# The Memory Capacity of Neural Networks

Course: Seminar/Project Machine Learning & Neuroinformatics/Brain-Computer Interfacing (708.415)

Supervisor: Dr. Robert Legenstein

Author: Christoph Rieger

Student number: 01530103

Date: 13.02.2023

## Introduction

This project aims to investigate the memory capacity of neural networks depending on their size and architecture. Randomly generated input and output patterns are handed to the network to be memorized. Two different network architectures, as well as two different loss functions, are analyzed.

## Methods

### Framework

The used programming language for this project was Python 3.8.18 with the neural network framework PyTorch 2.1.0.

### Data

The input vectors x and target vectors y are binary vectors of size N. The number of these vectors is defined by the dataset size DS. Different values for DS were tried for each network, until the maximum number of memorizable patterns was determined.  All input and target vectors for each DS together yield the matrices X and Y.
The vectors x and y have a sparsity s which was chosen as 0.1. This means that 10% of the bits of x and y were active with values of one and the rest of the bits were inactive with values of zero. Both vectors were generated randomly, but no duplicate vectors with the same bits were allowed within the input and output matrix respectively.

### Network

The first network used was a 1-Layer fully connected network and will be called simple network from now on. It has N input neurons and N output neurons. The input and output layer are fully connected. The activation function used was the sigmoid function, which scales the outputs between zero and one. This behavior is desired because the target vector's values lie between those values. For an arbitrary value a it is defined as

$$\sigma(a) = \frac{1}{1+e^{-a}} \tag{1}$$

After the application of the activation function the network's prediction/output z is obtained. The prediction and the target vector are then handed over to the loss function. In this project the network is performing a binary classification task for each single bit of z. Thus, Binary cross-entropy (BCE) was chosen because it is designed for such classification tasks. For a batch size of one it is given by

$$l(z, y) = -(y \cdot \log(z) + (1-y) \cdot \log(1-z)). \tag{2}$$

As optimizer stochastic gradient descent and ADAM were tested. After evaluation ADAM was chosen as optimizer for this project because it was converging faster than stochastic gradient descent with no apparent drawback.

The described "simple" network was later expanded by a recurrence between the output neurons. The training process thus consisted of two steps through time, while using the same input vector. The first step was identical to the first network, the input goes to the N input neurons and is passed via the fully connected layer to the N output neurons.
After that, in the second step, the input was again given to the input neurons of the network. Additionally, the output neurons of step 1 were connected to the output neurons of step 2, creating a

recurrent layer with trainable weights. These recurring connections were initialized fully connected between the two output layers, but all weights of connections between the same output neurons (e.g.: output neuron 1 of step 1 to output neuron 1 of step 2) were always kept at zero. By that $N^2 – N$ trainable connections were added to the network.

At last, the recurrent network was expanded by adding a third step, which functioned in the same way as step 2, taking the original input vector as input and connecting the output neurons of step two to the output neurons of step 3. This resulted in three total steps with the same input vector.

## Loss and classification

In the evaluation phase each bit of the network's prediction z had to be cast to either one or zero. This was done by setting all outputs greater or equal than 0.5 to 1 and all smaller than 0.5 to 0.

Two different definitions of when a pattern counts as correctly memorized were analyzed. The first one only counted a pattern as memorized if the output vector is equal to the target vector.
For the second definition the number of bits to ignore #bti was calculated as

$$\#bti = N \cdot s \cdot 0.1\,. \tag{3}$$

During each training step $2 * \#bti$ output neurons with the worst loss were disabled. This was done #bti times for the group of output neurons that had a target of 1 and another #bti times for the group that had a target of 0. The neurons were "disabled" by setting their loss to 0. The idea behind this procedure was that stochastically the input and output patterns could overlap many of their active bits, making it more difficult for the network to 100% correctly classify those overlapping patterns. The bits to ignore are supposed to counteract this stochastic increased difficulty of memorization. Due to this disabling of output neurons, and thus preventing them from learning, the validation paradigm also had to be adjusted. Instead of requiring all the bits of the output and target vector to be equal, #bti of the active group and #bti of the inactive group were allowed to be unequal.

## Analyzed networks

To summarize, four different networks were trained and analyzed.

1. simple network
2. simple network with the custom loss function
3. recurrent network that took two steps through time with the same input data and with the custom loss function
4. recurrent network that took three steps through time with the same input data and with the custom loss function

## Determination of the maximum memorizable patterns

To determine the maximum number of memorizable patterns the appropriate DS for the network had to be found. It was not possible to simply take a DS much bigger than the maximum number of memorizable patterns, because for those the network was not trainable properly and it performed poorly.
Instead the biggest dataset a network can still memorize perfectly was searched. After finding it the DS was increased in steps of 100 until the biggest DS was found where the network could still memorize more than 90% of the dataset.

## Calculation of mean and standard deviation of the memorized patterns

The simple network without a recurrent layer was trained five times with each set of parameters and the mean and standard deviation of the memorized patterns was calculated over those five training runs. For each of these runs the data was randomly generated anew. For the more complex recurrent network only three runs were performed, to speed up the training process.

### Calculation of the number of trainable parameters

The number of trainable parameters $N_p$ was calculated to understand the complexity of the different networks and to later visualize the number of memorizable patterns per trainable parameters. For the simple network it is given by

$$N_p(simple) = N^2 + N. \tag{4}$$

$N^2$ is the result of the full connection between the input and output neurons and the N output neurons have one bias parameter each. For the recurrent network $N_p$ is given by

$$N_p(recurrent) = 2 \cdot N^2, \tag{5}$$

as an additional $N^2 - N$ connections between the output neurons of the time steps were added. These calculated numbers were verified by comparing them to the number of parameters the PyTorch model returned.

## Results

The learning rate used for training was 0.01 for the simple networks and the recurrent network with 2 steps. For the recurrent network with 3 steps the learning rate was reduced to 0.001 because the loss was not converging.

The results of the project for all four networks are given in Table 1 and 2. The accuracy was calculated as the maximum number of memorized patterns divided by the dataset size.

| | | N | | | | |
|---|---|---|---|---|---|---|
| | | 100 | 200 | 300 | 400 | 500 |
| Simple network | Maximum number of memorized patterns | 283 | 580 | 897 | 1194 | 1494 |
| | Accuracy [%] | 94.33 | 96.67 | 99.67 | 99.5 | 99.6 |
| | Mean of memorized patterns | 243 | 550 | 883.8 | 1165.6 | 1486.8 |
| | Standard deviation of memorized patterns | 23.95 | 32 | 11.95 | 23.56 | 7.93 |
| Simple network with custom loss function | Maximum number of memorized patterns | 337 | 694 | 1077 | 1580 | 2008 |
| | Accuracy [%] | 96.29 | 99.14 | 97.91 | 98.75 | 95.62 |
| | Mean of memorized patterns | 305 | 674.8 | 1046 | 1395.2 | 1904 |
| | Standard deviation of memorized patterns | 24 | 20.7 | 26.56 | 169 | 141.7 |

*Table 1: Results for the simple network, once with basic loss and once with custom loss*

|  |  | N |  |  |  |
|---|---|---|---|---|---|
|  |  | 50 | 100 | 150 | 200 |
| Recurrent network (2 steps) | Maximum number of memorized patterns | 534 | 920 | 1241 | 1549 |
|  | Accuracy [%] | 97.09 | 92 | 95.46 | 96.81 |
|  | Mean of memorized patterns | 524.8 | 902.2 | 1228 | 1539.33 |
|  | Standard deviation of memorized patterns | 6.18 | 12.5 | 9.27 | 7.13 |
| Recurrent network (3 steps) | Maximum number of memorized patterns | 589 | 1219 | 1528 | 1950 |
|  | Accuracy [%] | 98.17 | 93.77 | 95.5 | 92.86 |
|  | Mean of memorized patterns | 577.66 | 1199 | 1503 | 1929.33 |
|  | Standard deviation of memorized patterns | 13.3 | 14.51 | 17.72 | 14.7 |

*Table 2: Results for the recurrent network, once with 2 steps and once with 3 steps*

In Figure 1 the number of memorized patterns depending on N of each network can be seen.

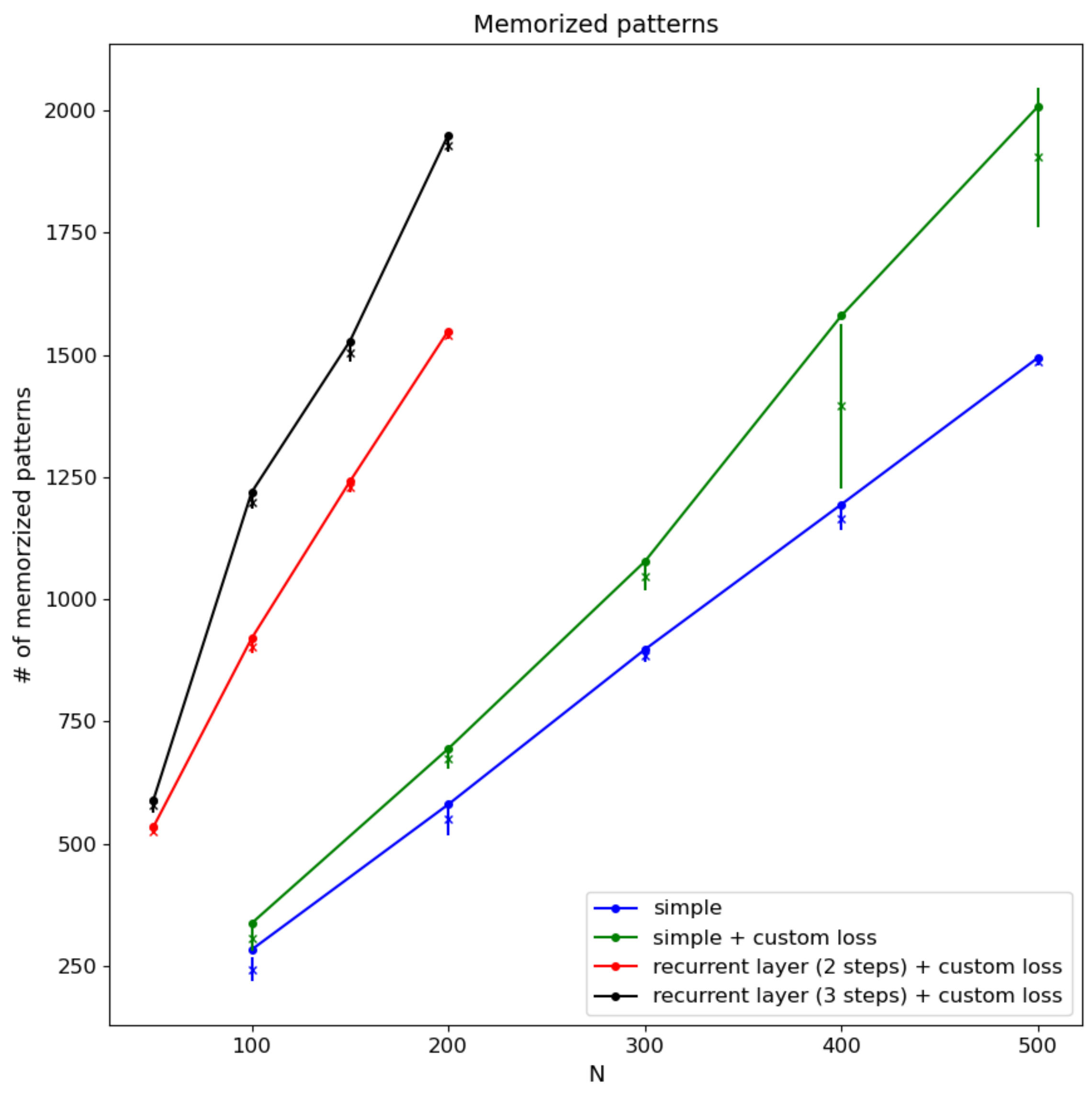

*Figure 1: Number of memorized patterns of the 4 networks. The datapoints of the graphs are the maximum number of memorized patterns of all training runs. Marked with "x" is the mean of the memorized patterns of the runs, with the standard deviation as a vertical line.*

The number of memorized patterns per N, depending on N, for each network can be seen in Figure 2. The graphs in this figure correspond to the slope of the graphs in Figure 1.

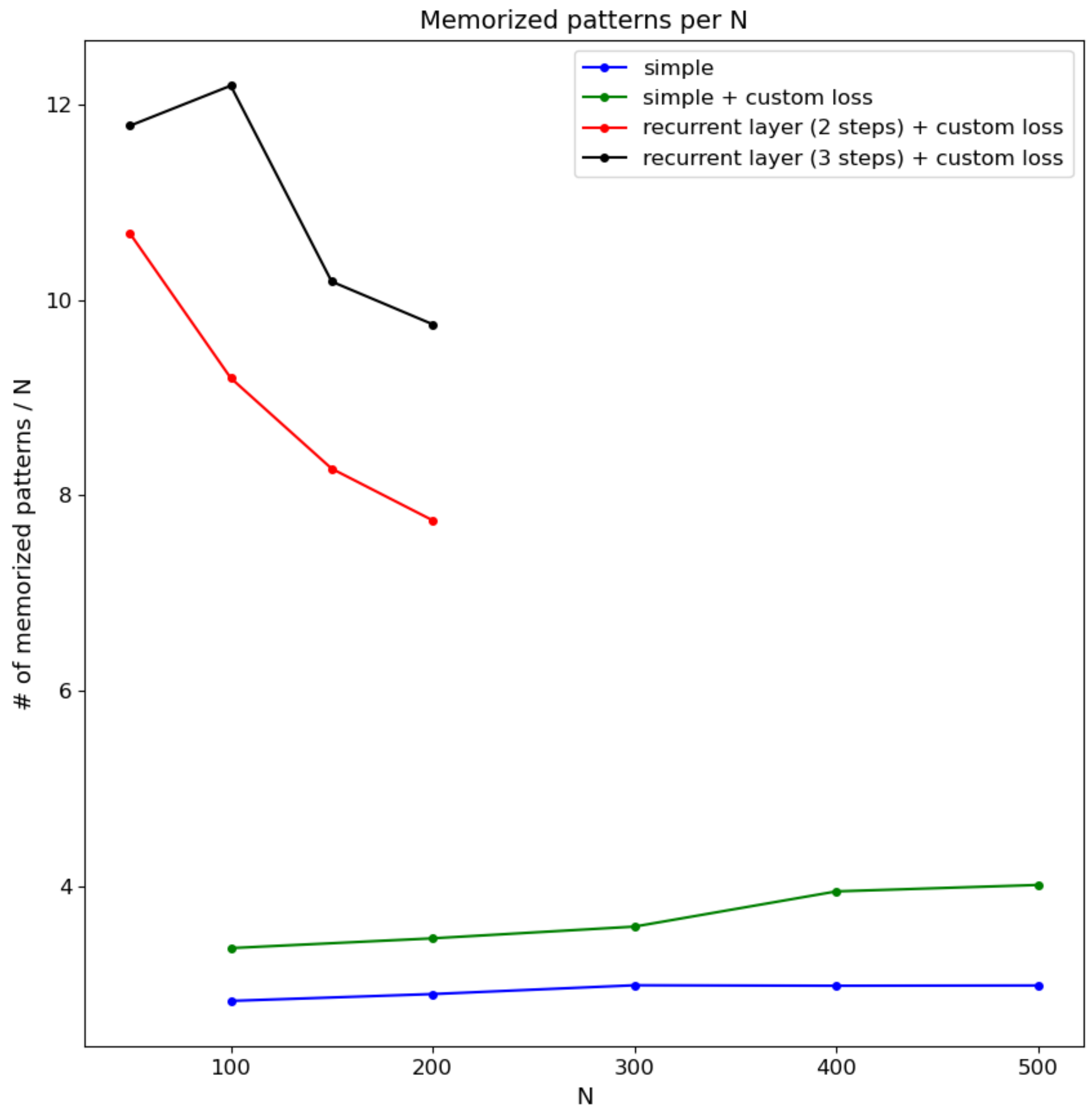

*Figure 2: Number of memorized patterns per N.*

The number of memorized patterns per trainable parameter, depending on N, for each network can be seen in Figure 3.

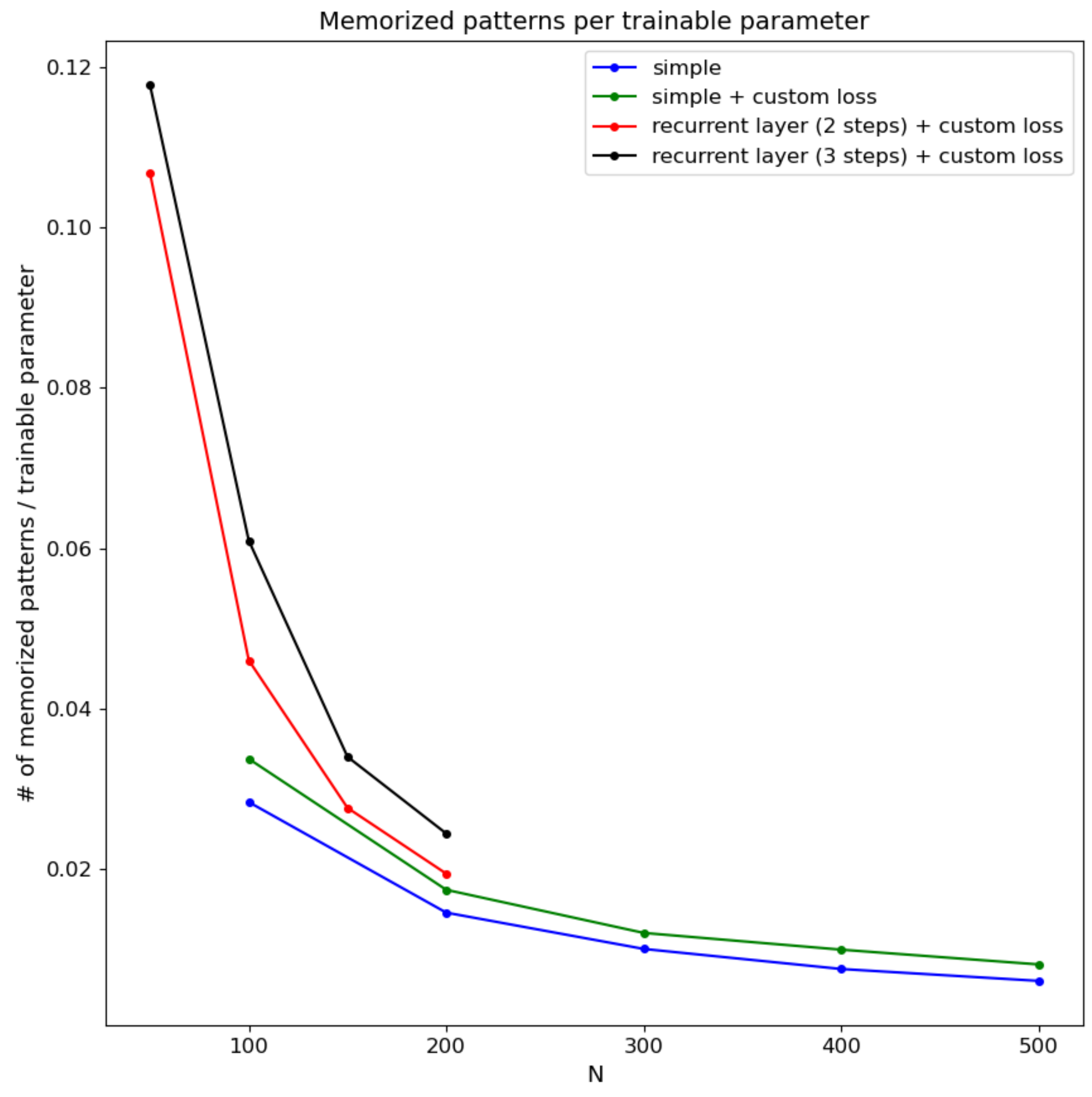

*Figure 3: Number of memorized patterns per trainable parameter.*

## Discussion

In Figure 1 the simple network was memorizing the least patterns of all networks. In Figure 2 its slope was constant, for each added neuron about three more patterns could be memorized. One would expect the number of memorized patterns per N to rise because with rising N the number of connections in the network increases by $N^2$. But it must not be forgotten that the complexity of the data also rises with rising N. Due to the sparsity of 0.1 of the data, for each increase of N by ten, one more active bit is added to the input and target vectors. Another factor to be considered is that the more the dataset size increases the higher the chance that some active bits are overlapping between different vectors. For example, it might happen that a vector pair $x_k$ and $y_k$ have an active bit at location $l_1$ and $l_2$. If another pair $x_l$ and $y_l$ also have an active bit at location $l_1$ and $l_2$ it becomes more difficult for the network to learn both vector pairs correctly. Because of that the custom loss function was devised and tried next.
Next the simple network was expanded with the custom loss function. Through this it memorized more patterns, the bits to ignore acting like a positive offset. As the complexity of the data was reduced by the custom loss the network's memory capacity rose faster with rising N. This can be seen in Figure 2 for N of 400 and 500, where the number of memorized patterns per N increased. As seen in Table 1 the standard deviation for N of 400 and 500 was high compared to the standard deviations of other N values and networks. It is unknown why it deviated, as the standard deviation was calculated over 5 training runs as for the simple network. But it could stochastically explain why the number of patterns per N was higher for N of 400 and 500, as the maximum values are further from the mean values, compared to the other datapoints. This would mean that that the network's memory capacity did not start to outpace the complexity of the data, but rather that it was a statistical outlier. One factor why these training runs yielded high standard deviations could be the dataset, which is generated randomly at the beginning of each run. Depending on chance active bits of data vectors are overlapping, making them more difficult for the network to memorize. One interesting metric to record would have been the percentage of overlapping bits in the input and target vectors. Then one could calculate the correlation between that and the final accuracy of a training run. Another reason for the runs to differ in accuracy is the random initialization of the weights. To determine this impact multiple training runs would have had to be performed with the same dataset.

When looking at Figure 1 and Table 2 the recurrent network with 2 steps and custom loss performed better than the simple networks. This was expected, as the additional recurrent layer introduced another $N^2 - N$ trainable parameters to the network, making it more complex. When looking at Figure 2 its number of patterns per N is higher than for the simple networks, however the value is falling with rising N. This behavior most likely occurred because the training process was more difficult. The loss of the simple networks strictly decreased with each training epoch, making the training process straight forward. In contrast the loss of the recurrent network increased and decreased many times during the training process. Furthermore, the network took longer to train than the simple network and thus, due to time constraints, could not be trained until the loss stopped decreasing. This could explain the decreasing memorized patterns per N of the recurrent network.

When performing 3 instead of 2 steps through time with the recurrent network the memory capacity increased even further, as seen in Figure 1 and Table 2. This was also expected, as the complexity of what the network is able to learn increased by adding an additional step through time. However the training of the network became more difficult once again, and the learning rate had to be reduced from 0.01 to 0.001 for the loss function to converge. This increased the training time further, causing the training quality to be lower. The network performed similarly to the recurrent network with 2 steps, just with a positive offset. In Figure 2 there was one outlier at N of 50 where the number of patterns per N was lower than for N of 100. To reduce the chance of it being a statistical outlier the training process for this N was repeated an additional time with even more epochs, yielding a similar result. This behavior most likely occurred because the custom loss was never intended to be used for N smaller than 100. For N of 100 the #bti equals 1. For all smaller N #bti is zero, as it is cast to an integer. Thus, for N of 50 the custom loss was having no effect, while for the other N it did. As the training time of the networks rises with rising N the N of 50 was used for the recurrent networks to generate an additional data point which was obtainable in a reasonable time.

The memorized patterns per trainable parameter can be seen in Figure 3. If the simple networks were able to memorize $N^2$ patterns, as was initially theorized, the graphs would have a constant slope of zero at a value of one. As the simple network has $N^2 + N$ parameters this would mean that each connection of the network would have to be able to memorize one pattern on its own. But this can never be the case, because with rising N the number of active bits in the data rises, thus more than one connection is needed to memorize a pattern. The results also reflect this, as even though the networks get more capable with rising N, all graphs have values below one and a negative slope. If the data vectors always had only one active bit, then the simple networks should be able to memorize $N^2$ patterns. In this project however the number of active bits rises with N, thus increasing the complexity of the data.
When comparing the networks that utilize the custom loss in this figure the recurrent networks were able memorize more patterns per parameter than the simple network. It seems that adding a time dependency increased the network's memory efficiency. The recurrent network with 3 steps also had a higher memory efficiency than the recurrent network with 2 steps, showing that increasing the time dependency is beneficial.