-step training strategy	devel. set	core test set
lower, then both	15.41%	18.66%
lower, then upper	16.04%	19.12%
lower+sp, then both	15.36%	18.55%
lower, then both+sp	15.95%	18.46%
lower+sp, then both+sp	16.02%	18.36%
lower+dr, then both+dr	**14.66%**	**17.76%**

Table 3: Phone error rates on the development set and on the core test set using various two-step training strategies ('sp' stands for sparsity and 'dr' for dropout).

seem to behave favorably in this respect, we thought that the option of two-step training should be investigated experimentally.

In these experiments we used the network structure that performed best in the previous experiments; that is, the lower part contained three hidden layers of 2000 units, the upper part had two hidden layers of 2000 units, and the bottleneck layer consisted of 400 units. The steps of two-step training were as follows (for similar training methods see [9] as well):

- Step 1: The lower part of the network is trained by attaching the output layer directly to the bottleneck layer (the input to this network consists of only one local block of data, so no convolution is involved).
- Step 2: The output layer is thrown away and the full convolutional network is constructed (with randomly initialized upper layers). Only the upper part is trained for one iteration, then the whole network is trained.

The results obtained with the two-step training method are shown in the first row of Table 3. Compared with the best result in Table 2, this method provides a significant improvement on the development set and a similar result on the core test set, though here the sparsity penalty term has not yet been activated. This suggests that there is still room for improvements in the one-step training of our convolutional deep network.

Next, we sought to imitate the separate training of the lower and upper networks, in a similar way to what happens in the hierarchical systems mentioned above. For this purpose, the second training step was modified so that it updated only the weights of the upper part. The results are shown in the second row of Table 3, and a comparison with the scores of the first row justified our expectation that the separate training of the lower and upper parts ought to perform worse. Notice, however, that the result is still slightly better than that obtained when training the full network in one go (see the row before last in Table 3, ignoring the results with the sparsity term for a fair comparison).

The following step was to exploit the gain offered by the sparsity penalty term. However, as was explained in Section 3, earlier we found that for good results the sparsity penalty should be turned on only in the final phase of training, when the weights are relatively stable. Hence, its application in the case of two-step training is not obvious at all. Here three strategies were tried: applying it only during the first training step, applying it only during the second training step, and applying it during both steps. As the results in Table 3 show, none of these strategies really managed to improve on the previous best result on the development set. The lowest error rate attained was 15.36%, and the same model gave 18.55% on the core test set.

After being disappointed with the sparsity penalty results, we tried out a quite new technique called 'dropout' [20]. Here, on each presentation of each training case, each hidden neuron is randomly omitted from the network. This trick helps prevent the co-adaptation of units, while its implementation is also very simple. However, the price is that many more training iterations are required for convergence: in our case, we modified our code so that one training iteration consisted of ten sweeps through the data instead of just one. Also, instead of the value of 0.5 proposed originally, we decreased the chance of 'dropout' to 0.2, because this gave better results in some pilot tests. With these settings we obtained a 14.66% error rate on the development set and 17.76% on the core test set (using 2-step training), and these scores are significantly better than any of our previous results. This shows that this technique has a huge potential, and we intend to investigate it more systematically in the future.

Now let us compare our results with other reported scores on TIMIT. As regards our earlier study, there we reported 19.8% using a non-convolutional deep rectifier neural net trained on the same features and triphone units [6]. The improvement due to the convolutional structure is 1.2%, which is about 6% relative. As regards other authors, they mostly use monophone labels only, so the results are not fully comparable. For example, Abdel-Hamid achieved 20.07% with their convolutional network using monophone labels [8]. In a recent paper Hinton et al. reported 19.7%, and they claimed it to be a new record [20]. The paper by Plahl et al. is one of the few examples were triphone labels are applied; they reported an error rate of 19.1% using a discriminatively trained, boosted feature set [5]. Thus, to the best of our knowledge, the 17.76% reported here is the lowest error rate achieved so far on the TIMIT core test set.

5. Conclusions and Future Work

In this paper, we extended our earlier results with deep rectifier neural nets to a convolutional network structure. This structure allows the network to process a longer observation context without requiring significantly more parameters. The network was constructed from rectifier units, as these were supposed to aid the efficient training of the deep architecture presented by the convolutional network. On the one hand, we achieved a 6% relative error reduction compared to the non-convolutional version of the same net, so the convolutional structure definitely proved efficient. On the other hand, although we obtained good results by simply training the full network in one go, the two-step training strategy operated more convincingly, even so that in this case we could not really make the sparsity penalty work. Thus, finding the optimal way of its usage requires more investigation. Further, though Glorot et al. found that pre-training does not help rectifier nets [10], this may vary from task to task, so we intend to test it in the speech recognition case. And, finally, we see a great potential in the more sophisticated convolutional structure of Abdel-Hamid [8] and in the dropout method [20].

6. References

[1] A. Mohamed, G. E. Dahl, and G. Hinton, "Acoustic modeling using deep belief networks," *IEEE Trans. ASLP*, vol. 20, no. 1, pp. 14–22, 2012.

[2] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large vocabulary speech recognition," *IEEE Trans. ASLP*, vol. 20, no. 1, pp. 30–42, 2012.

[3] D. Yu, L. Deng, and G. E. Dahl, "Roles of pre-training and fine-tuning in context-dependent DBN-HMMs for real-world speech recognition," in *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning*, 2010.

[4] F. Seide, G. Li, L. Chen, and D. Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Proc. ASRU*, 2011, pp. 24–29.

[5] C. Plahl, T. N. Sainath, B. Ramabhadran, and D. Nahamoo, "Improved pre-training of deep belief networks using sparse encoding symmetric machines," in *Proc. ICASSP*, 2012, pp. 4165–4168.

[6] L. Tóth, "Phone recognition with deep sparse rectifier neural networks," in *Proc. ICASSP*. 2013, accepted, in print.

[7] Y. Lecun and Y. Bengio, "Convolutional networks for images, speech and time series," in *The Handbook of Brain Theory and Neural Networks*, Michael A. Arbib, Ed. 1995, pp. 255–258, MIT Press.

[8] O. Abdel-Hamid, M. Abdel-rahman, H. Jiang, and G. Penn, "Applying convolutional neural network concepts to hybrid NN-HMM model for speech recognition," in *Proc. ICASSP*, 2012, pp. 4277 – 4280.

[9] K. Veselý, M. Karafiát, and F. Grézl, "Convolutive bottleneck network features for LVCSR," in *Proc. ASRU*, 2011, pp. 42 – 47.

[10] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. AISTATS*, 2011.

[11] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. ICML*, 2010, pp. 807–814.

[12] G.S.V.S. Sivaram and H. Hermansky, "Sparse multilayer perceptron for phoneme recognition," *IEEE Trans. ASLP*, vol. 20, no. 1, pp. 23–29, 2012.

[13] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. AISTATS*, 2010, pp. 249–256.

[14] B. Kingsbury, "Lattice-based optimization of sequence classification criteria for neural-network acoustic modeling," in *Proc. ICASSP*, 2009, pp. 3761–3764.

[15] A. Mohamed, D. Yu, and L. Deng, "Investigation of full-sequence training of deep belief networks for speech recognition," in *Proc. Interspeech*, 2010, pp. 2846–2849.

[16] H. Ketabdar and H. Bourlard, "Enhanced phone posteriors for improving speech recognition systems," *IEEE Trans. ASLP*, vol. 18, no. 6, pp. 1094–1106, 2010.

[17] J. Pinto et al., "Analysis of MLP based hierarchical phoneme posterior probability estimator," *IEEE Trans. ASLP*, vol. 19, no. 2, pp. 225–241, 2010.

[18] L. Tóth, "A hierarchical, context-dependent neural network architecture for improved phone recognition," in *Proc. ICASSP*, 2011, pp. 5040 – 5043.

[19] D. Vásquez, G. Aradilla, R. Gruhn, and W. Minker, "A hierarchical structure for modeling inter and intra phonetic information for phoneme recognition," in *Proc. ASRU*, 2009, pp. 124–129.

[20] G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.