Neural Networks 2018

Assignment 2

Group 20

April 18, 2018

1 Introduction

Assignment 2 of the course Neural Networks is centered around Keras[1]. Keras is a deep learning library for Python, which runs on Tensorflow, CNTK or Theano (see Abadi et al., 2016; The Theano Development Team et al., 2016; Yu et al., 2014). In this report, we will guide the reader through the different possibilities that Keras offers, by introducing four different types of networks in three different tasks. Task 1 will focus on multi-layer perceptrons and convolutional neural networks. Task 2 will contain three different applications of recurrent neural networks. Finally, in Task 3, we will examine three different types of auto-encoders. More in-depth information about these different networks will be given in the respective chapters.

2 Task 1

We start by running the example *mnist_mlp.py* from the Keras examples Github[2]. This trains a simple MLP on the MNIST training data[3] (60,000 examples) and verifies it on a separate test set of 10,000 examples. The architecture of the network is comprised of three layers: the first layer has input shape 784, output shape 512, a dropout rate of 0.2 and uses the *relu* activation function. The second layer has input shape 512, output shape 512, the same dropout rate and activation function. The final layer is a dense layer with 512 inputs and 10 outputs, corresponding to the 10 digits, with a *softmax* activation function. The model is then trained by using the *categorical crossentropy* as the *loss* function and the *RMSprop* as the optimizer (e.g. Kurbiel and Khaleghian, 2017). With a batch size of 128, and after only 20 epochs this already achieves an accuracy of 0.983 on the test set. But this result is dependent on the initial configuration of weights, which is why we have trained 50 separately initialized networks with the same parameters (but different initial randomized weight values). The results of this endeavour can be viewed in Figure 1.

It turns out that in Keras it is nearly impossible to set a random seed to make exactly reproducible results, because it is using multiple seed generators for the initialization of the network. This means that our results are not reproducible up to the last available decimal, unless we save the initial network made by the script. But as is apparent from Figure 1, this is not necessary since the accuracy does not vary that much per run. The mean accuracy after 50 runs is 98.22% and the standard deviation is 0.1281%, so there is very little deviation in the final accuracy.

Next, the most misclassified digits will be defined and examined. Defining most misclassified digits is an abstract task that can be approached from different angles. For this task we consider a digit (0-9) most misclassified, when it differs significantly in performance with respect to the other digits. More concretely, the digits that have the lowest precision will be the most misclassified digits. We define precision as is shown in the following formula:

$$precision = \frac{tp}{tp + fp} \tag{1}$$

where tp is the number of true positives, the digits that are classified correctly. fp are the false positives, which are the digits that are assigned to the class, while they actually are not part of the class. For example, when defining the precision on digit 7, the true positives are the digits that are assigned to be a 7 and are actually a 7. And the false positives are the digits that are assigned to be 7 while they are actually not a 7.

[1] https://keras.io

[2] `https://github.com/keras-team/keras/blob/master/examples/mnist_mlp.py`

[3] http://yann.lecun.com/exdb/mnist/

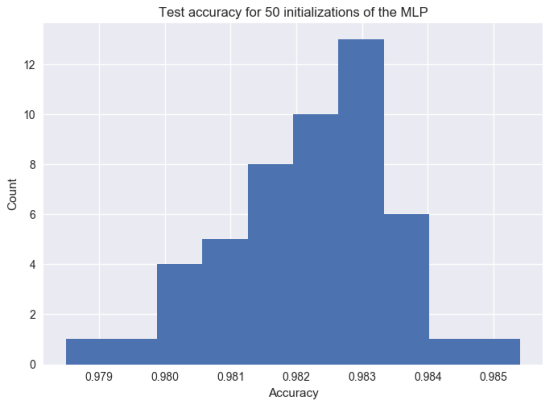


Figure 1: Accuracy on the test set for a multi-layer perceptron as described in the text, after 50 different initial networks are trained.

In addition we introduce two other evaluation measures to quantify the misclassified digits. These measures are the *recall* and $F_{measure}$. *Recall* is defined as

$$recall = \frac{tp}{tp + fn}, \tag{2}$$

in which the fn is the number of false negatives, i.e. the digits that aren't classified as a 7, while they are actually a 7.

Equation 3 is used to calculate the $F_{measure}$ which is the harmonic mean of the *precision* and *recall*.

$$F_{measure} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3}$$

Table 1 shows the results of these three measure in order to quantify the correctly classified digits in our experiment. As it can be seen, for each of the three measures a different digit is classified correctly. The most accurate classified digit according to the *Precision* measure is digit 9, for *recall* is digit 1 and for F1-score is digit 4. However, the overall average accuracy for all the three measures is equal to 99.15%

Table 1: Three different measures to quantify the misclassified digits, namely *Precision, Recall* and *F1-score*.

Digit	**Precision**	**Recall**	**F1-score**	**Support**
0	0.98987	0.99694	0.99339	980
1	0.99124	**0.99648**	0.99385	1135
2	0.98934	0.98934	0.98934	1032
3	0.99111	0.99307	0.99209	1010
4	0.99289	0.99491	**0.99390**	982
5	0.99325	0.98991	0.99158	892
6	0.99371	0.98956	0.99163	958
7	0.99316	0.98833	0.99074	1028
8	0.98573	0.99281	0.98926	974
9	**0.99498**	0.98315	0.98903	1009
Average/Sum	**0.99151**	**0.99150**	**0.99150**	**10000**

We can also check the effect of applying the MLP and CNN to a random permutation of the data. Our hypothesis is that for an MLP the result would not change, since this takes all pixels as input values and their position is not relevant while they are propagated through the network. However, the convolutional layer will try to identify features from groups of pixels which, if we scramble the pixels, will disappear. Meaning that the convolutional network will probably perform worse when applied to a random permutation of the data. The permutation that we have used is simply rearranging the 784 pixels in a random way for every example.

As can be viewed from Figure 1-4, the accuracy does not change significantly with the MLP network (<0.2 %), but it does change for about 2 percent for the CNN. This is much less than we had initially expected, so it seems that the feature selection is actually a smaller part of the effectiveness of the network that we initially thought. Most likely, the high amount of tune-able parameters still learn to classify the digits, even though the features of the data are scrambled. In the end, an MLP seems to perform better on the permuted version of the

dataset than the CNN, but the CNN outperforms the MLP when the data is unscrambled, as is expected due to adding more information by identifying features in the images.

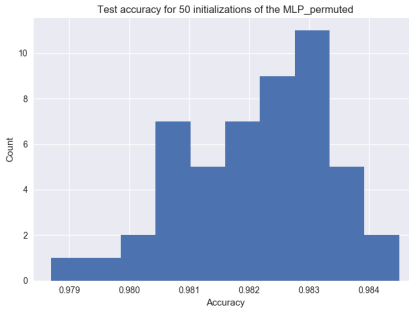


Figure 2: Accuracy of the MLP on the permuted data

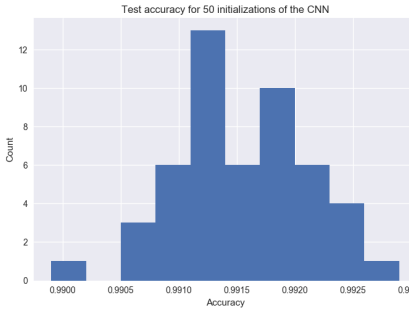


Figure 3: Accuracy of the CNN on the unpermuted data.

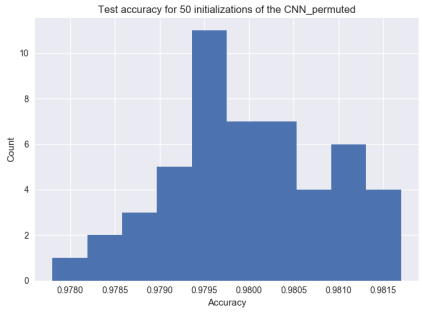


Figure 4: Accuracy of the CNN on the permuted data.

3 Task 2

Task 2 is centered around recurrent neural networks. Recurrent neural networks, at their basis, are networks that have loops (see Fig 5). These loops allow for some sort of 'memory' in these networks, allowing recurrent neural networks to excel at sequence-to-sequence learning. However, when sequences become too large, it is not possible anymore for this architecture to connect the relevant information that it got as input some time ago to the information that it gets as input now. This is why a new kind of recurrent neural network was invented, called Long Short Term Memory (LSTM) networks. LSTMs have essentially the same architecture as shown in Figure 5 except that the inner structure now contains four interacting layers. The figures in this section are based on the now famous blog by Christopher Olah[4].

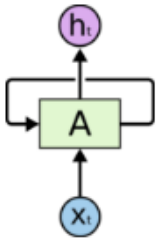


Figure 5: Simple visualization of a recurrent neural network with input x, output h and the internal state A being redirected into the network.

The first layer is the 'forget gate', which gets the input data and the output of the previous module or step. This layer learns whether to remember or forget certain information from the previous cell state. The second and third layer are the input gate and a *tanh* layer that combine into the new cell state. These two layers learn together which values should update the previous state. A visualization of this process can be viewed in Figure 6.

Finally, the output is based on a final filter of the current state. This filter is a combination of the *sigmoid* function applied to the previous output and current input and a *tanh* function applied to the current state. This decides which information is relevant to output finally as h_t. This process is illustrated in Figure 7.

In the following sections we will see different examples using *recurrent neural networks*. Two LSTMs will be used to learn two different, relatively complicated tasks, namely text generation and text translation. Additionally, we will teach a simple RNN to do additions and divisions.

[4]http://colah.github.io/posts/2015-08-Understanding-LSTMs/

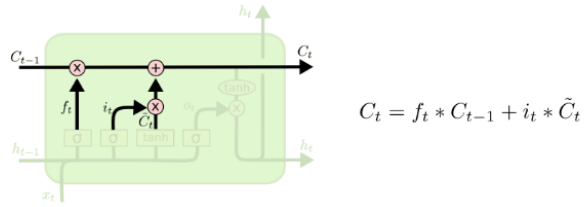


Figure 6: Input and forget gate that decide how to update the previous state. C_{t-1} indicates the previous state, f_t indicates the forget gate. The combination of the input gate i_t and the current state $\tilde{C}_t$ is then added to the previous state, after the forget gate has been applied.

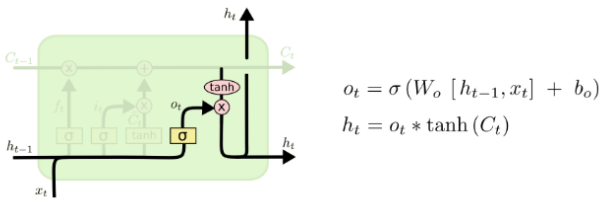


Figure 7: The final output of the current LSTM module is generated.

3.1 Text generation

The example script *lstm_text_generation.py*[5] uses text from Nietzsche's writings. The data consists of 9935 lines and contains in total 600,901 characters. The text is split into semi-redundant sequences of 40 characters, after which these are vectorized by using one-hot character encoding. This means each character is represented by a vector of length 57 (as there are 57 unique characters in the data), in which all of the entries in the vector are 0, except one entry corresponding to the current character, which is 1. An example of one-hot encoding is given later, in Section 3.3. Additionally, all upper case characters are converted into lower case characters to reduce the dimensionality of the data, since upper case characters would almost double the amount of unique characters.

Considering a sentence is then made up of 40 characters, and each character is one-hot encoded as being one out of 57 characters, one input example will be a matrix or tensor of shape (40,57) corresponding to a sentence of 40 characters. 200,285 examples are fed to the network, which has the simplest possible architecture. The network consists of a single LSTM cell with an output dimensionality of 128, after which a dense layer with 56 output nodes is computed, corresponding to the 56 available characters. Finally, a *soft-max* function is applied, and the model is compiled with *categorical cross-entropy* as the loss function and an *RMSprop* optimizer with a learning rate of 0.1. The model is fitted with a batch size of 128 and is trained for 60 epochs.

After the first epoch, the network already makes a surprising amount of real words, with the seed "there is no doubt that for the discover" and a diversity (a restriction on the different characters used) of 0.2 it generates the following:

> *"there is no doubt that for the discovered the consuched and and the more the more will the most and be also and the more as the most in the more to soul and who whole the more a profing the more to the most and the profound to the will the most and an an the present and the more who whole in the stronger to the world and and the more to the existion of the more the more and the more to the considered the more as the more as the more is"*

From which we can see the network has already learned to correctly spell most words that it uses, even though it does not use a lot of different words. If we ask for a higher diversity of words, the network produces almost no real words, as is seen from the following quote, generated with the same seed:

> *"there is no doubt that for the discoveresed peased–which whe progably atilatimalics and saser: religion and in to santed and for inpor for aga inst consrieb est lime dimpay dogition toountingconry an akem found in orities of tormago-bedonding tousout of that as the voaceden also one caless, he hald" so the constinutusinger of thut recalt menstered must expersence being, potere will the case correen in the forming peotrement, and or pod"*

[5] https://github.com/keras-team/keras/blob/master/examples/lstm_text_generation.py

But one should keep in mind, this is only after 1 epoch of training. After 60 epochs the output is as follows (with seed 'erhaps from below–"frog perspectives," '):

—— diversity: 0.2

"erhaps from below–"frog perspectives," in the soul of the substance of the substance of the superioral superior, and that the man and the problem of the substance of the sense of the standant of the substance of the man of the most substable the prosous conception of the substable the protection of the most strongest profound the substance of the substance of the substance of the substance of the most substable the sense of the standan

—— diversity: 1.0

erhaps from below–"frog perspectives," and rit things here beava", the pronous, morally map, as every one "betteref.=–it will such the delicatement; nemiless and strange madreman of the thoin extent far afly evident to impresences of origin" systemss of the most spirit, have been a philosophy please before comprosters without a living certain to shave how "fruct and can knowledge," there it just self-spitude hav

From these quotes we can infer that the network is starting to learn to spell more and more words correctly, even when imposed with a higher diversity. It also seems to show a mixture of prepositions, verbs and nouns that matches the general grammar rules that humans adhere to. Meaning that it does not follow up verbs with more verbs, and often has a subject of a sentence. However, the network is still not making a lot of sense, and a lot more epochs are probably required before the network stars making sense.

The fact that we only use lower-case letters might make it much harder for the algorithm to learn the data properly, since names are usually capitalized and now they cannot be easily recognized by the algorithm. New sentences, starting after a dot and starting with a capital letter are also not identified as easily anymore. This loss of information increases the difficulty for the LSTM and it reduces its correctness.

After training on Nietschze, a new dataset is introduced to the network. We use the plain-text (incomplete) version of the books from the famous Harry Potter saga. The text does not contain all books, because some books were found with bad formatting. We only kept parts of the books that were formatted correctly, as quickly inferred from visual inspection of the books. In total, the dataset contains 20,524 lines and 1,451,978 characters. After 10 epochs, with diversity of respectively 0.2 and 1.0 the network outputs the following, when given the seed 'no, but i hope it's really hurting him,"'

—— diversity: 0.2

no, but i hope it's really hurting him," said harry, smiling a stairs staring the stairs of the stairs to the door and started to the dementors and staring to his stone and the door and hermione along the first stairs to his starter to his face as he was still had been a stairs to the death stairs to the class was a stairs shoring the stairs starter and started to his room and hermione along the first stairs to the class was a stairs an

—— diversity: 1.0

no, but i hope it's really hurting him," both like ron clasted his class, take the hill before he wexwin backneas was a preitute. he'd has all the very sortaitling tinys again. onded uncle lot, trying at his outside of the great phool, find out.

Where we can see that in the first case it identifies the ending quotes as something after which the word 'said' usually appears. However, in the second case this has dissapeared again due to the larger diversity constraint. Again it becomes apparent that a higher diversity leads to more non-existing words at a low number of epochs.

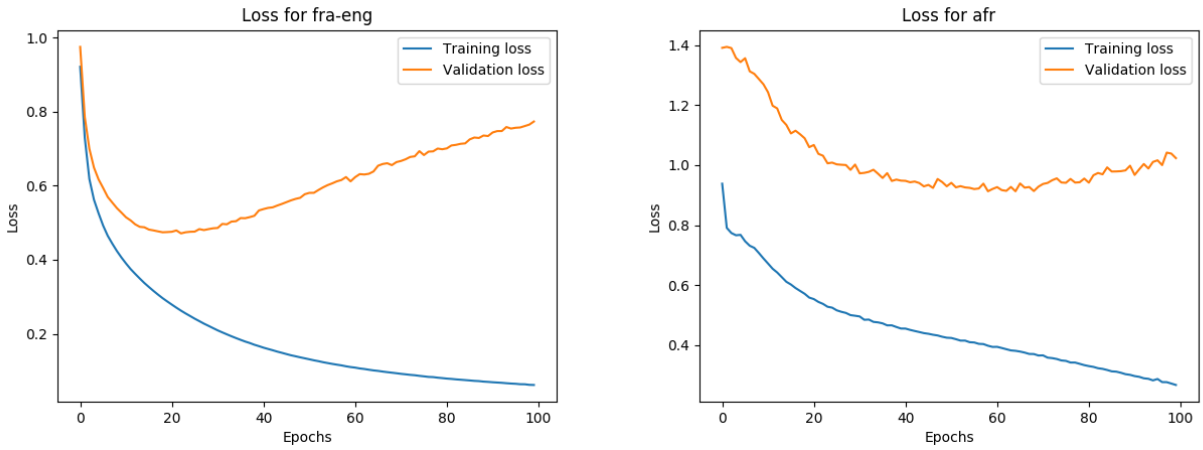
Interestingly, it seems that after epoch 21 the algorithm starts to learn the wrong way, since this is one of the last semi-coherent sentences that it speaks (with seed: 'itory deserted, dressed, and gone down t')

—— diversity: 0.2

itory deserted, dressed, and gone down to see him a slytheri ?" said harry, and he had a sight of the staircase and sharts and hear the staircase and hermione was something and they were a should seemed to be and seemed to the dark art of the should seemed to be a second should hear the stone should have the man he was a stone second to the hall that he was still hear the should be and sharts to the stone was something to have a should

—— diversity: 1.0

itory deserted, dressed, and gone down to watch the last pointion than themselvesing me?" "knowstur," said fred had needed smabled aunt madained and potter and gones goedon rup he'd believe them is a that for at a while he face in around curious as ever branding his itch in a broomstick others, as entrut to have i gaftdautuly,, he sat they looking for them with he's been her. "boney a wither mrs. weasley want up her do packed, but har



(a) Loss for English to French translation.

(b) Loss for English to Afrikaans translation.

Figure 8: Training and validation loss for two different language translations.

After these epochs, the network kept training until 100 epochs, but it was not outputting any words anymore. For some reason its output, even at low diversity was something resembling 'hh s hhe s hs h s s s a h h se sn h s sa oh he s h s s hs xas t n s s se n sh sh '. We think this might be due to incorrectly formatted text having slipped through the cracks. When after 20 epochs this bad formatted part of the text is reached, the network will strongly reduce the error/loss function by outputting letters separated by spaces. A training text fully checked by hand would eliminate this possibility, sadly this is not feasible due to the large amount of lines.

3.2 Text translation

Continuing our LSTM endeavours, we will have an attempt at text translation in this section. To accomplish this, we start with French to English translation pairs[6]. We have a dataset consisting of 10,000 samples of long and short sentences translated from English to French. 8,000 of these are used as a training set, and 2,000 as a validation set. The sentences are one-hot encoded on character-by-character basis again, and the architecture is as follows. An encoder *LSTM* cell takes the input sequence of characters and produces a state vector. Another *LSTM* cell, the decoder, is then trained to turn the sequences into the same sequence, but offset by one timestep in the future. The model then uses the encoder and decoder to predict the next character, and it keeps doing this until the 'END' character is reached, or the character limit is reached. The model is fit with an *rmsprop* optimizer and uses the *categorical crossentropy* loss function.

As is apparent from Fig 8, the validation loss seems to drop quickly in the first 30 epochs for both languages, but is not improved upon after this. While the training loss decreases steadily, the validation loss actually goes up again, indicating that language translation is prone to over-fitting. This can be explained by the fact that every new sentence in the test set, is a combination of words which, because of certain grammar rules, has to be ordered in a certain way that the algorithm has not seen before. A fix for this issue would be to include more training data. This would generalize the translated sentences and train the network with examples for every different combination of grammar rules in the translation language.

3.3 Mathematical operations

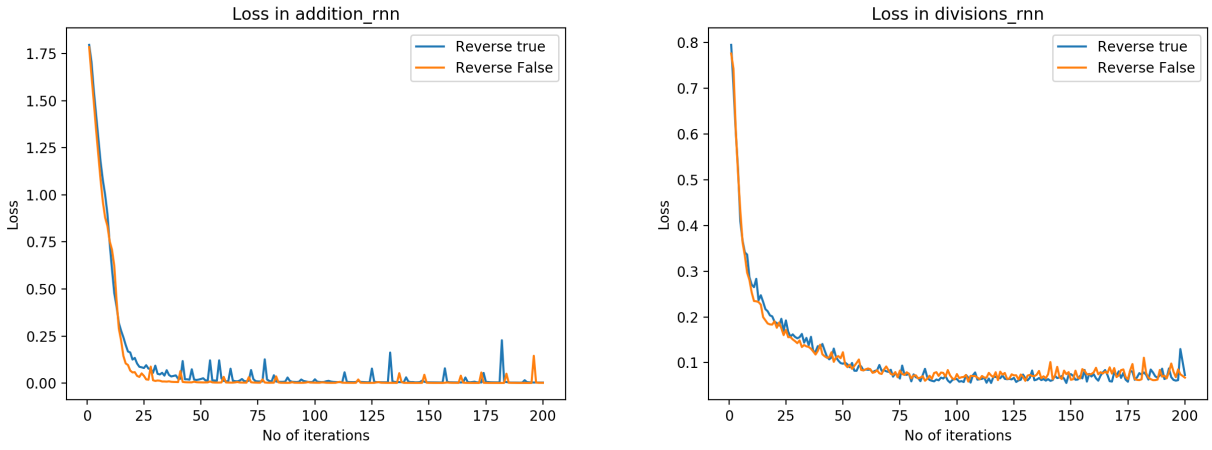
The *addition_rnn.py* is a sequence to sequence learning algorithm implemented to calculate mathematical operations. The algorithm is hard-coded to generate two three-digit integer numbers consisted of the digits 0-9 and gap that are encoded into one hot integer representation or a 10 character long vector each and then decode each one hot integer into their output characters and finally vectorizes the probabilities of each output character. An example of how digits are encoded is shown in Table 2. The example shows how the number "572" is encoded to one hot integer representation. As it can be seen, each number is represented with nine '0' and one '1' in the corresponding position depending on the number, so that each number has a unique representation consisted of a 10 character long vector.

Then the algorithm learns how to perform mathematical additions itself and predict the outcome given the two numbers that are going to be added. To increase its performance, the algorithm is capable of reversing the order of the digits in order to reduce dependencies which consequently affect its accuracy.

[6]http://www.manythings.org/anki/fra-eng.zip

Table 2: A set of characters encoded into one hot integer representation.

number/position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
5	0	0	0	0	0	**1**	0	0	0	0
7	0	0	0	0	0	0	0	**1**	0	0
2	0	0	**1**	0	0	0	0	0	0	0



(a) Loss in additions.

(b) Loss in divisions.

Figure 9: Validation loss for two different mathematical operations, addition and division each in two different scenarios: regular order of the numbers and reversed order of the numbers.

Another version of the addition Recurrent Neural Network has been implemented. In this version the algorithm learns to compute divisions instead of additions. The only modifications in the source code are the type of the operation and a condition that must be checked before the operation is performed. This condition constraints the numerator to always be greater than the denominator so the result is always an integer equal to or above 1. Looking at Figure 9, one can notice that the addition, although it shows initially a higher loss value, can be learned faster by our algorithm. Not surprising, the algorithm that performs the division of the two generated numbers shows lower loss in comparison to the addition algorithm. This is likely due to the fact that many outputs will be the integer 1. However the network requires more iterations in order to improve its performance and predict the correct results. At this point it is worth noting that reversing the order of the numbers does not seem to affect the performance of the algorithm significantly. In the addition algorithm, the regular order of the numbers seems to perform slightly better, while for the divisions algorithm the reversed order of the numbers shows slightly better results.

4 Task 3

In this task we will use auto-encoders to compress three different types of data. We will start with the MNIST dataset to see if it is possible to reduce the amount of information needed to classify digits, and then experiment with two other data-sets.

Auto-encoding is a type of data compression algorithm that is learned from the examples instead of one single algorithm created to be used on many different cases. This means these encoders will only work on the data they are trained on, and usually will not work in general. They are very handy, however, when there is one specific problem in which the data compression algorithm might not be obvious.

4.1 MNIST data

The auto-encoder we will build first has of the simplest architectures available. The encoder consists of a single layer that has 784 inputs and 32 outputs (nodes). The decoder is then the reconstruction of the encoder, hence it will have 32 inputs and 784 outputs (nodes). The auto-encoder is created as a model that maps an image to the reconstructed image, so it encodes first, and then decodes the encoded image again. In this way, the encoder and decoder are trained at the same time.

The chosen optimizer is '*adadelta*' (Zeiler, 2012) and the loss is the binary crossentropy loss[7]. After fitting

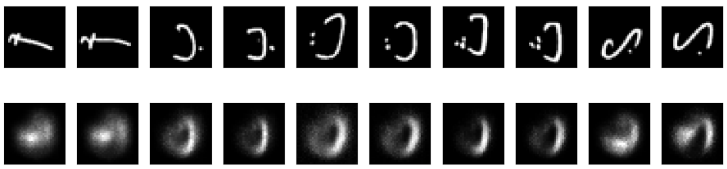
[7]https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/

the model for 50 epochs, with a batch size of 256, the result is visible in Figure 10. Visually, the decoded images can be recognized very easily, even after compressing the images with a compression factor of 24.5. This is a rather impressive result, and it would be interesting to see if an MLP or CNN would still be able to correctly classify these decoded images.

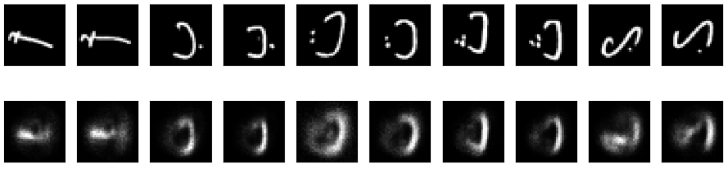
Figure 10: Top: Original test-set images. Bottom: Encoded and subsequently decoded images.

4.2 Arabic Characters

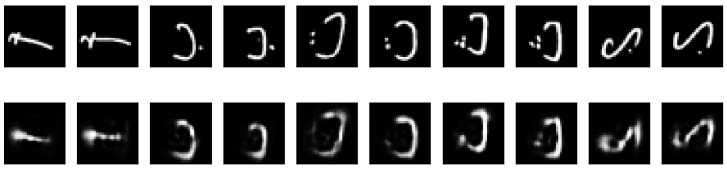
A quick transition from digits to characters can be made, to see whether the same performance can be reached when training on a bit more complicated problem. To make it even more interesting, we have decided to use a dataset of Arabic characters[8]. Using the same exact layout as in the previous section, except for a new compression factor of 32, we find the result in Figure 11. We can see that the network has much more trouble with encoding this data in the same amount of time, which makes sense because there are now 28 different classes instead of 10 and we have increased the compression factor. After the first 50 epochs the network learns to identify features that can correspond to the rough shape of the input, but is not very readable yet. Figure 11 shows very neatly how the shapes of the decoded digits become sharper and sharper when training for more and more epochs, until the character can be recognized quite easily after 500 epochs.



(a)



(b)



(c)

Figure 11: As Fig. 10, but after 50(a), 100(b) and 500(c) epochs of training on Arabic characters.

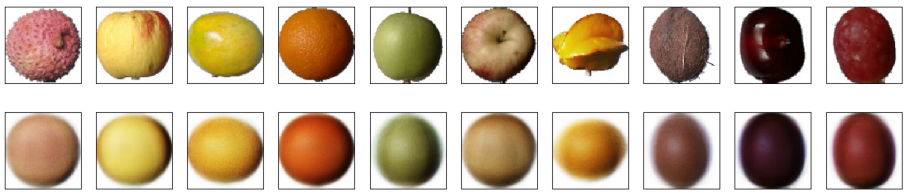
[8]https://www.kaggle.com/mloey1/ahcd1

4.3 Fruits

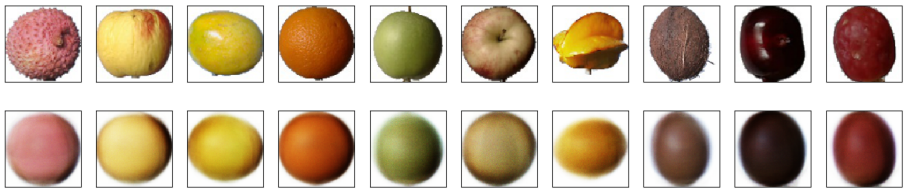
The final dataset we will use in this task, is a dataset containing fruits[9]. It is a rather impressive dataset containing 60 classes of fruits with in total a training set size of 28,736 images and a validation set size of 9,673 images. The images are RGB images of 100x100 pixels, meaning they can be loaded as a 3d array of shape (100,100,3).

The layout of the network is similar to the previous section, but we have increased the compression factor once again to reduce the amount of training time. The compression factor is now 300, meaning we encode the original 3d array into *only* 100 floats.

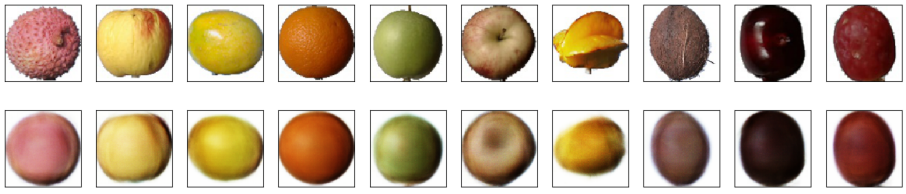
An interesting aspect of the current dataset in this task, is that our dataset has colors. This aspect seems to be learned surprisingly quick from our network. In Figure 12, after 50 epochs, it seems colors are easily identified from the original image. This is not surprising however, since most fruits do not have intricate color compositions, so it would be quite easy to compress the color information into less floats than originally used. This does mean that spots and details of the original fruit are quickly lost, as can be seen in Fig. 12(a), where for example the first two fruits originally have a black and red spot, but these features are lost in the encoding. For most fruits, losing some features is not a big problem, since they are mostly round, but for the 3rd to last fruit, the carambola or starfruit it is clear that this fruit is now unrecognizable. This loss of shape features seems to hold until even 500 epochs, where it is starting to show multiple different colors of the fruit of the right place. This loss of shape features is likely because the network is training on 60 different types of fruit of which most of them the shape does not vary like the shape of the carambola varies. So the carambola is sadly the odd one out, and will not be auto-encoded as successfully as the other fruits. This shows the low versatility of auto-encoders and the fact that they can only be used for very specific types of input data.



(a)



(b)



(c)

Figure 12: As Fig. 11, but after 50(a), 100(b) and 500(c) epochs of training on 60 different kinds of fruits.

[9]https://www.kaggle.com/moltean/fruits

5 Conclusion

In this report, we have looked at a plethora of different deep learning architectures applied to very different types of problems. First a simple deep multi-layer perceptron was applied to the MNIST data, which already showed a rather high accuracy of 98.22% with very small dependence on the initial configuration as inferred from the low standard deviation of 0.1281%. A convolutional neural network was also applied to this task, which outperformed the MLP with 1 percent point by identifying interesting features. However, on a permuted version of the dataset the MLP outperforms the CNN, as features are scrambled, yet apparently not scrambled enough to fool the CNN into a more than two percent point loss on the accuracy.

Secondly, three different applications of recurrent neural networks were explored. Text generation was found to be the hardest problem we had to deal with. The networks seemed to show a basic understanding of which words to use and even some grammar rules, but the text still did not make very much sense. In the case of the Nietschze data, this is likely due to not enough epochs of learning, and Nietschze himself does not make sense all the time as well. In the case of the Harry Potter data, the network eventually scrambled letters together, likely due to badly formatted input data. Text translation was found to be prone to over-fitting, with the training loss dropping steadily and rapidly, but the validation loss rising again after 20 epochs. A more generalized training set will be needed to reduce this problem. Finally, a simple recurrent neural network was learned two mathematical operations, namely add and divide integers. This problem was found to be the easiest, with the network not having much trouble to learn division, and even less trouble to learn addition.

Finally, auto-encoders were put to three different tests, auto-encoding the MNIST data, Arabic characters and fruits. They were found to be rather effective when trained on simple problems, only requiring more epochs as the amount of classes and compression factor increased. However, it is hard to quantify the effectiveness from these images alone. An interesting follow-up research would be training an auto-encoder together with a CNN to see how much compression can still reach the same classification accuracy as uncompressed images. This would be interesting from a data-storage point of view, as we could then learn how to compress the images without losing valuable information.

Overall, Keras has proved to be a powerful high-level API for researchers that are interested in machine learning. The high versatility and user-friendly implementation makes running many different kind of networks and problems achievable within a relatively short time-frame, while coding all the networks used in this paper from scratch would take months.

References

Abadi, M. et al. (2016). "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *ArXiv e-prints*. arXiv: 1603.04467 [cs.DC].

Kurbiel, T. and S. Khaleghian (2017). "Training of Deep Neural Networks based on Distance Measures using RMSProp". In: *ArXiv e-prints*. arXiv: 1708.01911 [cs.LG].

The Theano Development Team et al. (2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *ArXiv e-prints*. arXiv: 1605.02688 [cs.SC].

Yu, Dong et al. (2014). *An Introduction to Computational Networks and the Computational Network Toolkit*. Tech. rep. URL: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/.

Zeiler, Matthew D. (2012). "ADADELTA: An Adaptive Learning Rate Method". In: ISSN: 09252312. DOI: http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503. arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.