NN Assignment 2

April 2018

Introduction

In this assignment we will be implementing neural network using Keras, making the process of creating and tuning neural networks significantly easier. Keras is such a package, allowing the user to create a neural network with only a couple of lines of code, and though some customizability might be lost, all the most commonly used architectures can be easily created using Keras. Keras uses a more general neural networks package such as Tensorflow or Theano as a backend, functioning as a sort of wrapper for these specialized packages. For our purposes, there is no reason to deviate from the norm, so the Tensorflow backend is used. We will also we will studying Recurrent neural network and autoencoders implementations.

Task 1: Keras - MNIST

In the previous assignment we created our own multi-class perceptron and neural network in order to classify handwritten digits from the MNIST data set. This famous data set has several Keras example scripts dedicated to it, meaning that we can now look at some basic architectures and see how they perform on a data set we are already familiar with.

Multilayer perceptron (MLP)

The multilayer perceptron works like a classical neural network. Some input is given, in this case the 28x28 images from the MNIST dataset, the network goes through some hidden layers, and then produces an output, in this case a vector of size 10, each output node containing the probability of the input image being the digit that belongs to said node. Each layer has an activation function, the most common being the 'ReLu' (rectified linear units) function. Additionally, after each layer there is something called the 'Dropout' layer, which randomly sets some fraction of the inputs to zero in order to prevent over-fitting. We will stick with the architecture that is specified in the script, as our only aim here is to see how well

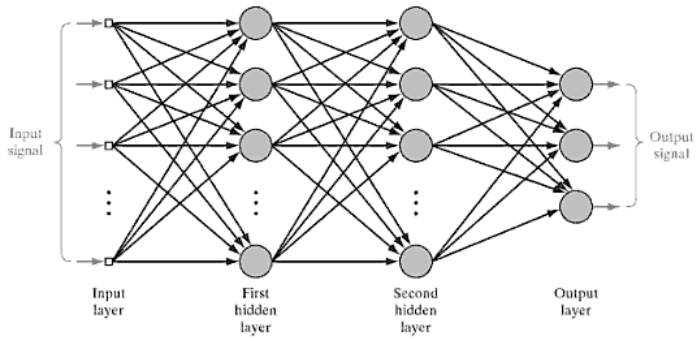
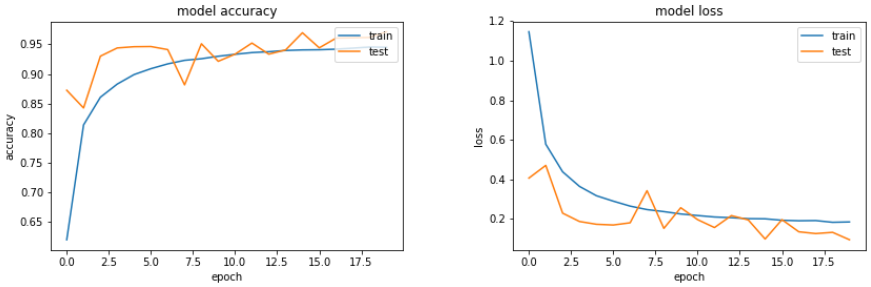


Figure 1: A graphical representation of a multilayer perceptron with two hidden layers.

this multilayer perceptron does under different circumstances. When running them as is, an accuracy of about 98% on the test set is achieved when training the network for 20 epochs. This is already a lot

better than the perceptron we trained in the previous assignment, which only achieved an accuracy of about 87% on the test set. Besides the overall accuracy, it's probably interesting to look at what the top three most 'misclassified' digits are. The term missclassified is somewhat ambiguous here, and can be taken to mean several things. Here, we take most missclassified to mean the digits that are classified as other digits the most, or in other words, have the lowest accuracy associated with them. We run the experiment for two different loss functions, namely categorical crossentropy (the default loss function in this script) and the mean squared error.

Loss function	Accuracy	Most missclassified digits
Mean squared error	98%	5 (96.19%) 8 (96.82%) 9 (97.20%)
Categorical cross entropy	98%	8 (94.15%) 4 (97.86%) 9 (98.02%)



As can be seen, the accuracy between the loss functions stays pretty much the same, and are deliberately rounded off as these are the results of running the algorithm several times to account for the random initialization of the weights. The most missclassified digits are picked from the first run, and can highly vary between runs, although the digits 8 and 9 do appear pretty consistently in the top three.

Convolutional neural network (CNN)

While the previously mentioned multilayer perceptron can be applied to all kinds of data, we now turn to a more specialized algorithm that performs well on very specific data types. The convolutional neural network works especially well on images, consisting of several convolutional layers that can learn 'features' present in images. The shallow layers can learn very general features, while the deeper ones can learn more and more complex ones. The algorithm in the Keras example scripts has two of these convolutional layers, following by a 'max pooling' layer which reduces the size of the output by taking 2x2 squares and returning the maximum value of the square as the input for the next layer. This is followed by two more regular layers with some added dropout layers in between. The input and hidden layers all have the 'ReLu' activation function, while the output layer uses 'softmax' as activation function. Running this

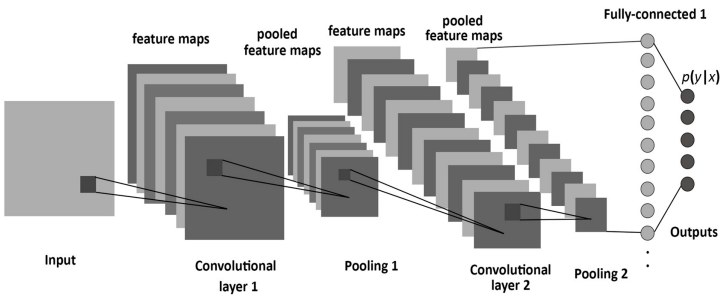
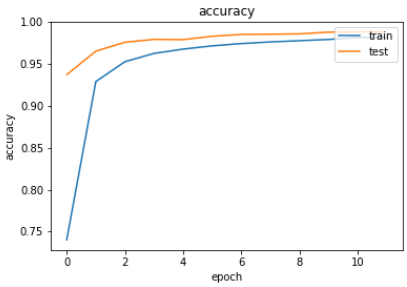
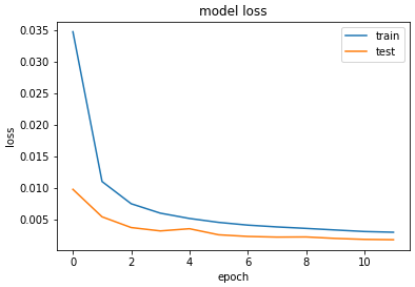


Figure 2: A graphical representation of a convolutional neural network with two convolutional layers.

network as is gives an accuracy of 99% on the test set when training the network for 12 epochs, so it seems this already performs better than the MLP from the previous section. Again, we want to see how it performs for different loss functions as well as what the most missclassified digits are.





Loss function	Accuracy	Most missclassified digits
Mean squared error	98.5%	9 (97.32%) 8 (98.05%) 7 (98.05%)
Categorical crossentropy	99%	8 (98.36%) 6 (98.43%) 9 (98.51%)

When running the network multiple times, it is clear that the results for this network are more consistent than for the MLP from the previous section. The random initialization does not seem to affect this network as much, meaning the results are easier to reproduce and thus these tests give us a good idea of how well the network is truly performing. Using the categorical crossentropy as a loss function seems to give a consistently higher accuracy than using the mean squared error. The most misclassified digits also appear more consistent, with 8 and 9 again consistently appearing in the top three.

Permuting the MNIST dataset

At the end of the day, the entire MNIST data set is just arrays with numbers in them indicating the intensity of each pixel in an image. Keeping this in mind, we can easily permute the entire dataset with some random number and see how the networks perform on it. We can already make a prediction on what happens in this scenario, namely that the networks will not be affected at all. The networks will learn the same features, so as long as everything is permuted in the same way it shouldn't make a difference. Applying such a permutation to the MNIST data set and then running the CNN and MLP algorithms again shows this. There is no discernible effect (negative or positive) on accuracy, proving that a random permutation of the input has no effect on the algorithms. The MLP has an accuracy on this permuted data set of 98%, and the CNN an accuracy of 99%. Even if there is an effect on the accuracy, it is so small that the effect on the accuracy caused by the random initialization of the model far outweighs this.

Task 2: Recurrent Neural Networks (RNN)

So far we've been working with forward neural network, that is the information moves in one direction from the input layer through the hidden layers to the output layer. Neural networks were able to address most of the machine learning problems, but they have the some restrictions and limitations:

1. A neural network takes in a fixed size input and produces a fixed output
2. Input data is independent, so the order of the input data has no affect on the output
3. There is a fixed number of layers
4. They can't understand sequences, in which the current state is affected by its previous states

Recurrent Neural Networks provide some solutions for the above limitations, they are networks with loops in them allowing for information retention,so they remember sequences and make decisions based on current and previous input.
In Figure 3 the left hand side, a neural network with input x and output o, a loop allows information to be passed from one step to the next, by stacking the hidden layers of previous steps, each hidden layer depends on the corresponding input at that step and the preceding step. Making it possible to use the network in predicting the next in a sequences, i.e remember sequences and make decisions based on current and previous input. Figure 3 is a diagram showing an example of an RNN being unfolded into a full network. That is showing the network for the complete sequence, repeating the network some x amount of times.

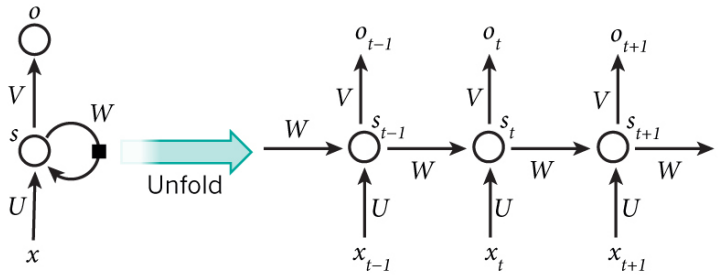


Figure 3: A recurrent neural network and the unfolding in time of the computation involved in its forward computation.

Every iteration then looks as follows:

1. x_t is the input at step t
2. s_t is the hidden state at step t, it is the memory of the network, where $s_t = f(Ux_t + Ws_{t-1})$
3. O_t is the output at step t
4. Since the RNN is performing the same task across all steps t the weights and parameters (U, V, W) are the same.

Now we have the RNN that can remember, but in practice it can't remember across many time steps, and it can't decide which information of some time step is valuable (which it should keep) and which information is not valuable (which it should forget). The way to address this problem was through Long Short-Term Memory, or LSTM. LSTM is an improvement on RNN where there is an extra piece of information "memory" in the LSTM cell at each step, enabling it to decide what information to store in long term and what to forget. [1]

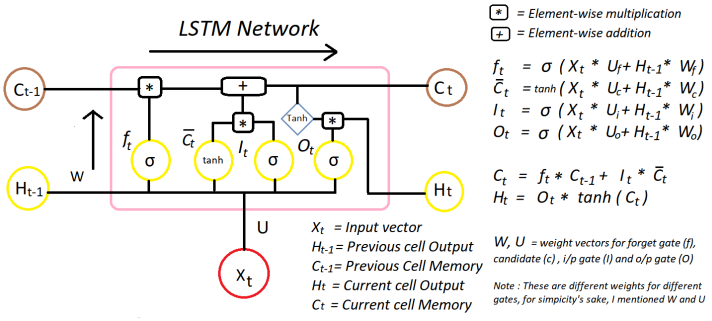


Figure 4: A single LSTM cell.

[1] https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235

Figure 4 shows a LSTM cell at time step t. The different components are: The forget gate f, candidate values $\bar{C}$, the input gate I and the output gate O. The LSTM then operates as follows:

- Take the previous memory state C_{t-1} and do element wise multiplication with forget gate (f): $C_t = C_{t-1} * f_t$. If value is 0 then previous memory state is completely forgotten, and if forget gate value is 1 then previous memory state is completely passed to the cell. Now with current memory state C_t we calculate new memory state from input state and C layer.
- The current memory state at time step t is given by $C_t = C +_t +(It * \bar{C}_t)$, and it gets passed to next time step. $\bar{C}$ decides what information to store from current input, and has two parts.
- The output is given by $H_t = O_t \tanh(C_t)$ which we do element wise multiplication the output gate with the current state (after going through an activation function). This will be our current hidden state H_t.

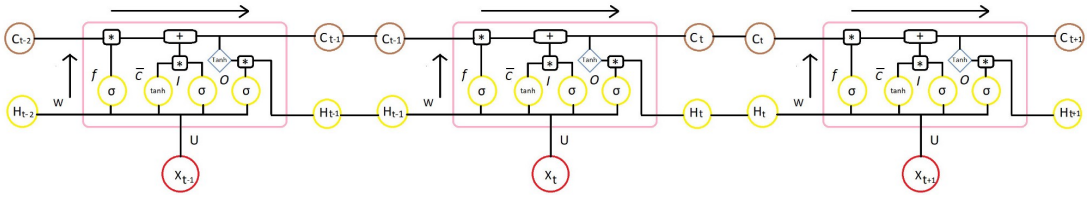


Figure 5: LSTM time steps

It is also worth mentioning that the RNN are very flexible in the sense that the input and output can be of different sizes, hence it can be used in various applications like image captioning, video classification and sentiment analysis.

Many-to-many

We will study the implementations form the keras/examples github, with the focus on the many-to-many LSTM.

Text generation

Text generator is an LSTM neural network that learns concepts from text input (sequence of characters). The text-generator in this study is a single LSTM and this is how it is implemented.

- The Data: Any large text file, Project Gutenberg has a huge repository of public domain books, so any book that is at least 100K in size will suffice. We chose the adventures of Tom Sawyer by Mark Twain from Project Gutenberg to be our training text, we also ran it with the Nietzsche text, which was Keras team choice.

  To prepare the data to be used in NN model:

  1. Convert all characters to lower case (less character sequences to learn from)
  2. Map the characters into integers (to use in the Neural network)
  3. With the mapping we created we convert the character sequences to integers
  4. seq_len = sentence length (arbitrary fixed number)
  5. Step = number of charters to skip to start a new sequence at a time
  6. Transforming the data into vectors(sentences) by splitting the text into sequences with fixed length seq_len, sliding Step characters at a time. If Step = 1 we will shift one character every time thus generating the maximum number of sentences.

- Model : a single hidden LSTM layer with 180 memory units. The network uses RMSprop (Root mean square propagation). The output layer is a Dense layer using the softmax activation function to output a probability prediction for each of the characters between 0 and 1. with categorical cross entropy as the loss function) We are modeling the entire training data set to learn the probability of each character in a sequence.
- Text generation: this is done by picking a random input sequence to be the seed sequence, generate the next character then update the seed sequence to add the generated character on the end, and remove the first character from the input sequence. This process is repeated for as long as we want to predict new characters.
- Experiment: We ran the text generator with different parameters, the step and sentence size, and the number of nodes. The text generator produced text that looked like a well structured sentences, with quotation, punctuation marks, even though some of the words didn't make sence, but we think if the network had more layers it would produce better results. The box below is a sample of the text generator output:

chapter xxx it. huck do, too, says down and a good to see you the door." "well, you'll take it?" "yes." "well, it's a consider's meet you can so him." "there was auntie, tom, i don't be a whis had seen door, why, it's for you and then it ain't to find it." "oh you'll do the energsy?" "i know she don't come him that i lay a shore of the day and i didn't sid. and i didn't do that?" "why, i didn't cot?"

Interestingly enough when we ran the text generator with Step = 1 and 100 epochs the loss was decreasing, then around the 59^{th} epoch it started increasing and it reached 7.4 at the 68^{th} epoch.

Sequence to sequence LSTM

In sequence to sequence LSTM maps an input sequence to an output sequence, their lengths are not necessarily of the same length. It can be used for different application, like translation, speech recognition. For this we use the keras encoder-decoder model. The procedure is to use two RNN, one to Encode the input (source) sequence, and the other to Decode the encoded source to the desired (target) output.

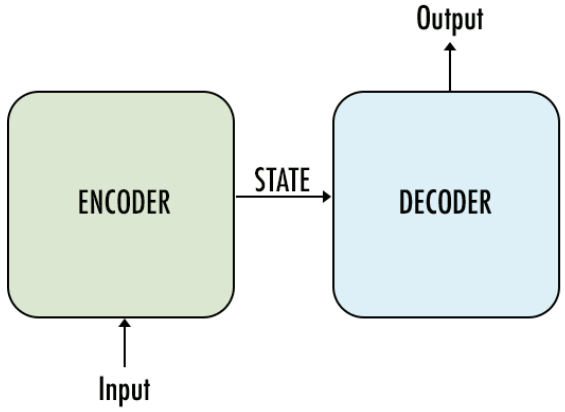


Figure 6: Sequence to sequence architecture

Addition

The simplest case of a sequence to sequence model is one where the inputs and outputs are the same length. One such model is provided in the Keras examples, featuring a sequence to sequence setup and using a recurrent neural network to learn the network to perform additions of two- or three digit numbers. Instead of the regular training in epochs, the model is iterated upon as is done with RNN. When running the script as is, an accuracy of 99.92% on the validation set is achieved after 200 iterations. The network appears to have learned to add numbers very effectively. In order to test this network a bit, we want to see how well it does on other similar tasks. For example, we can change the input to hexadecimal and see how well it does then. Even though for a human this would be harder to figure out, the RNN learned regular decimal addition from scratch, so it is expected to perform just as well. Sure enough, after 200 iterations of training an accuracy of 99.87% on the validation set is achieved.

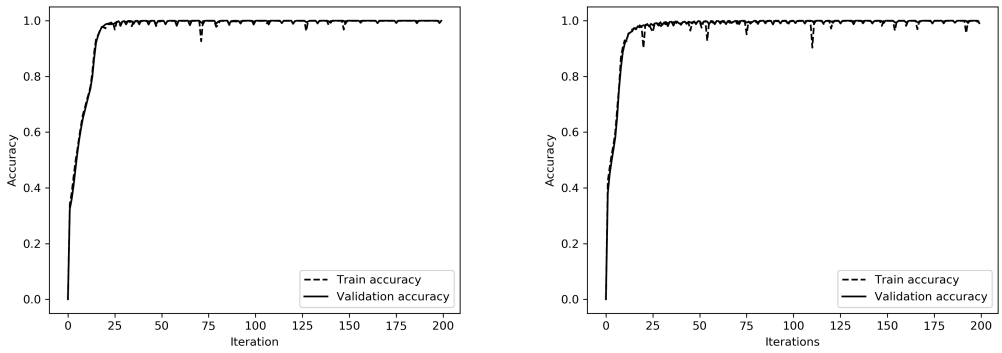


Figure 7: The accuracies over time of both the regular addition (left) and hexidecimal addition (right). Both follow the same structure, quickly climbing up to a high accuracy and then fluctuating over time. The argument might even be made that 50 iterations is sufficient for both models.

Translation

For translation process the "teacher forcing" techniques is used in the training process, it works by using the actual or expected output from the training data set at the current time step y(t) as input in the next time step X(t+1), rather than the output generated by the network.
Teacher forcing is a procedure [...] in which during training the model receives the ground truth output y(t) as input at time t + 1. — Page 372, Deep Learning, 2016.
So we will need 3 "input" , the encoder input, decoder input and decoder target (to be used for the teacher forcing" and for the inference process the algorithm turn samples from input into corresponding samples from decoder target. We ran the program on English-Dutch language I expected the results to be better since the test data is part of the training data, but we think this might be because the translator is character based, not word based. [2]

Task 3: Autoencoders

While recurrent and convolutional neural networks are all state of the art and are being implemented for many types of complex problems. Taking a step back in complexity, autoencoders one of the simpler architectures one could think of for a neural network, but nonetheless interesting to look at. An autoencoder has the architecture of multilayer perceptron, with an input layer, an output layer being the same size as the input layer, and one or more hidden layers that are always lower in dimensionality than the input and output layers. This architecture, though simple, can be very effective in some specific cases. Since the dimensionality of data is reduced dramatically reduced going through the encoder, only essential features will be learned and thus when properly trained, it can produce denoised versions of noisy input data similar to the training data. Another even more interesting application is to not care about the output

[2] https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

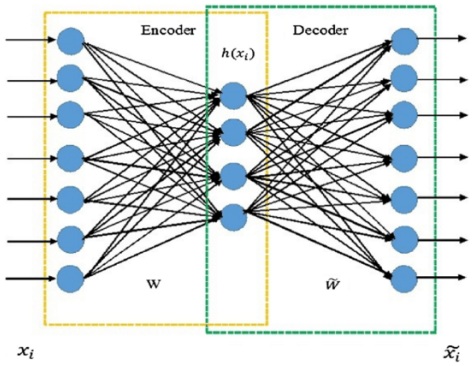


Figure 8: A graphical representation of an autoencoder with one hidden layer.

at all, but to simply look at the data in reduced dimensions. Since the autoencoder can transform highly dimensional data to only a couple of dimensions without the loss of essential information, complex data such as images can be reduced to a point in say, two-dimensional space and be compared to other similar images to see how they differ.

EMNIST Handwritten letters

Similar to the MNIST data set of handwritten digits, there is of course also a data set containing images of handwritten letters. This means that if you have an autoencoder for the MNIST data set, it will likely work on the EMNIST data set as well. Such an autoencoder is already present as a Keras example script, so feeding the EMNIST data set into it is a simple matter. The autoencoder in question reduces the dimensionality of the input significantly, from 784 from the input to only 2 dimensions in the encoded layer. This allows us to plot different letters and see how similar or dissimilar they are compared to each other. Making such a plot for all letters in the alphabet would make it a bit too chaotic, so we only plot the letters a-i. As can be seen in Figure 9 images of letters are not easily separable, at least in 2D

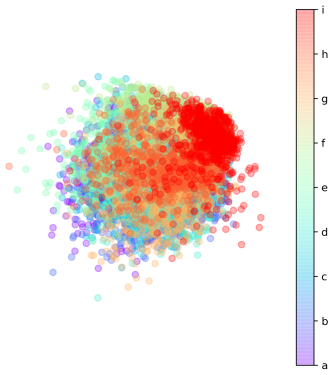


Figure 9: A 2D representation of the first ten letters of the alphabet.

space. From this we can infer that a network would probably have more trouble when trying to classify handwritten letters than when trying to classify digits.

Recommender

Another interseting application was a recommendor system using the denoising autoencoder This was implemented using tensorflow, noisy or corrupted basket with some items missing will be used as input in the training process. The model is a four layers denoising autoencoder model (i.e. 4 layers in encoder and

4 layers in the decoder). The soft sign activation function will be used in the first three layers of encoder and decoder, while sigmoid is used in the fourth. The binary cross entropy is used as a loss function that will be minimized using Adam. input is 169 the number of unique items in the dataset). Likewise, the number of neurons in first, second, third and fourth layers are 128, 64, 32 and 16 respectively. The dropout is introduced at the input layer with probability of 0.6 (i.e. dropping 40% of the input)[3]. Some of the recommendations are:
Items in basket: Instant food products, beverages, detergent, pickled vegetables, root vegetables
Recommended item(s): bottled beer, bottled water, canned beer, citrus fruit, newspapers, other vegetables, pastry,rolls/buns, root vegetables, sausage, shopping bags, soda, tropical fruit, whole milk, yogurt
Items in basket: frankfurter, rolls/buns, soda
Recommended item(s): other vegetables, rolls/buns, soda,whole milk
Items in basket: cream cheese, detergent, newspapers, processed cheese, tropical fruit
Recommended item(s): bottled beer, bottled water, canned beer,newspapers, other vegetables, pastry rolls/buns, root vegetables, sausage, shopping bags, soda, tropical fruit, whole milk, yogurt

Denoising images of text

As previously mentioned, one of the main interesting applications of autoencoders is denoising data. There are many types of problems one could think that would benefit from denoising data, and here we look at the problem of text written on paper, where the paper could have all kinds of noise added to it such as coffee stains and folding lines. This is a convolutional autoencoder, as is most effective when considering images, and contains two convolutional layers in both encoding and decoding part, the center layer being a convolutional layer as well, significantly compressing the data. The encoding part also has max pooling layers after each convolutional layer, and the decoding part has upsampling layer before each convolutional layer, to mirror the pooling layers of the encoding. Training this autoencoder for 10 epochs already gives a very decent result, as seen in Figure 10. All noise is removed, but at the cost of some clarity of the text.

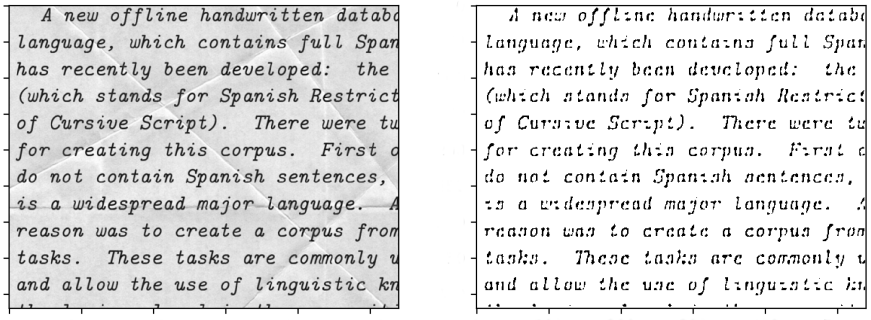


Figure 10: A noisy text image on the left and its denoised version on the right.

[3] http://aqibsaeed.github.io/2017-09-14-denoising-autoencoder-market-basket-cf/