Neural Networks - Assignment 2

Michelle Willebrands & Eva van Weenen

1 Introduction

The human brain is a fascinating network of neurons that is able to learn and memorize. This can be applied to driving cars, recognize faces, read handwritten texts and translating languages. However, the human brain has its limitations. We can only learn at a limited rate and sometimes the data set is so large that our brain cannot process it. It is therefore a logical step to let a computer do the learning. A computer is often much faster and has less limitations regarding the size of the data set. This learning cannot be done by simple programming. We must try to imitate the network of neurons in the human brain in order to be able to learn and memorize, thus programming a neural network.

Neural networks consist of various layers of nodes that are partially or fully connected. There is always an input and output layer and sometimes there are hidden layers in between. All nodes of consecutive layers are connected by weights. During the learning process these weights can be updated in order to obtain a model that resembles the data as much as possible. Most often, the weights are updated by minimizing the loss - the difference between the prediction of the model and the expected output (using data of the training set). Applying the model to the test set then gives a good indication of the accuracy of the network.

Neural networks can thus be designed in various manners. In this report, we will apply different types of neural networks to different applications. In sections 2.1 and 2.2 we will apply a multi-layer perceptron neural network (MLP) and a convolutional neural network (CNN) respectively to handwritten digits of the MNIST data set. In section 2.3 we will apply a recurrent neural network (RNN) to generate and translate text, as well as to carry out simple mathematical operations. And in section 2.4 we will test simple, denoising and variational autoencoders to the MNIST data set. The designs of these different networks will be briefly discussed at the beginning of each section, for readers that are unfamiliar with these types of neural networks. For our project we use several codes of the keras/examples, see [2]. More information about the code can be found in the acknowledgements.

2 Methods and results

2.1 Multi-layer perceptron (MLP)

Neural networks can easily be applied to recognize handwritten digits. In our project we use the MNIST data set ([3]) containing 70000 images of handwritten digits. Each image has 28x28 pixels and the MNIST data set can be split up into 60000 training images and 10000 test images. On the data set we apply two neural networks: a multi-layer perceptron (hereafter: MLP) and a convolutional neural network (hereafter: CNN). A MLP is a network that is fully connected and a CNN is only partially connected. We test the performance of both networks while using different loss functions and by permuting the input and compare the performances of both networks. We will analyze the results in the subsections below.

The first neural network we apply to our data set of handwritten digits is a MLP. This network is designed as follows: the network has an input layer of 784 nodes corresponding to the size of an image (28x28). The input layer is followed by two hidden layers each consisting of 512 nodes and both with activation function Rectified Linear Unit (ReLU). To both hidden layers we add a dropout of 20%, which disables 20% of all connections in order to prevent overfitting of the training data. The output layer consists of 10 nodes corresponding to our digit classes. The output is returned with the softmax activation function. The softmax activation function returns an array with the i^{th} element being the probability that the image is classified as digit i. The image is classified as the digit with the highest probability.

We apply our MLP to the training data and use a learning rate of $\eta = 0.001$, batch size of 128 and a total 20 epochs. We use a small batch size because feeding the computer all training images every epoch is too computationally expensive. We test the performance of our network using two different loss-functions: *categorical crossentropy* and *mean squared error*. Categorical cross-entropy is defined as:

$$H_{y'}(y) = -\sum_i y'_i \log(y_i) \tag{1}$$

where y_i is the predicted distribution and y'_i is the true distribution. Categorical crossentropy therefore compares the two softmax distributions returned and returns an indication of how much they overlap. The mean squared error is defined as:

$$MSE = \sum_{i=1}^{n} (y'_i - y_i)^2 \tag{2}$$

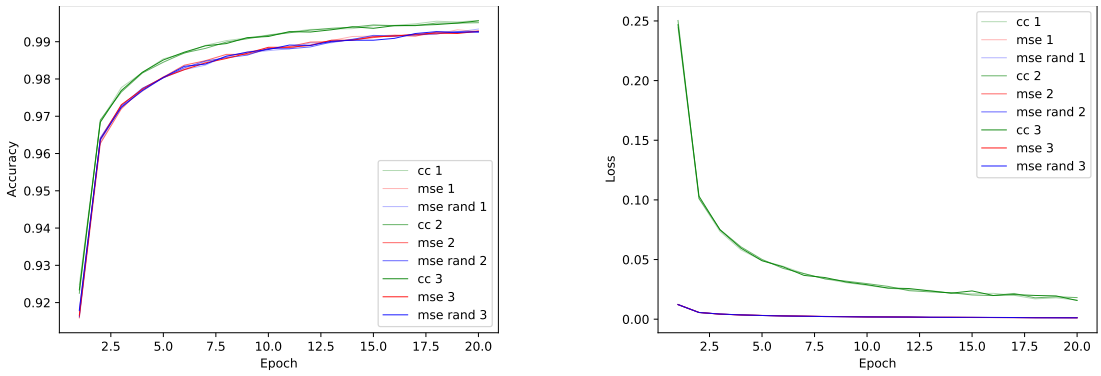
where y_i is the one-hot vector representation of the predicted classification and y_i' is the one-hot vector representation of the true classification. *One-hot encoding* of an image classified as digit i is a vector with value one at the ith element and zeros everywhere else. We will test the difference in performance of these two loss-functions.

Furthermore we test the performance of our network using a random permutation on all of our images, which is the same for all images. For each different initialization of the network we train the network individually three times to see how reproducible our results are. We expect that the performance of our network will not change if we permute all images in the same way, because a MLP is fully connected, meaning that all individual nodes of all consecutive layers are connected. Therefore it should not matter if an image is permuted or not.

The results that we obtain can be observed in figure 1 and table 1, displaying the performance of our network on the training and test set respectively. In both figure 1 and table 1 we display the loss and the accuracy of our network. The loss is an indication of how well the model corresponds to the data and is defined by the loss-function. We observe that the loss-function categorical crossentropy reaches the highest accuracy on the training set, and has the largest loss. However, from table 1 we observe that the differences between accuracies of the different loss-functions on the test set are minimal. These differences can be explained by the fact that categorical crossentropy is a better indication of difference between the true and predicted distribution than the mean squared error, as the categorical crossentropy compares complete probability distributions and the mean squared error only compares one-hot encodings. What is interesting to see is that there are small, but non-negligible, differences between the results of each run. In other words, the results are not exactly reproducible to all decimals that are provided in table 1.

When we compare the performance of our network on the original images with the performance of our network on randomly permuted images, we can conclude that there is no difference between the two. This can be observed by the overlapping red and blue lines in figure 1 for the training set and from table 1 for the test set, confirming our hypothesis.

Besides analyzing the loss and accuracy on the test and training set for our network, we also analyze the most misclassified images to verify whether our network is actually working. As we noted previously, for each digit class (one to ten) our network returns the probability that an image is classified as this digit. An image is classified as digit i when it has the highest probability of being in that class. A digit is misclassified when the classified digit does not correspond to the real value of the image. The *most* misclassified images are then the misclassified images for which the predicted digits have the highest probability. In these cases the network is very sure that this image should be the 'wrong' digit. These images are thus most often misclassified. As



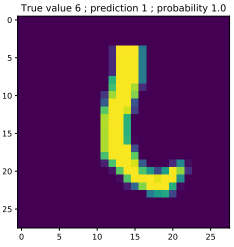
(a) Accuracy on the training data per epoch

(b) Loss of the training data per epoch

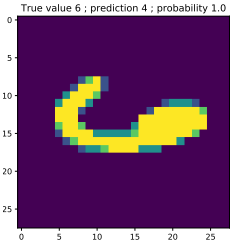
Figure 1: Performance of the MLP applied to the MNIST data set. Displayed are two figures containing the accuracy and loss per training epoch on the training set. Different colors indicate different initializations of training our network: *cc* loss-function categorical crossentropy; *mse* loss-function mean squared error; and *rand* randomly permuted images. We trained each initialization three times individually therefore the number behind the label (1,2 or 3) indicates the run.

Initialization		**Run 1**	**Run 2**	**Run 3**
Categorical crossentropy	accuracy	98.42%	98.29%	98.29%
	loss	$1.0547 \cdot 10^{-1}$	$1.0840 \cdot 10^{-1}$	$1.2082 \cdot 10^{-1}$
Mean squared error	accuracy	98.41%	98.19%	98.26%
	loss	$2.5612 \cdot 10^{-3}$	$3.0080 \cdot 10^{-3}$	$3.0084 \cdot 10^{-3}$
Mean squared error & permutation	accuracy	98.50%	98.41%	98.19%
	loss	$2.5702 \cdot 10^{-3}$	$2.6580 \cdot 10^{-3}$	$3.0486 \cdot 10^{-3}$

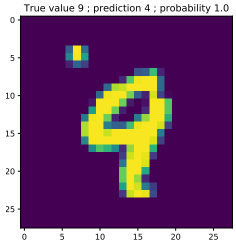
Table 1: Performance of the MLP applied to the MNIST data set. The accuracy and loss on the test set are displayed for three individual training runs.



(a) Image 2654 has a true value of 6 and is five times misclassified as 1 with a probability of 100%



(b) Image 3520 has a true value of 6 and is five times misclassified as 4 with a probability of 100%



(c) Image 9587 has a true value of 9 and is three times misclassified as 4 with a probability of 100%

Figure 2: The three most misclassified images for all the different initializations of the MLP.

we ran all three different initializations three times and each time returned the three misclassified images with the highest probability, leading to a total of 27 most misclassified images, we will only show the images that occurred more than twice as most misclassified images.

These images are shown in figure 2. We observe that these most misclassified images that are difficult to identify for a neural network, are difficult to identify for the human brain as well, as these digits are very illegible.

2.2 Convolutional Neural Networks (CNN)

In order to improve our the classification of the MNIST data set we experiment with using a CNN. With the MLP we have used layers that are fully connected, meaning that every node has a separate connection to the input layer. However, this is not the best method to classify images as it does not take into account features and structure of the images. A MLP treats pixels that are far apart and close together in the same manner and is thus not able to identify for example a horizontal line in the image. A CNN, however, is able to detect spatial structures, because it connects groups of nodes of the input layer to individual nodes of the hidden layer as can be seen in figure 3. A filter of for example 5×5 pixels with a certain pattern slides over the entire image, selecting these groups of pixels. The group of pixels is correlated with the filter, which yields a high value in the convolutional layer if the pixel values overlap well. CNNs reduce the number of weights compared to MLPs and are in this way more efficient. Furthermore, a result of reducing the number of weights is that it also solves the vanishing and exploding gradient problems.

We initialize our network as follows. The input layer has 28x28 nodes, the size of our image. The first hidden layer is a convolutional layer. It has 32 nodes, each mapped by kernels of size 3x3 on the input layer and has activation function Rectified Linear Unit (ReLU). We add a second convolutional layer but now with 64 input nodes. The kernel size remains 3x3 and it has a ReLU activation function. The next layer is a pooling layer with pool size of 2x2. Pooling reduces the number of parameters in the model, because it is a form of down-sampling. The 2x2 pooling filter is moved over the previous layer and selects only the pixel with the maximum value within the area. Effectively, this discards 75% of the pixels. Consecutively, we add a dropout of 25%, disabling 25% of all connections to prevent overfitting. Then, we flatten the output, converting a 2D matrix to vector form and add a fully connected layer of size 128 and with a ReLU activation function with a dropout of 50%. The final output layer has 10 nodes and a softmax activation function, which returns the probability of an image being in each class.

We apply our CNN to the MNIST data set with the same initializations as the MLP, except for the randomly permuted initializations. For randomly permuted images we use categorical crossentropy instead of the mse as loss-function, as we noticed it had a much better performance regarding the accuracy. Furthermore, we reduced the number of epochs to 12. The results are shown in figure 4 and table 2. We observe that even with reducing the number of epochs, CNNs are able to reach a higher accuracy and are thus better at classifying handwritten

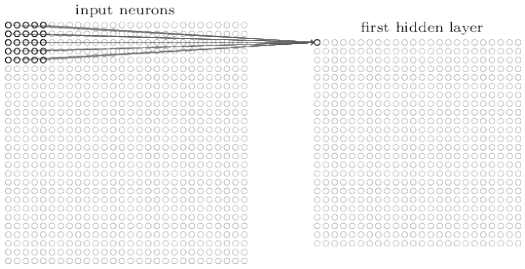
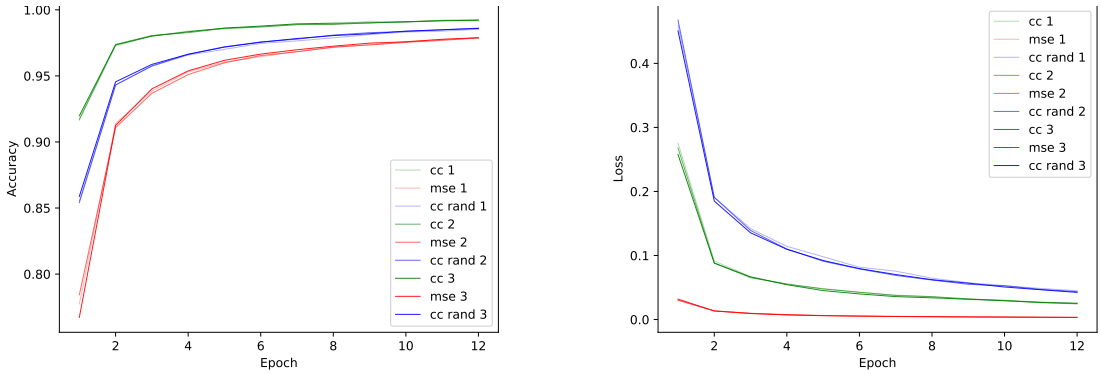


Figure 3: In a CNN a filter of fixed size connects groups of input nodes to the nodes of the hidden layer [4].



(a) Accuracy on training data per epoch

(b) Loss of training data per epoch

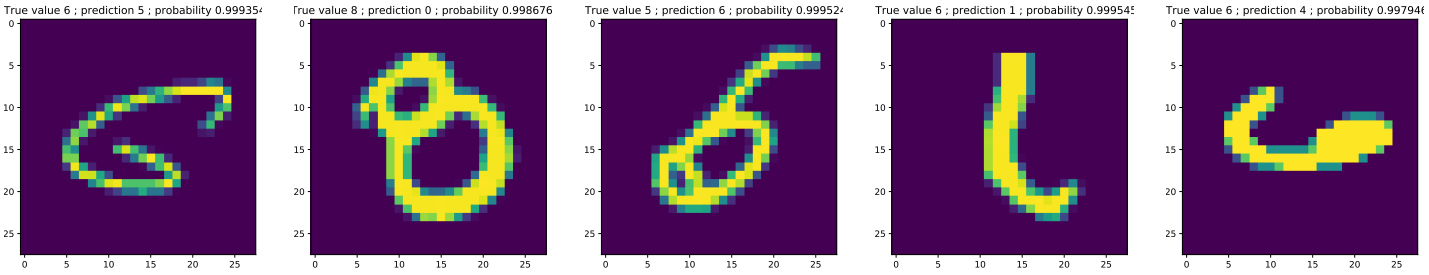
Figure 4: The performance of the CNN applied to the MNIST data set. Different colors indicate different initializations of training our network: *cc* loss-function categorical crossentropy; *mse* loss-function mean squared error; and *rand* randomly permuted images. We trained each initialization three times individually, which is indicated with the number behind the label (1,2 or 3).

Initialization		**Run 1**	**Run 2**	**Run 3**
Categorical crossentropy	accuracy	99.14%	99.30%	99.10%
	loss	$2.7924 \cdot 10^{-2}$	$2.4655 \cdot 10^{-2}$	$2.8034 \cdot 10^{-2}$
Mean squared error	accuracy	98.76%	98.67%	98.60%
	loss	$1.9290 \cdot 10^{-3}$	$1.9486 \cdot 10^{-3}$	$2.0309 \cdot 10^{-3}$
Categorical crossentropy & permutation	accuracy	97.92%	98.04%	97.89%
	loss	$7.9852 \cdot 10^{-3}$	$6.8932 \cdot 10^{-3}$	$8.2377 \cdot 10^{-3}$

Table 2: The performance of the CNN applied to the MNIST data set. The accuracy and loss on the test set are displayed for three individual training runs.

digits. Again, there are small deviations between the results from each run. From figures 4a and 4b we observe that the loss-function categorical crossentropy has the highest accuracy on the test and training set and also the highest loss. We observe that the mean squared error loss-function has a much lower accuracy and the lowest loss in fact. This can again be explained by the difference in loss-functions as mentioned in the previous section. With a MLP we observed that it made no difference whether the images were randomly permuted or not, as a MLP is fully connected. However, for a CNN we observe that the performance is different for randomly permuted images. The accuracy of the network is a few percent lower and the loss is higher. As we mentioned earlier, a CNN is not fully connected and mainly looks at features of the image. As all features are now distorted we would expect the accuracy to be ∼20% lower. However, somehow the CNN is able to make sense of these local distortions and is still able to learn, therefore reaching an accuracy only a few percent lower.

As we did for the MLP, we show the most misclassified images. These images are misclassified images with the highest probability of being misclassified. As we have performed many runs with our network, we only show the images that occurred more than two times as most misclassified in our runs. The images are shown in figure 5. We see that there is an overlap of two images between MLPs and CNNs. The first three images in the figure are new. These images are difficult to identify by the human brain as well.



(a) Image 1014 has a true value of 6 and is four times misclassified as 5 with a probability of 99.9%

(b) Image 2896 has a true value of 8 and is three times misclassified as 0 with a probability of 99.8%

(c) Image 9729 has a true value of 9 and is three times misclassified as 4 with a probability of 99.9%

(d) Image 2654 has a true value of 6 and is three times misclassified as 1 with a probability of 99.8%

(e) Image 3520 has a true value of 6 and is five times misclassified as 4 with a probability of 99.8%

Figure 5: The five most misclassified images in the MNIST data set for the CNN.

2.3 Recurrent Neural Networks (RNN)

Recurrent neural networks (hereafter: RNN) can be used to recognize and predict patterns in sequences of data. We can apply a RNN for example to videos, text translation, music composition and handwritten texts. In these examples, fragments of the data depend on fragments of data at previous timesteps. A RNN is able to store previous fragments of data in a feedback loop. The input of a layer of a RNN at timestep t consists therefore of the input x_t and a hidden state h_{t-1} containing information of a previous timestep $t-1$ (see figure 6a). In this way, the decision of a state at a previous timestep $t-1$ affects the decision at later time step t and each decision is therefore indirectly related to all previous decisions. The feedback loop is thus comparable to memory in the human brain. Training a RNN is therefore done by *backprogpagation through time*.

Long Short-Term Memory units (LSTMs) are able to select what information from previous timesteps they want to keep. A LSTM consists of many cells and each cell decides which information from previous timesteps they want to store and which information is thrown away. An overview of a cell of a LSTM can be viewed in figure 6b. The information from previous timesteps that is considered useful by the cell at the forget gate is stored in the cell state C_t - the "long term memory". Each cell receives information from the hidden state h_{t-1} - the "short term memory" - which is the direct output of the previous step $t-1$. At the input gate, the important information from the hidden state is selected and later added to the cell state. The cell state is then propagated to the next cell (C_{t+1}). Using the information from the current input x_t, hidden state h_{t-1} and cell state C_t, the output of the cell is calculated using various activations functions and propagated by the output gate to the hidden state of the next cell h_t. In other words, a LSTM unit is able to recognize important inputs, store it in its long-term memory for as long as needed and learn use it when needed. In this way, an LSTM cell is able to learn over many time steps and able to recognize long-term patterns in time series.

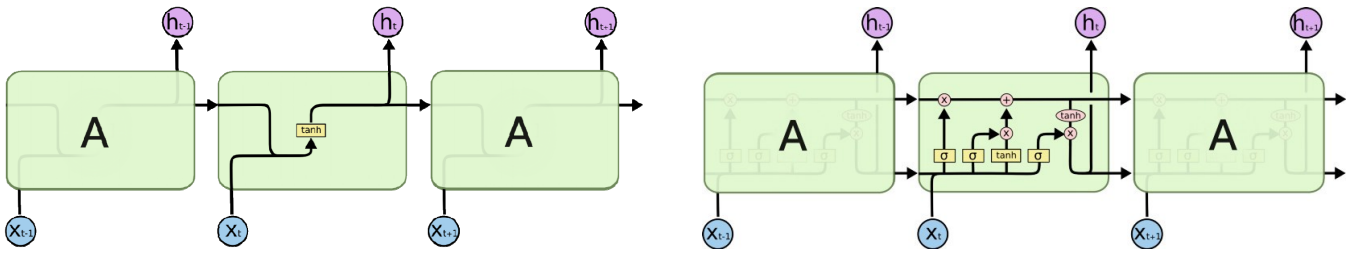
RNNs can be used for predicting value(s) from sequences of data, such as predicting the score of a movie (value) from a movie review (sequence). RNNs can also be used vice versa: predicting the caption of an image (sequence) from an image (values). Converting a sequence to a value is called *encoding* and converting a value to a sequence is called *decoding*. These terms will be useful in the next subsections.

2.3.1 Text generation

We will begin by applying a RNN to text generation. We will train a LSTM cell on a body of text written by Nietzsche and the network will then generate entirely new sentences based on this text. These new sentences should be in the same style, plausibly written by Nietzsche himself. This process is called language modeling and it is a generative model. The text for training the network is obtained from [1] and contains 600893 characters. The entire character length of the text is called the *corpus length*. There are 57 unique characters in this corpus - the *vocabulary*. In the learning process, a fixed sequence length of 40 characters is used in order to learn the subsequent 41^{th} character. This window slides over the whole text, character-by-character, to learn each character from the 40 characters that precede it. Prior to starting the training process, the data needs to be converted to a generalized representation. For this representation, we used one-hot encoding, meaning that each character of a sentence is represented by a vector of the vocabulary-length with zeros everywhere and a one where the character equals the character of the vocabulary.

After 60 epochs of training, the network is fed a sequence of 40 characters randomly chosen from the text and produces a short body of text based on this sequence. This output depends on a variable called the *diversity*. When this variable has a low value, the network is less likely to sample from low probability candidate characters. 'Low probability candidate characters' here means a low probability in the outcome of the softmax activation function [6]. This results in the text being more correct but also less diverse. On the other hand, a high diversity value results in a more diverse output, but also results in more mistakes being made.

The outputs were both generated with the seed: *"d accomplishing her mission therein–he "* and the final loss



(a) Recurrent Neural Network (RNN)

(b) Long Short-Term Memory unit (LSTM)

Figure 6: Overview of cells of a RNN and of a LSTM

was 1.3. To show the difference in generated output, we show the results of the diversity parameter set to 0.2 and 1.2. With the diversity set to 0.2, the output of the model is:

> *d accomplishing her mission therein–he will be our own things and conscience and the morality of the present, and the conscience of the probably and the more standards and exceptions the conscience of the man the pression and exception of the present strength, and a skeptic will to the case of the case of the spirituality and standards and stands and the spirituality of the conscience and devil, and the morality in the extender of the*

When we change the diversity to 1.2, the produced text is:

> *d accomplishing her mission therein–he happosscrevoled near bst. poingouncher two bedocp notioning lifd toas" itleve europeans! with our scavent est spucuy learniss fellyp trees and more aspect from hisdenior for a regard it not almost preidkeent, the "by at gremanm through by inversties or fivally bra.". elot everyth inbelic knowledge to liisternt, men maolutity by this s"qual uncroppgisch: once more curious impetty, to it for maladu*

In the first output text, almost exclusively real words are generated, that are indeed in the same style as Nietzsches writings. However, when the diversity is larger, the result becomes nonsensical.

We run the code again, but now we use a different text, namely from the children's book "A dog with a bad name" by Talbot Baines Reed. We obtained this text from [5] and removed all the excess white spaces. The style of this text is clearly very different from Nietzsches writings, so we should be able to distinguish these results. The corpus length of this text is 530012 and it contains 77 unique characters. After the same amount of epochs, the final value of the loss was 1.2 and the model now generates text using the seed: *"r Rimbolt, he decided to resist at all h"*. With a diversity of 0.2, the output of the model is:

> *r Rimbolt, he decided to resist at all he was a shout of the street of the stards, and seemed to his head, when the boy was the sign to see his ward was a sorry and seemed to his head, when the service of the sound and seemed to his eyes of the second of the train of the reply to the sound of the second to the sound and the boy was a shosJedding to his shoulders to be the servant. "I have hard he went of the street of the stumes the ser*

With a diversity of 1.2 the produced text is:

> *r Rimbolt, he decided to resist at all hours. duar? Musters notchisting, and at heads from it. Brite the ruit of small, and Forrester's weam, but Jeffreys is Mo; " footing him so, effors of old one or along terribly to his mind fallent to him awid." "I head–John' eageh. He wrighed to engiled that and weuld Jeffreys was rorm till ehe? Look oun thing at onceposited, provinde. His enk was pleashing, and aboyt down time has too, not must b*

We observe that even for the lowest diversity value, the network is not able to generate exclusively real words and the text is nonsensical. This is worse for a higher diversity.

For both texts, the sentences are grammatically incorrect and make little sense. For a higher diversity, the output becomes worse and contains many nonexistent words. The writing style, however, is captured to some extend.

2.3.2 Text translation

Our next application of a RNN is translation. We create a character-level sequence-to-sequence model, in order to translate English text to French. The network processes the English input sentences character-by-character and generates the French output sentences character-by-character as well. Note that this is not word-by-word translation as one would intuitively think; the network inspects each sentence character-by-character. Each sentence is then one-hot vectorized, just as we did for the Nietzsche text generation. One layer of this RNN acts as an encoder that processes the input data (the English sentences) and returns a vector to the 'decoder', which is another layer in the model. The decoder will output French translations. As we mentioned, the networks looks at the text character by character. Therefore, the decoder will be trained to predict the next character of a sentence, given what the previous characters are. Besides receiving input from the encoder, the decoder also receives information about what text it was supposed to generate at a previous timestep.

Once the training is completed, the performance of the model is tested in the inference mode, in which unknown input sentences are decoded. This unknown input is the validation data set. The first step of this process is to encode an input sentence and retrieve the initial state of the decoder. One step of the decoder is carried out with this initial state and a 'start of sequence' token as target. This produces a prediction for the next character of the sequence, which is subsequently appended to the target sentence. These steps are repeated until a fixed maximum number of characters is reached.

The English-to-French data set we use for translating consists of 10.000 pairs of English and French sentences. In the English sentences, there are 71 unique characters and in the French sentences there are 94 unique

characters. The longest English sentence is 16 characters long, whereas for the French ones this is 59. The network is trained on 8000 samples and validated on the remaining 2000. We used a number of 100 epochs for learning and some examples of the inference loop are shown in table 3. These sentences are selected from the data set at random and indicate how well the network performs. After the training process has finished, the loss has reached a value of 0.0576. The accuracy is difficult to determine as some sentences are correctly translated for 50% and some words in translations are completely nonsensical.

Input sentence	**Decoded sentence**	**Target sentence**
Get in the boat.	Grimpez dans la camionnette !	Grimpez dans le bateau !
I didn't shower.	Je n'ai pas dormi.	Je n'ai pas pris de douche.
Ask Tom.	Demande à qui que ce soit !	Demande à Tom.
I feel terrible.	Je me sens bien.	Je me sens super mal.
I was sick.	J'étais en train de parler.	J'étais malade.
Hug me.	Serre-moi dans tes bras !	Serre-moi dans tes bras !
He dislikes me.	Il éprouve de l'antipathie à mon égard.	Il éprouve de l'antipathie à mon égard.

Table 3: Examples of sentences from the inference loop, translated to French. The decoded sentence is produced by the neural network and the target sentence is the actual translation.

As can be seen in table 3, the decoded sentences are not perfect in all cases. In these sparse examples, only the last two sentences are translated properly. In the other examples, the personal pronoun and the first verb (e.g. 'I was') appear to translate well, whereas the remaining part of the sentence causes problems. Of the 100 samples printed in the inference loop, approximately half of the sentences seem to be translated well. Note that this is a very rough estimate.

We would like to examine whether translation to a different language would alter the outcomes significantly. French and English belong to different linguistic groups, whereas Dutch and English are both part of the West Germanic language family. Therefore, we implement this change of target language in the code to see whether this modifies the quality of the translations. The English-to-Dutch data set consists of 10.000 pairs of sentences, equivalent to the English-to-French data set. In the English sentences, there are again 71 unique characters and in the Dutch sentences there are 78. The longest English sentence is 24 characters long, whereas for the Dutch ones this is 58. The number of epochs is set to 100 for consistency with the previous experiment. The achieved loss has a value of 0.0876.

Input sentence	**Decoded sentence**	**Target sentence**
It never happened.	Het is niet belangrijk.	Het is nooit gebeurd.
No one is perfect.	Niemand is gewond geraakt.	Niemand is perfect.
I sort of understand.	Ik heb haar een pop gezonden.	Ik begrijp het min of meer.
Choose a dress you like.	Kopparijk zijt dit ik.	Kies een jurk die je bevalt.
Playing baseball is fun.	Geweldigen jeider in het huisverhod.	Het is leuk om honkbal te spelen.
He knows where we live.	Hij ontwoodde de buist.	Hij weet waar we wonen.
That wasn't very nice.	Dat was niet zo aardig.	Dat was niet zo aardig.

Table 4: Examples of sentences from the inference loop, translated to Dutch. The decoded sentence is produced by the neural network and the target sentence is the actual translation.

As becomes clear from table 4, the performance of the network is low. There are multiple problems with the translations created by the network: in some of the samples, the translations deviate substantially from the target; this is the case in the first three examples provided in table 4. Moreover, in many cases the decoded sentences contain nonsensical 'words' (e.g. *kopparijk*, *jeider* or *ontwoodde* in the table). Only in a modest fraction of the samples the sentences are translated properly.

It is clear that the translation from English to French works better than translating to Dutch, even though it is far from perfect. It is not clear why the translation to Dutch performs so much worse. Perhaps more epochs are required to achieve better results. On the other hand, the validation loss was already increasing again after ~ 50 epochs, so increasing the amount of epoch might not prove to be a solution a all. In both cases, we would expect word-by-word models to perform better than the character-by-character model.

2.3.3 Integer addition

In our last application we will examine if we can apply a RNN for learning to add short integers. We generate random integers with maximum length of three and train our network on the outcomes of these additions. The size of our training samples is 45000 and the size of our test samples is 5000. In training our network we use 1

epoch and 100 iterations. In each iteration we print 10 random sample calculations and we check them. After 15 iterations, the accuracy of the network on the training set is quite good, over 90 %. After a total of 100 iterations, the accuracy on the training set is 100%, the loss on the training set is $6.6939 \cdot 10^{-4}$; the accuracy on the test set is 99.98% and the loss on the test set is $1.7 \cdot 10^{-3}$.

We examine whether the network is able to perform equally well if we change our operation from addition to multiplication. We use a larger number of iterations for our network, namely 500. Besides, we use smaller numbers of maximum length two and now only use 5000 generated samples, of which 10% is used for the test set and 90% is used for the training set. We have to decrease the number of samples, because with numbers with a maximum length of two digits, there are simply less possible combinations. We find that after 500 iterations the accuracy on the training set is 100%. However the accuracy on the test set remained 62.75% and thus the performance is not good. For simple multiplications such as $0 \cdot x = 0$ the network seems to find the correct answer, however for large multiplications the network is more often incorrect. It is able to estimate the correct size of the target number and often the last and first digit is correct but all digits in the middle seem to be troublesome. We can only speculate on the causes, namely that multiplication is much more difficult than addition and also leads to many more possible outcomes. For example, the addition of two numbers with maximum length two has in total 200 possible outcomes while the multiplication of two numbers with maximum length two has in total a little less than 10,000 possible outcomes. Therefore, training for a longer number of iterations might help solve this problem.

2.4 Autoencoders

Autoencoders are neural networks that are able to compress data from their input layer and later decompress it into something that closely resembles the original data. These compressions are called *codings*. The training set is unlabeled and therefore autoencoders use unsupervised learning. Autoencoders are useful for dimensionality reduction and they can be applied in generative models.

2.4.1 Simple autoencoder

We first generate a simple autoencoder and apply it to the MNIST data set. We use a simple autoencoder first as a sort of 'control group' to be able to quantify and compare the results of the next experiments with. Our autoencoder consists of an input layer with the shape of the image; an encoding layer of size 32 that is fully connected and has a ReLU activation function; and a decoding layer of the size of an image, fully connected and with a sigmoid activation function. The encoding layer is a compressed representation of the input, and the decoding layer is a reconstruction of the input. For training our model we use the binary crossentropy loss-function.

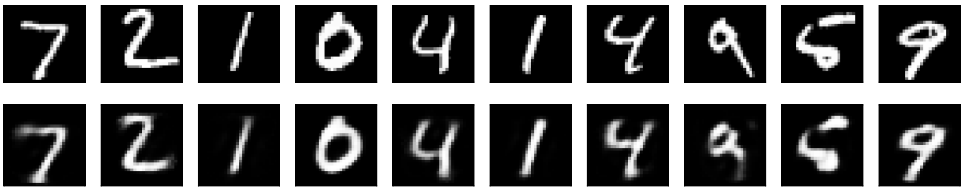


Figure 7: Results of a simple autoencoder applied to the MNIST data set. The top row shows the original images and the bottom row shows the output of the autoencoder - the compressed images.

After 50 epochs, we find a loss of 0.1 on both the training and validation set. The reconstructed images are shown in figure 7. We observe that the original images are somewhat retrieved, however the lines are sometimes more vague and less discrete. This is in line with our expectations, as we cannot expect the output of the autoencoder to be exactly equal to input that was compressed in the process.

2.4.2 Denoising autoencoder

Secondly, we apply a denoising autoencoder to the MNIST data set. A denoising autoencoder adds noise to its input and in this way tries to recover an image with less noise. This prevents the autoencoder from simply copying inputs to outputs. Our encoder consists of two convolutional layers and our decoder consists of two convolutional layers as well, both with the ReLU activation function. Furthermore, we generate the noise from a Gaussian distribution.

After 30 epochs, we find a loss of 0.015 on both the training and the test set. The reconstructed images are shown in figure 8. We observe that the original images are quite well recovered and the network is really able to learn features of the data.

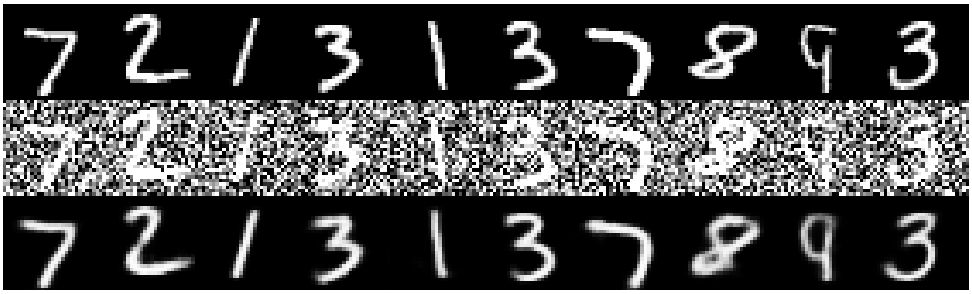
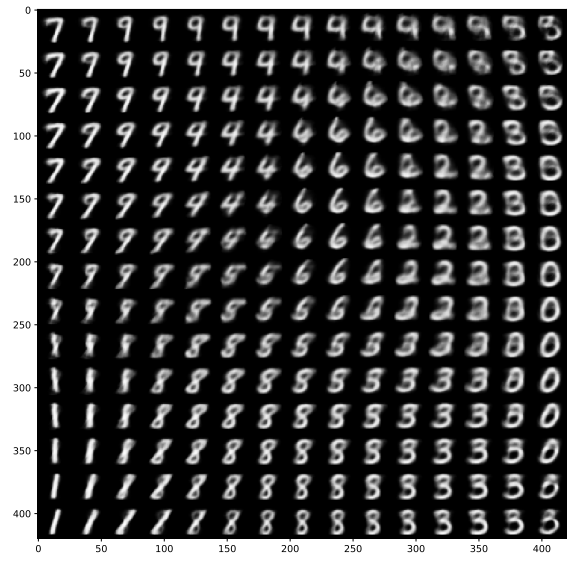
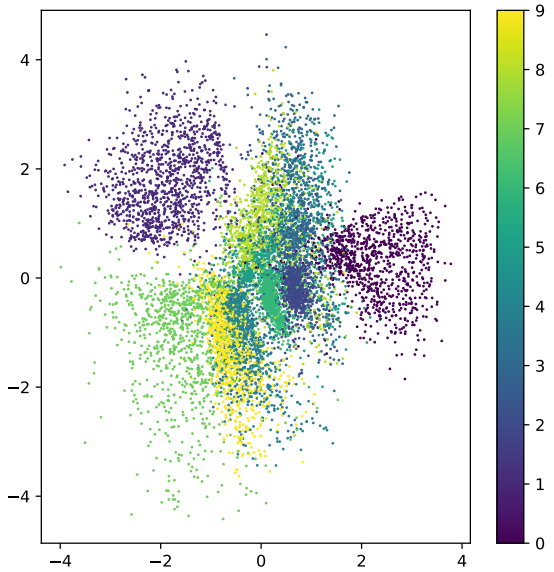


Figure 8: Results of a denoising autoencoder applied to the MNIST data set. The top row shows the original input, the middle row shows the input with noise added to it and the bottom row shows the recovered input by the autoencoder.



(a) Range of generated digits by the variational autoencoder based on different latent points.



(b) Latent space representation by autoencoder of digits in MNIST data set

Figure 9: Results of variational autoencoder applied to the MNIST data set.

2.4.3 Variational autoencoder

Lastly we apply a variational autoencoder to the MNIST data set. Instead of directly autoencoding from a given input, the autoencoder learns the parameters of the probability distribution that models the data. These parameters of the distribution, μ and σ are given in *latent space*. From this latent normal distribution, the encoding input is then randomly sampled. The decoder maps the parameters in latent space back to the original input data. In this way a variational autoencoder is a generative model. This also means that the outputs are determined by chance even after training.

After 50 epochs, the loss on the training and test set is approximately 150. The parameters of the probability distribution learned from the input can be plotted in a two dimensional plot, see figure 9b. Each color indicates a different digit cluster. Digits that are close have overlapping distributions and are structurally similar. The digits that are generated by the autoencoder at different latent points can be viewed in figure 9a.

3 Conclusion & Discussion

For classifying the MNIST data set we find that the categorical crossentropy loss-function has a better performance for both a MLP and a CNN. This can be explained by the fact that the mean squared error is the difference between one-hot vectors of the images and is thus not a perfect representation of the images. The categorical crossentropy examines the difference between the distributions and is therefore a better estimate of the loss. A second result we find from classifying the MNIST data set, is that CNNs have a better performance in classifying digits than MLPs. CNNs are able to look at features and spatial structure of the images and can thus reach a higher performance. When we randomly permute all images in the same way, we find that for a MLP there is no difference in performance but for a CNN the performance decreases by approximately 1%. As we mentioned, CNNs investigate features of the data and by randomly permuting images, these features are removed. We would therefore expect the performance to decrease by approximately 20%. We see that this

is not the case therefore we think the CNN is still able to recognize local patterns and can somehow create some understanding of these patterns. The most misclassified digits that the neural networks cannot classify are illegible for the human brain as well.

The first RNN that we used generated text in the style of Nietzsche and a children's book called "A dog with a bad name". Depending on the value of the diversity variable, the generated output text contained predominantly real or nonexistent words. In all cases, the sentences in the output were grammatically incorrect, but did capture the distinctive writing style of the original text. With the second RNN we tried to translate English sentences to French and Dutch. The result for French was significantly better than for Dutch, even though we expected it to be the other way around based on the linguistic relation between the languages. The French output consisted of real words, but not always the correct ones, whereas the Dutch translations contained many nonsensical words. The losses after training were 0.0576 and 0.0876 for French and Dutch respectively. The third and final RNN was designed to learn how to add a pair of integers. For this, it achieved an accuracy of 99.98% on the validation data and a loss of 1.7×10^{-3} after 100 iterations. Then, we changed the mathematical operation from addition to multiplication. The network had difficulties learning this and after 500 iterations the accuracy was only 62.75%. However, the predicted values that the model provided were close to the real values in most samples.

We applied three autoencoders to the MNIST data set. The first autoencoder was a simple autoencoder with the smallest representation a vector of size 32. The recovered images resembled the original images. The detailed structures in the images were lost. The second autoencoder we applied was a denoising autoencoder. By adding Gaussian noise to the input, the denoising autoencoder was able to learn the features of the data and able to recover images that looked like the original images. The third autoencoder was a variational autoencoder that compressed the data into probability distributions and in this way was able to retrieve original images as well.

4 Acknowledgements

For our report we have used code of the keras/examples github. We have used the following codes: *mnist_mlp.py*, *minst_cnn.py*, *lstm_seq2seq.py*, *lstm_text_generation.py*, *addition_rnn.py*,

References

[1] Data set nietzsche. https://s3.amazonaws.com/text-datasets/nietzsche.txt. Accessed: 2018-04-12.

[2] Keras-team. keras/examples/. https://github.com/keras-team/keras/tree/master/examples, 2018. Accessed: 2018-04-03.

[3] Y. Lecun et al. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2018-03-12.

[4] M. Nielsen. Deep learning: Chapter 6. http://neuralnetworksanddeeplearning.com/chap6.html, 2017. Accessed: 2018-04-13.

[5] PBWorks. Demo corpora (text collections ready for use): A dog with a bad name. http://dhresourcesforprojectbuilding.pbworks.com/w/page/69244469/Data%20Collections%20and%20Datasets, 2013. Accessed: 2018-04-18.

[6] J. Shenk. What is temperature in lstm (and neural networks generally)? https://cs.stackexchange.com/questions/79241/what-is-temperature-in-lstm-and-neural-networks-generally/79242#79242, 2017. Accessed: 2018-04-18.