Analysis of denoising autoencoder properties through misspelling correction task

Karol Draszawka, Julian Szymański

Department of Computer Systems Architecture
Faculty of Electronic Telecommunications and Informatics
Gdańsk University of Technology, Poland,
kadr@eti.pg.gda.pl, julian.szymanski@eti.pg.gda.pl

Abstract. The paper analyzes some properties of denoising autoencoders using the problem of misspellings correction as an exemplary task. We evaluate the capacity of the network in its classical feed-forward form. We also propose a modification to the output layer of the net, which we called multi-softmax. Experiments show that the model trained with this output layer outperforms traditional network both in learning time and accuracy. We test the influence of the noise introduced to training data on the learning speed and generalization quality. The proposed approach of evaluating various properties of autoencoders using misspellings correction task serves as an open framework for further experiments, e.g. incorporating other neural network topologies into an autoencoder setting.

Keywords: Autoencoder, misspellings, autoassociative memory

1 Introduction

Experiments presented in this paper aims at researching properties of autoencoders – neural network-based models of autoassociative memory [1]. To do this we need data that is scalable, easily obtainable, can be modified in such a way that allows to test particular properties of autoencoders, and corresponds to some popular, common sense task. The task of misspelling correction fulfills these requirements – it is widely used for supporting writers to ensure quality of their work [2]. The data can be represented in the form of large dictionaries of strings, that can be modified according to experiment requirements, e.g. by injecting a particular type of noise (typos).

Typical approach to complete the task of misspellings correction is to provide a priori dictionary and to select the entries that are the most probable given the distorted one presented on input. This requires the comparison between the input string and all entries in the dictionary using some similarity measure. This approach has a linear complexity $O(n)$, where n is the size of the dictionary. The quality of the results depends on the representation of dictionary entries and similarity measure used for string comparison [3] [4]. It may be e.g.: Cosine, Hamming, Levenshtein distances or their modifications [5]. Regardless of what

similarity measure is used the complexity in that approach remains the same. One way to improve the efficiency is to used dedicated indexes [6] or hashing [7] [8].

In this work we use misspellings correction as the task under which we examine properties of denoising autoencoders. Conducted experiments show that this task can be successfully completed using such neural networks. This solution provides high quality results, the $O(1)$ complexity and the possibility to tune the model for user preferences. Also representation of the dictionary is much more compact and saves memory.

2 Autoencoders as Typo Correctors

2.1 Denoising Autoencoders

As stated, for large scale dictionaries finding closest matches to a mistyped sequence of characters using some similarity measure may be inapplicable due to $O(n)$ complexity. On the other hand, existing techniques of scalability improvements introduce accuracy problems. For example, solutions based on Locality Sensitive Hashing [6], may decrease precision and/or recall of dictionary entries retrieval. The quality of results is strongly related to the similarity function that is employed – the more precise results we want to obtain the more sophisticated, and computationally expensive, function need to be used. In some circumstances, there is also a need for tuning of the similarity function so that it works better with a particular dictionary, e.g. to construct user personalized spelling corrections, when characteristic examples of his or her misspellings are provided.

Because of these drawbacks of traditional methods, we tested autoencoders as an alternative misspellings correctors. Autoencoder is an artificial neural network model, that tries to map a given input vector to itself (see fig. 1). The computation is not trivial, because the information flowing from input to output goes first through encoding part (function f), which squeezes it into smaller number of a *bottleneck* layer (it may also be otherwise restricted, e.g. through sparse regularization), before it is recovered back with decoder function g. Autoencoders are trained by minimizing a loss function $L(\mathbf{x}, g(f(\mathbf{x})))$ that penalizes net output $g(f(\mathbf{x}))$ for being dissimilar from net input $\mathbf{x}$. For real valued input vectors L is typically chosen to be mean squared error (MSE), but for binary data it can also be binary cross-entropy (CE) loss [9]. When decoder is linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as Primary Component Analysis (PCA) [10]. Autoencoders with nonlinear f and g can learn more powerful nonlinear embeddings than PCA.

Typically, the aim of training such neural networks is to obtain a useful compact representation for a given data set, i.e. for the purpose of dimensionality reduction [11–13]. In these cases, we are not interested in the output of the decoder, but in the hidden layer, where internal representations of the input data are formed. However, autoencoders can also be used as noise reductors, for

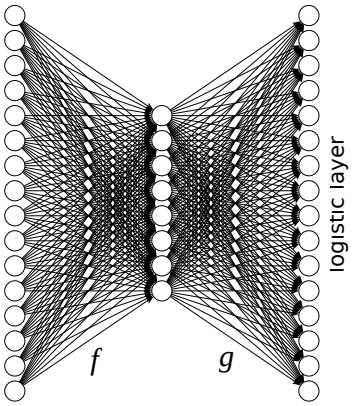


Fig. 1. Typical shallow autoencoder architecture: feed forward neural net with one *bottleneck* hidden layer. Sigmoid activation function at output.

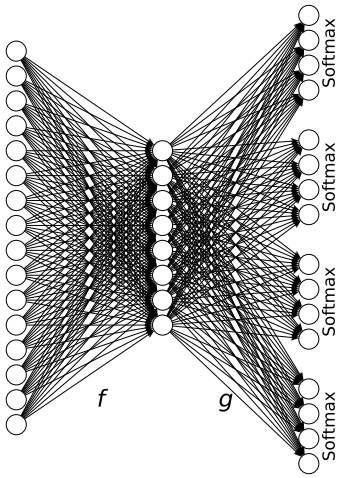


Fig. 2. Autoencoder with multi-softmax output layer.

example like shown in [14] for enhancing speech signal, or in [15] for removing different types of noise from images. For such *denoising* autoencoders, training dataset consists of $(\hat{\mathbf{x}}, \mathbf{x})$ input-target data pairs, where $\hat{\mathbf{x}}$ is a corrupted version of the original $\mathbf{x}$ instance. The corrupted version is formed from original one during generation of training pairs.

2.2 Words Representation

In this context, misspellings correction can be seen as a denoising process, for which denoising autoencoders are perfectly suitable. All what has to be done is to transform sequence of characters into a fixed length feature vector, which is then directly applicable to neural networks input and targets. In this work, we used *one-hot* encoding of characters: let n be the number of symbols in the alphabet, then i-th symbol is associated with a vector $v(s^{(i)})$ of length n with all elements equal to 0 except i-th, where it has 1. Sequence of symbols is represented as a concatenation of all associated $v(s)$ vectors. For example, if alphabet $\mathcal{A} = \{$'a', 'b', 'c'$\}$, then v('a') $= [1, 0, 0]$, v('b') $= [0, 1, 0]$, v('c') $= [0, 0, 1]$ and v('cab') $= [0, 0, 1, 1, 0, 0, 0, 1, 0]$.

The reverse transformation, i.e. from binary vectors to strings of characters is analogical. However, vectors returned by a neural net will not be binary, but real valued, with all elements between 0 and 1. Thus, the returned vector is split into vectors representing each character, and then the position of the maximum element in each of the vectors determines decoded symbols. For example, if $\mathcal{A}$ is the same as in previous example, then vector $[0.780.210.01, 0.1, 0.89, 0.01, 0.0, 0.99, 0.1, 0.6, 0.37, 0.3]$ is decoded as string 'abba', because we have symbol vectors: $[0.780.210.01]$ with maximum at first position, $[0.1, 0.89, 0.01]$ with second

element being maximal, $[0.0, 0.99, 0.1]$ also with max at second position, and $[0.6, 0.37, 0.3]$ with argmax equal to 1.

To obtain fixed-sized vectors for all input sequences, regardless of their length, space padding is used (space being added to alphabet). For the purpose of the experiments we assumed maximum length of the sequence, denoted as C, equal 16. All shorter expressions where padded to this size with spaces. In result, 576-element ($C \cdot |\mathcal{A}|$) binary sparse vector represents a word that can be formed from 36-symbols alphabet not longer that 16 characters.

2.3 Output Layers and Loss Functions

The output of our model for typos correction is specific: it is interpreted as a concatenation of C probability distributions of characters from alphabet $\mathcal{A}$. Since this is a concatenation of distributions, and not a single probability distribution, we cannot apply a softmax output function to here. Instead, two possibilities were tested.

In the first one, the models is as seen in fig. 1, i.e. with a normal dense layer of sigmoid units (with standard logistic activation function $\sigma(z) = 1/(1 + \exp(-z))$). This type of network models arbitrary binary outputs. The loss function L for a single training target $\mathbf{t}$ is a binary cross-entropy:

$$L(\mathbf{y}, \mathbf{t}) = -\sum_{i=0}^{|\mathcal{A}|C-1} \left[t_i \log(y_i) + (1 - t_i) \log(1 - y_i) \right], \tag{1}$$

where $\mathbf{y}$ is the output of the net.

The other one is presented in fig. 2. This is a configuration, which we call *multi-softmax*, which is a concatenation of dense softmax layers each calculating normal softmax function $\sigma(\mathbf{z})_j = \exp(z_j) / \sum_{k=1}^{K} \exp(z_k)$. The loss function L for a single training target $\mathbf{t}$ for such an output layer is a sum of categorical cross-entropies, each corresponding to appropriate softmax:

$$L(\mathbf{y}, \mathbf{t}) = -\sum_{i=0}^{C-1} \sum_{j=0}^{|\mathcal{A}|} t_{i|\mathcal{A}|+j} \log(y_{i|\mathcal{A}|+j}). \tag{2}$$

This model has the same number of parameters as normal sigmoid layer, yet it enforces the output vector to be a valid concatenation of probability distributions.

3 Experiments

3.1 Experimental Setup

In all the experiments described below, we used our implementation of autoencoders in Keras [16] with Theano [17] back-end. We trained models for 1000

epochs using AdaDelta optimizer with default settings [18]. If not stated otherwise, training and testing data is based on a 1000 long Polish words dictionary (av. length of word is 13.02 characters).

Except an experiment that investigates generalization capabilities of presented models in detail (discussed further), all other experiments were conducted using the following strategy. Instead of creating fixed-sized sets of training and testing data, we use generators that create data on-the-fly during training and testing phases. During training, generator creates training pairs $(\mathbf{x}, \mathbf{x})$ simply taking a random word from a dictionary and transforming it into vector representation, or creates $(\hat{\mathbf{x}}, \mathbf{x})$ substituting one character of the word with a different random one from the alphabet. This way, any possible misspelling of a word (resulting from mistyped a single character) could be possibly generated. In one epoch, we give 100 such pairs per each dictionary entry. After each 10 epochs, models are evaluated using testing data generator: for each word in a dictionary, 100 one-character typos is generated. So, for 1000 words, 100000 tests are performed, and the percentage of times, when the net returns a correct word from typo is reported as test accuracy.

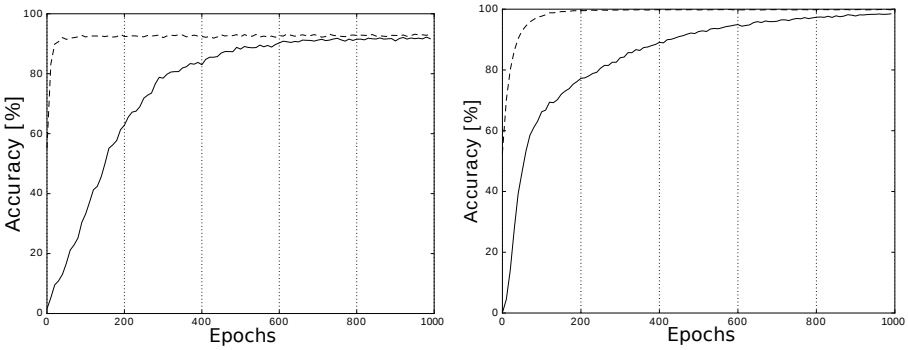
The number of units in the hidden layer is crucial for autoencoder to work properly. First experiment under these settings was conducted to selected the number of hidden units in the bottleneck layer. The results are shown in Table 1. It can be seen that more units give better results. However, this comes at the cost of complexity and computation time. Therefore we have chosen a shallow autoencoder of 300 hidden units as a good compromise between accuracy and complexity and all subsequent experiments use models with 300 neurons in the middle layer. We also tested deeper architectures (such as e.g. 576-1000-300-1000-576), but they performed similarity at much higher computational cost.

Table 1. The impact of the number of hidden units in the bottleneck layer of a shallow autoencoder. Each setup was trained and tested two times (dashed lines) and averaged (solid line).

Hidden layer size	10	30	100	300	400	576	700	1000
Average test accuracy [%]	0.13	36.2	67.3	71.2	71.1	71.9	72.6	73.5

3.2 Sigmoid vs Multi-Softmax Output Layer

Providing the information to the network that each of the characters is coded as separate probability distribution significantly increases the speed of learning process. Our multi-softmax adjustment causes the network to perform much better than simple layer with sigmoid units, see Fig. 3. There, we present test accuracy results and number of epochs required to train the shallow autoencoders with 1 hidden layer of 300 units on word dictionaries containing 100 and 1000 elements.



(a) 100 words. (b) 1000 words.

Fig. 3. Sigmoid (solid line) and multi-softmax (dashed line) test accuracy for different sizes of training dictionary.

Multi-softmax output layer constantly outperforms normal sigmoid layer in both cases. In the case of 1000 words dictionary, multi-softmax enables autoencoder to achieve almost 100% test accuracy. Interestingly, performance on 100 word set is worse than on 1000 word set. This is because our 100 word dictionary contains very similar words (each begin with the same three letters), whereas 1000 word dictionary contains diverse entries and it is easier to find a correct word of misspelled one.

The time required to train the network is also very important factor to have practical applications of proposed approach. Training on low-end GPU, our 576-300-576 autoencoder took a couple of minutes for a dictionary of 100 words and about 2 hours for 1000 words. We tested those models also on 10000 dictionary, where 1000 epochs took over 22 hours, which was was not enough to obtain a good test accuracy (69.3% for multi-softmax, 64.1% for sigmoid), showing that 576-300-576 models are too small for 10000 words.

3.3 Impact of Input Noise on the Model

Table 2. Impact of the percentage of distorted training input examples on model performance.

How often a training word contains a typo	0%	50%	67%	80%	90%	97%	99%	100%
Average test accuracy [%]	53.3	80.3	85.9	89.3	91.0	92.6	**93.3**	91.7

Misspellings correction can be seen as a transformation of a noised input into an output without the noise. The type noise we use in the experiments

is random character substitution, but other examples of misspellings, such as reversed order of neighboring chars or additional/missing chars, are possible as well. The question is: how much noise is needed for a denoising autoencoder to train it maximally effectively.

Table 2 shows the impact of the percentage of input training examples having typo. The case when all examples are without any single character changed is denoted as 0% case. On the other extreme, 100% case means that every time a word from the dictionary is presented to the model, it has a random character substitution at one random place. It can be seen that presenting distorted words at training significantly improves generalization of the autoencoder. This is expected, because the task is just to do this (at test time, all examples have one random character substitution). The results indicate that training an autoencoder on the correct data leads to very poor generalization. On the other hand, it is also beneficial to include in a training data an correct examples of words. The best result (93.3% test accuracy) was obtained when one 1 in a 100 examples was correct. Also 97% setting achieved better result than 100% distorted data.

Another factor we evaluated was the level of distortion the network is still able to reconstruct. The assumption of the experiment presented here was that if we add more distorted data into the learning set the network can achieve better generalization properties. Positional representation of characters in the input vector means that there is a lot of independence among positions in the word. We tried to make use of this independence and tested whether more than one random substitutions can be made *in each* input example, although at test time examples still have only one substitution. The hope was that more 'denoising work' could be learned in a mini-batch processing than merely one random substitution in a random place per example.

In Table 3 we present results of the experiment showing how distortion level influence accuracy. The best performance was achieved when each training example contains 3 typos (test accuracy 97.3%). 5 typos per training word performs almost equally well. But 10 typos per word (when words are of 16 characters maximally) is too much – the words are distorted to such a degree that they cannot be reconstructed any more. This experiment was performed in two settings (same as in previous experiment), one in which 50% of examples is distorted and other are left unchanged, and second when 100% of examples is distorted. The second setting constantly outperforms the first one, and the results from this experiment conform previously drawn conclusions: it is better to show various forms of distorted data than to present many times (50% means each second epoch) the same correct version words.

3.4 Generalization Abilities

In all experiments presented so far, training and testing data are generated online from two generators, which are sampling words from the same dictionary of words. Although typos made in training and testing examples are constantly different, it is possible that in some epoch t test generator will generate a typo

Table 3. The impact of the number of character substitutions in every one distorted word presented to the model during training. Test accuracy [%].

Number of typos in each perturbed example	**1**	**2**	**3**	**5**	**10**
50% perturbed training examples	80.1	89.6	91.2	91.6	85.4
100% perturbed training examples	91.9	96.2	**97.3**	97.1	88.6

exactly the same as previously created by train generator in epochs $\leq t$. In fact, the probability of such situation is monotonically growing during training, therefore, test accuracies will slowly converge to train accuracies. Although presented procedure is good for training misspellings correctors, it cannot reliably answer a question whether such systems can generalize to correct other typos than presented during training.

To answer this question we created four disjoint fixed-sized sets: one training and three testing data. The first one, denoted as L1, contains typos very similar to those from the training set: they are made in the same words and at the same positions as in the train dataset, only wrong character is different. The second test set, L2, holds typos made in the same words as examples from the training set, but at positions which are never altered in the training set (each word in the training set has one randomly chosen position in it, where it always has its correct character). The third test set, L3, has typos made in words not present in the training set dictionary at all.

Fig. 4 shows accuracies of typos corrections for each of the datasets for 576-250-576 autoencoders with sigmoid and multi-softmax output layers. Correction unseen typos of L1 type is not a problem at all, the performance being essentially the same as for train data. Correction of L2 typos is only slightly harder for multi-softmax model, while sigmoid net struggles here, 500 epochs is not enough to obtain good accuracy, but increasing tendency indicates that it is not overfitting training data yet, and is able to generalize to L2 dataset. Neither model is able to correct words that they did not see during training. This is not a surprise, because the models are trained to associate their input to whole words from the dictionary used during training, and have no clue that out-of-vocabulary output is possible.

4 Summary and Further Works

In this paper we present results of the research on autoencoders for misspelling correction task. We argue that the observations given here regarding the properties of autoencoders' behavior are general and may be used for other applications that employ this type of networks. We shown that introduction of so-called *multi-softmax* layer significantly improves learning results. The results of the experiments shown that there is a finite capacity of the network. Developing it into larger topology is not a good way to extend the dictionary size because the time required to train the network significantly increases. Instead of enlarging the network, in future we plan to introduce hierarchical structure with root network at

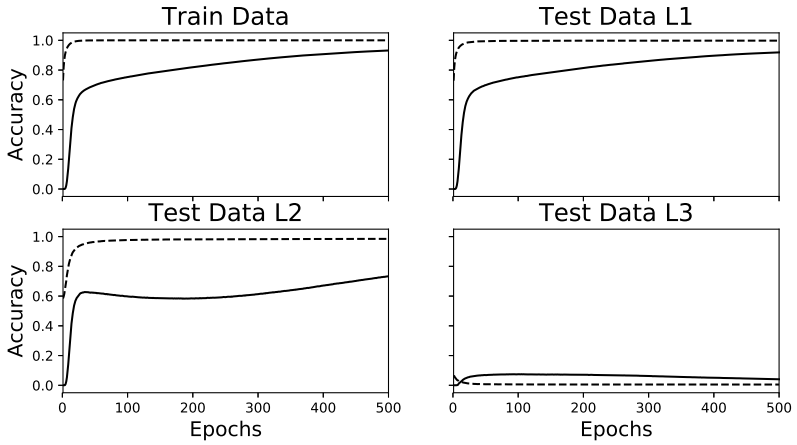


Fig. 4. Accuracy on train and three different test sets. Solid lines represent sigmoid model, dashed lines – multi-softmax model. Details describing datasets are in the text. Axes are shared between plots.

the top that will select subspace of the dictionary and indicates proper network to perform misspelling corrections in that subspace. This task can be performed using Self Organizing Map [19] that aggregate in each node the most similar entries from the dictionary. Selecting the size of the map allows us to configure the number of subnetworks used for a particular dictionary.

The framework we create during implementation of the experiments allows us to test other network topologies that may be applied as autoencoders. In future, instead of adding simple dense hidden layers we plan to test capacity of convolutional layers [20]. Also usage of deep residual networks [21] seems to be a good direction for extending the capacity and accuracy of the model. In the experiments presented here we used static vectors to represent the input strings. The other promising direction of research is to treat words as sequences where next character depends in some degree on the previous one. This naturally suggest recurrent networks [22], that can be incorporated to autoencoder model (as encoder as well as decoder part) and provide information on successive dependencies in training sequences.

References

1. Fausett, L.V.: Fundamentals of neural networks. Prentice-Hall (1994)
2. Kukich, K.: Techniques for automatically correcting words in text. ACM Computing Surveys (CSUR) **24** (1992) 377–439
3. Baeza-Yates, G. Navarro, R.: Faster approximate string matching. Algorithmica **23** (1999) 127–158
4. Szymanski, J., Boinski, T.: Improvement of imperfect string matching based on asymmetric n-grams. In: Computational Collective Intelligence. Technologies

and Applications - 5th International Conference, ICCCI 2013, Craiova, Romania, September 11-13, 2013, Proceedings. (2013) 306–315
5. Astrain, J.J., Garitagoitia, J.R., Villadangos, J.E., Fariña, F., Córdoba, A., de Mendıvil, J.G.: An imperfect string matching experience using deformed fuzzy automata. In: HIS. (2002) 115–123
6. Boguszewski, A., Szymański, J., Draszawka, K.: Towards increasing f-measure of approximate string matching in o(1) complexity. In: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS). (2016) 527–532
7. Hantler, S.L., Laker, M.M., Lenchner, J., Milch, D.: Methods and apparatus for performing spelling corrections using one or more variant hash tables (2006) US Patent App. 11/513,782.
8. Udupa, R., Kumar, S.: Hashing-based approaches to spelling correction of personal names. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics (2010) 1256–1265
9. Rubinstein, R.Y., Kroese, D.P.: The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning. Springer Science & Business Media (2013)
10. Jolliffe, I.: Principal component analysis. Wiley Online Library (2002)
11. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. science **313** (2006) 504–507
12. Bengio, Y., et al.: Learning deep architectures for ai. Foundations and trends® in Machine Learning **2** (2009) 1–127
13. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. Journal of Machine Learning Research **11** (2010) 3371–3408
14. Lu, X., Tsao, Y., Matsuda, S., Hori, C.: Speech enhancement based on deep denoising autoencoder. In: Interspeech. (2013) 436–440
15. Agostinelli, F., Anderson, M.R., Lee, H.: Adaptive multi-column deep neural networks with application to robust image denoising. In: Advances in Neural Information Processing Systems. (2013) 1493–1501
16. Chollet, F.: Keras. `https://github.com/fchollet/keras` (2016)
17. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints **abs/1605.02688** (2016)
18. Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 (2012)
19. Kohonen, T.: The self-organizing map. Neurocomputing **21** (1998) 1–6
20. Kalchbrenner, N., Grefenstette, E., Blunsom, P.: A convolutional neural network for modelling sentences. arXiv preprint arXiv:1404.2188 (2014)
21. Zagoruyko, S., Komodakis, N.: Wide residual networks. arXiv preprint arXiv:1605.07146 (2016)
22. Botvinick, M.M., Plaut, D.C.: Short-term memory for serial order: a recurrent neural network model. Psychological review **113** (2006) 201