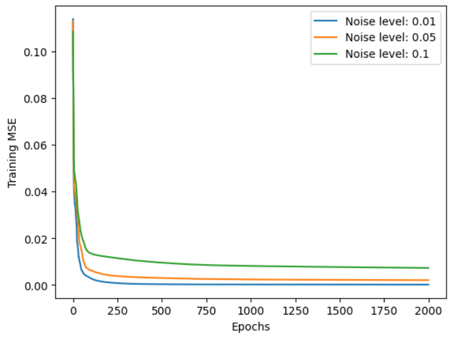
Bartosz Bogucki r1032166

Exercise 1

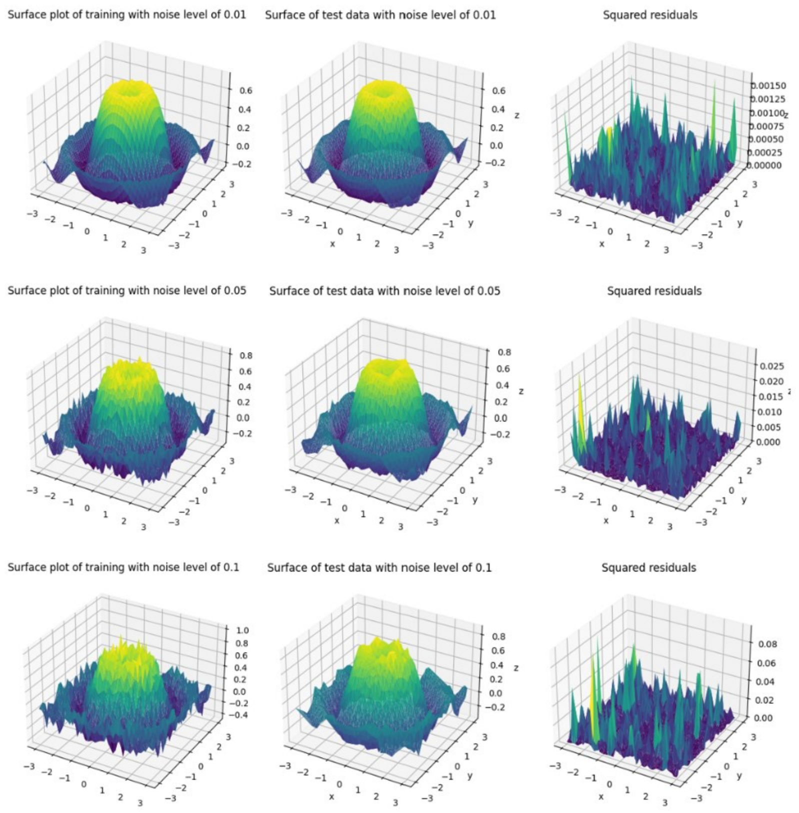
1.3.1.

As the noise level increases, the training data deviates further from the true underlying surface defined by the generating function. At higher noise levels, the model is trained on more distorted versions of the target function, making it harder to minimize the loss function effectively, as reflected in the training mean squared error (MSE) plot:



While all models eventually converge, they do so at different asymptotic MSE values. Higher noise levels result in higher final training MSE, indicating a less accurate approximation of the true function.

As can be seen in the plots below, the training data deviates more and more from the true function as the noise increases. The test surface plots, which evaluate the model on a target without noise, show that models trained on noisier data produce predictions that are further away from the true surface, resulting in higher error. The squared residuals on the test data become larger and more spread out as the noise increases, confirming that the model output is diverging from the true function. In addition, the predicted surfaces become bumpier and less smooth, resembling the noisy training data rather than the underlying function.

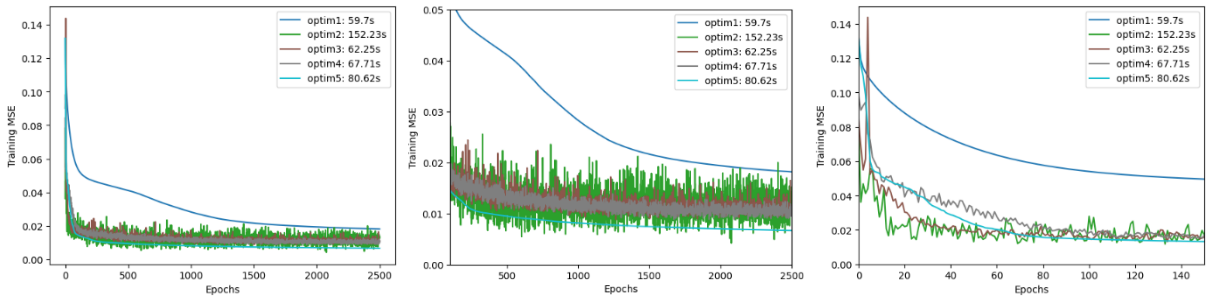


Increasing the level of noise in the training data makes the optimisation task more challenging and reduces the ability of the model to generalise. This is reflected in higher MSE values and visually larger residuals between the predicted and true surfaces.

1.3.2.

Vanilla gradient descent updates parameters using the entire dataset to compute gradients at each iteration. L-BFGS is a quasi-Newton optimizer that also uses full-batch gradients and is deterministic, but it includes second-order approximations and line search techniques to improve convergence. In contrast, stochastic gradient descent (SGD) computes gradients using random mini-batches, enabling faster but noisier updates. Accelerated versions

of SGD, such as SGD with momentum or Adam, improve convergence by incorporating inertia (momentum) or adaptive learning rates based on past gradients.
In the figure below, we compare several optimizers for training a neural network (with two hidden layers of 50 neurons each) on a dataset generated using the function defined in question 1.3, with a noise level of 0.1. The training MSE over 2500 epochs and the final test MSE are used for evaluation. The following optimizers were used here and in section 1.3.3:
- optim1: Stochastic Gradient Descent (SGD) with learning rate = 0.05
- optim2: SGD with learning rate = 0.1
- optim3: SGD with learning rate = 0.1, and Nesterov momentum = 0.9
- optim4: ADAM, a stochastic optimizer that accelerates convergence by adapting learning rates using first and second moments
- optim5: L-BFGS, with learning rate = 1, a single iteration, and strong Wolfe line search



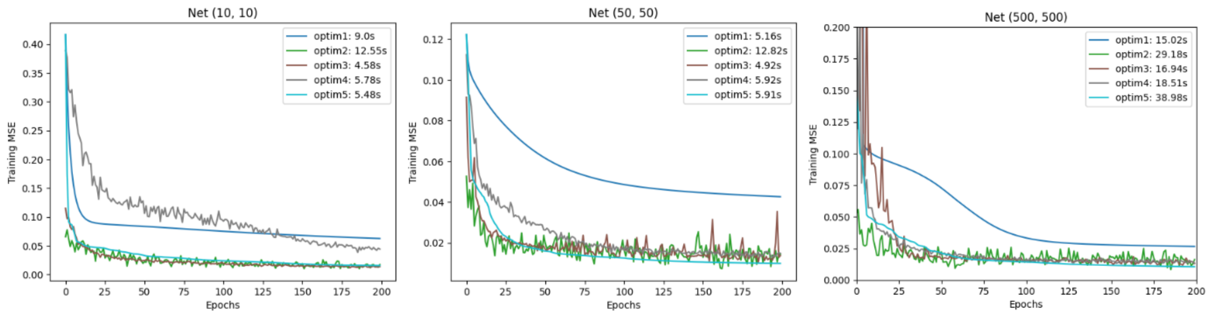
	optim1	optim2	optim3	optim4	optim5
Test MSE	7.314e-03	3.346e-03	1.806e-03	1.916e-03	4.345e-03

Basic SGD (optim1) with a learning rate of 0.05 converges slowly and results in the highest training and test MSE, although it has low variance and the shortest runtime. SGD with a higher learning rate (optim2) converges more quickly but exhibits high variance and the longest computation time, achieving a better test MSE of 0.0033. SGD with Nesterov momentum (optim3) improves convergence stability, reduces variance, and achieves the lowest test MSE of 0.0018 within a short training time. Adam (optim4) also converges quickly and smoothly due to its adaptive learning rate mechanism, yielding performance close to optim3 with a test MSE of 0.0019. L-BFGS (optim5) provides stable and fast convergence with very low variance and outperforms other algorithms by achieving the lowest training MSE. However, it does not outperform the best stochastic optimisers in terms of test accuracy, probably due to the noise introduced. Nevertheless, it achieves a very comparable test MSE of 0.0043 within a reasonable running time, but slightly longer than the accelerated stochastic algorithms.

Accelerated stochastic optimizers like SGD with momentum and ADAM outperform both basic SGD and vanilla-style methods in terms of convergence speed and generalization, especially on noisy data. While vanilla gradient descent is simple and stable in theory, it is often impractical for deep learning applications compared to these more advanced techniques.

1.3.3.

In the figure below, we compare several optimizers for training neural networks of the same architecture but with varying numbers of neurons in the two hidden layers - specifically, 10, 50, and 500 neurons each. The models are trained on a dataset generated using the function from question 1.3, with a noise level of 0.1. To evaluate performance, we use training MSE over 200 epochs and final test MSE.



	Net	optim1	optim2	optim3	optim4	optim5
	(10, 10)	5.070e-02	6.772e-03	3.759e-03	3.430e-02	6.337e-03
Test MSE	**(50, 50)**	3.248e-02	4.477e-03	4.187e-03	4.848e-03	2.042e-03
	(500, 500)	1.516e-02	5.008e-03	3.501e-03	5.045e-03	2.185e-03

The size of the neural network has a significant impact on the optimizer's effectiveness, as it influences both training time and generalization performance. As network size increases, the model's representational power grows, which often leads to lower training MSE. However, larger networks also risk overfitting and may not always yield better performance on unseen test data. Interestingly, while larger networks involve more computations per epoch, optimizers tend to converge in fewer epochs, so total training time does not necessarily increase proportionally.

Simple SGD with no momentum (optim1) converge relatively quickly in terms of time, especially in smaller networks. However, they often fail to achieve low error, as shown by their consistently the highest MSE on test data across all network sizes, indicating the poorest performance. Nevertheless for neural netork of 500 neurans in bboth hidden layers training MSE and final test MSE is more comparable. More advanced optimizers, such as SGD with momentum (optim3), strike a good balance: they scale well with network size and tend to offer strong generalization without excessive training time. On the other hand, L-BFGS (optim5) consistently delivers one of the best test MSE, but at a steep computational cost - its training time increases dramatically as model complexity grows. This suggests that while L-BFGS can be highly effective for complex models, it may be impractical for time-sensitive applications. Therefore, choosing an optimizer should take into account not just model size, but also the trade-off between training time and generalization performance. For most scalable training scenarios, SGD with momentum or ADAM appears to offer the best compromise between speed and performance.

1.3.4.

An epoch represents one complete pass of the algorithm over the entire training dataset and reflects how many times the model has seen all the training examples. It indicates the number of learning updates the model undergoes across the data. In contrast, time refers to the actual computational duration required to complete these epochs, meaning the total time taken by the training process. Convergence occurs when the model's performance, such as error or loss, stabilizes and no longer improves significantly with further training iterations. When we say that an algorithm converges quickly, we usually mean that it reaches this stable performance efficiently in fewer epochs - and often, though not always, in less computational time as well.

1.3.5.

The total number of model parameters, i.e., the parameters that will be trained, in the convolutional neural network is 34 826 parameters:

Layer	Input Shape	Output Shape	Parameters Calculation	# Params
Input	(28, 28, 1)	(28, 28, 1)	-	0
Conv2D (32 filters)	(28, 28, 1)	(26, 26, 32)	(3*3*1+1)*32=320	**320**
MaxPooling2D	(26, 26, 32)	(13, 13, 32)	No parameters	0
Conv2D (64 filters)	(13, 13, 32)	(11, 11, 64)	(3*3*32+1)*64=18 496	**18 496**
MaxPooling2D	(11, 11, 64)	(5, 5, 64)	No parameters	0
Flatten	(5, 5, 64)	(1600,)	No parameters	0
Dropout	(1600,)	(1600,)	No parameters	0
Dense (10 classes)	(1600,)	(10,)	(1600+1)*10=16 010	**16 010**
Total trainable parameters				**34 826**

Note that there may also be internal state variables maintained by the optimization algorithm (e.g., in Adam), but these are not considered part of the network's parameters..

1.3.6.

Here we are using the dataset MNIST of a convolutional neural network architecture as in 1.3.5 over 15 epochs of different optimizers:

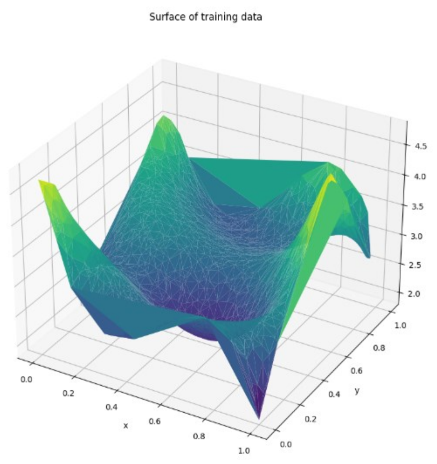
Optimizer	Test accuracy	Test loss
Adam with default initial learning rate of 0.001	99.15%	0.0253
SGD with default learning rate of 0.01	97.46%	0.0837
SGD with learning rate of 0.001	89.68%	0.408
SGD with learning rate of 0.1	98.8%	0.0357
Adadelta with default initial learning rate of 0.001	69.67%	1.9582
Adadelta with default initial learning rate of 1	99.03%	0.0303

Replacing the ADAM optimizer with Stochastic Gradient Descent (SGD) shows that excellent performance is still achievable, but only with appropriate tuning of the learning rate. With the default learning of 0.01 we achieve slightly lower test accuracy of 97.46% and slightly higher test loss of categorical crossentropy of 0.0837 comparing to 99.15% test accuracy and 0.0253 test loss of ADAM using the same number of 15 epochs. Potentially if we use more epochs it could have perform better, still slightly underfitting. If we use learning rate of 0.001 it is significantly worse as in this number of epochs it the model underfits. Using learning rate of 0.1 SGD shows excellent performance on test data of 98.8% test accuracy and 0.0357 test loss being very comparable to the results of ADAM. This illustrates that while SGD can perform well, it is much more sensitive to the choice of learning rate and often requires careful manual tuning to match the performance of adaptive optimizers like Adam. Furthermore, Adadelta is an another adaptive optimizer, like ADAM. Adadelta rescales each parameter update by the ratio of tracking recent squared gradients and the tracking recent squared updates. This mechanism allows it to automatically determine an effective step size, making it a truly self-adjusting method. The moving average over gradients acts similarly to momentum, yet unlike SGD with momentum or Adam, Adadelta does not require manual tuning of hyperparameters. However, its performance still depends on using an appropriate initial learning rate. When tested with the default learning rate of Keras of 0.001, Adadelta performed poorly, achieving only 69.7% accuracy and a high test loss of 1.958. In contrast, when used with its recommended learning rate of 1.0, it achieved

99.03% test accuracy and a test loss of 0.0303, being very comparable to the results of ADAM. Therefore, similar to Adam, both SGD and Adadelta can also achieve excellent results - provided their hyperparameters, especially the learning rate, are appropriately tuned.

1.4.1.

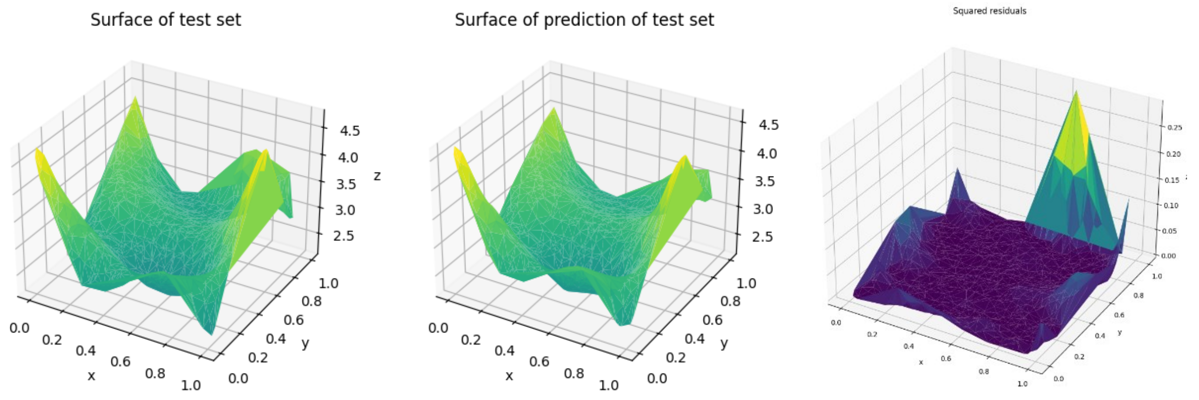
The purpose of using separate training and testing datasets is to evaluate the model's generalization ability - how well it performs on unseen, new data - rather than just memorizing the training examples. The surface of the training set is as follows:



1.4.2.

A feedforward neural network architecture was determined through empirical model selection, investigating different network depths, activation functions, and learning rates. The final model consists of six hidden layers with a combination of ReLU and tanh activations to enable rich nonlinear transformations, concluding with a single linear output neuron suited for continuous regression. The first three layers each contain 100 neurons, with alternating activations: the first and third use the ReLU (Rectified Linear Unit) function, while the second uses tanh. The next three hidden layers reduce the dimensionality to 50 neurons each, again alternating activation functions: the fourth and sixth use ReLU, and the fifth uses tanh. This design leverages both the unbounded, sparse activation of ReLU and the smoother, bounded characteristics of tanh. The model was compiled using the Adam optimizer with an initial learning rate of 0.01, and trained to minimize the mean squared error (MSE) loss. During training, a validation set was created by randomly splitting 20% of the 2,000 training samples (i.e., 400 points), which were set aside and not used during training for weight updates. This validation set was used to monitor the model's generalization performance throughout training.

1.4.3.



A comparison of the predicted surface and the ground-truth surface of test data confirms the model's strong approximation capability. The squared residual plot further highlights that most errors are very small and spatially localized, indicating accurate approximation across most of the domain. The final test MSE is 0.00435, demonstrating excellent predictive accuracy and good generalization as achieved on unseen data during training.

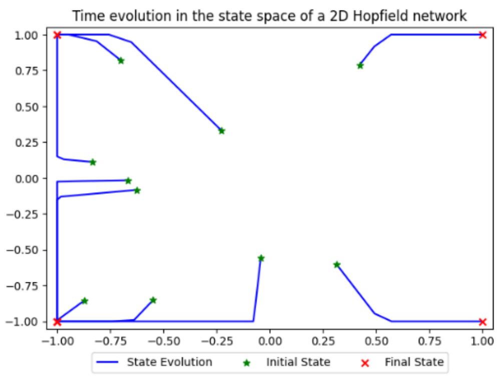
1.4.4.

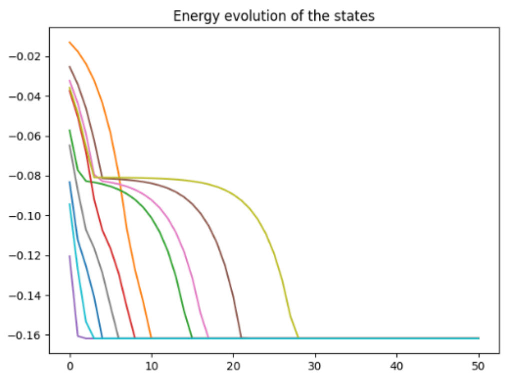
To avoid overfitting, early stopping is used - a regularization technique that halts training when the validation loss stops improving. Although training was set for 200 epochs, early stopping terminated the process at epoch 52, once the validation loss had not improved for 10 consecutive epochs, thereby preventing overfitting. However, other regularization strategies such as L2 regularization, L1 regularization, and dropout could also be considered. L2 discourages large weights, promoting smoother functions, L1 encourages sparsity by penalizing the absolute magnitude of weights and dropout randomly disables neurons during training to prevent over-reliance on specific pathways.

Exercise 2

2.1.1.

A Hopfield network with the target patterns [1, 1], [-1, -1] and [1, -1] and the corresponding number of neurons is created. Ten random point pairs in the range -1 to 1 are created as initial states.





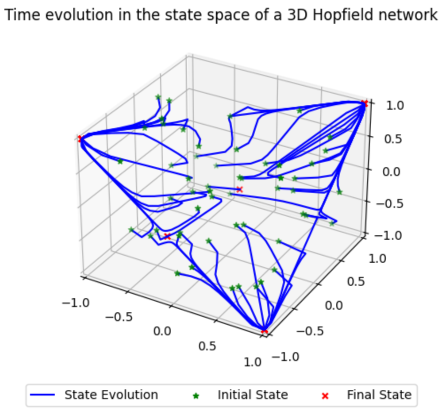
Not all of the resulting attractors matched the original target patterns - 7 out of 10 test inputs converged to correct target patterns. Notably, inputs such as [−0.23, 0.33], [−0.83, 0.11], and [−0.70, 0.82] converged to the spurious state of [−1, 1], which was not among the stored patterns. This unwanted attractor arises due to overloading the network's capacity. We can still compute a 2*2 weight matrix to store the three target patterns, although with only two neurons the network's capacity is smaller than the number of patterns, so perfect recall of all three is not guaranteed. With only 2 neurons, the empirical storage capacity is approximately 0.138*2=0.276 (from the empirical formula of $c \approx 0.138 * n$). The theoretical bound, given by $c \approx \frac{2}{2*log_2 n}$, equals 1 for n=2. Attempting to store 3 patterns exceeds both estimates. When multiple patterns are stored using Hebbian learning, the network may unintentionally store spurious states due to interactions between patterns. For example, the negation of a stored pattern, e.g. the spurious state of our case of [−1, 1] as the opposite of target patterns of [1, −1], or mixtures of patterns can become local energy minima, causing the network to converge to incorrect but stable states.

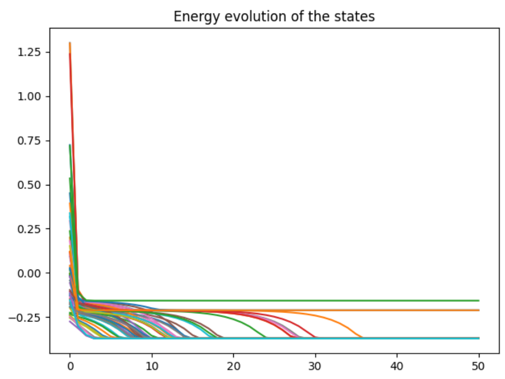
On average, the network took 12.5 iterations to reach an attractor.

The attractors are stable, as over 100 repeated runs with the same inputs, the network consistently converged to the same final state, confirming deterministic convergence.

2.1.2.

A Hopfield network with the target patterns [1, 1, 1], [-1, -1, 1] and [1, -1, -1] and the corresponding number of neurons is created. Fifty random 3D input vectors were generated within the range [−1, 1], along with three additional symmetric test cases: [−0.1, −0.1, −0.1], [−0.5, −0.5, −0.5], and [−0.5, 0.5, −0.5].:





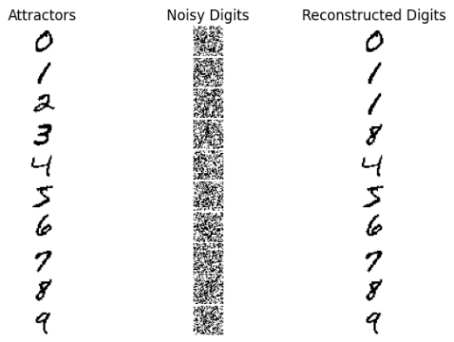
All 50 random initial states converged to one of the correct stored patterns, showing improved performance compared to the 2-neuron case for the same number of target patterns. However, the three symmetric test inputs did not converge to any of the target patterns. In particular, initial states of [-0.1, -0.1, -0.1] and [-0.5, -0.5, -0.5] converged to the final state of [-0.06, -1, -0.06], the initial state of [-0.5, 0.5, -0.5] converged to [0.37, -0.37, 0.37]. These results indicate a convergence to unwanted attractors of spurious states, again due to capacity limits. While 3 neurons allow a higher capacity, 0.138*3=0.414 patterns can be stored reliably, which is still less than 3 patterns. However, the performance is noticeably better as more inputs converge correctly due to the increased dimensionality.

On average, the network took 26 iterations to reach convergence. This higher iteration count is expected, as more neurons result in a more complex energy landscape.

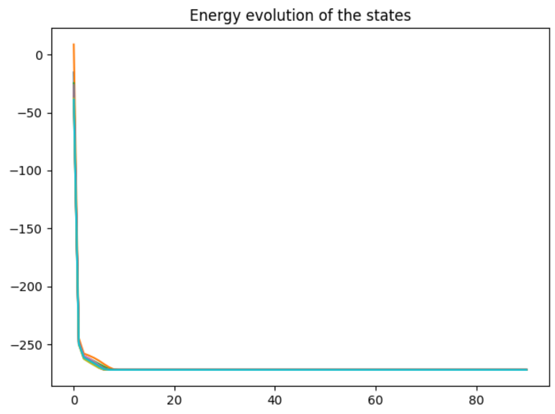
The attractors are stable again, as over 100 repeated runs with the same inputs, each input consistently converged to the same final state over multiple runs.

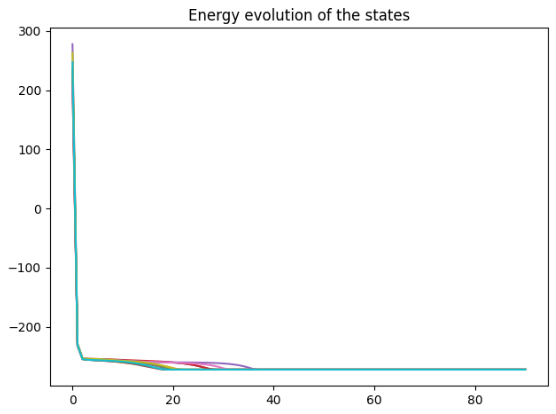
2.1.3.

The Hopfield model is not always able to successfully reconstruct the original digits, especially when the noise level is high. At lower noise levels, the network generally converges to the correct digit attractor, as the noisy input remains within the basin of attraction of the original pattern. However, as the noise increases, the input may fall outside the correct basin and instead converge to a wrong attractor - often another stored digit rather than the intended one. These are not spurious in the traditional sense (i.e., not unstable or untrained patterns), but rather valid stored patterns that do not match the noisy input's intended digit. Although the storage capacity of a Hopfield network is about 0.138*N, e.g., about 8 patterns for 64 neurons, even storing fewer than this number of patterns does not guarantee perfect recall. This is especially true at high noise levels, where inputs can be significantly distorted and may end up closer - energetically or in terms of similarity - to the basin of another digit. Digits such as 3 and 8 or 1 and 7, which share similar visual structures, are particularly prone to this kind of misclassification. The Hebbian learning may unintentionally reinforce overlapping features, increasing the chance of convergence to these incorrect attractors. For example, using a noise level of 5 with up to 90 iterations, noisy versions of digits 2 and 3 were observed to converge to attractors representing digits 1 and 8, respectively:



Furthermore, higher noise levels generally result in longer convergence times. When the input is more corrupted, the network takes more iterations to reach a stable state because it must navigate through a more ambiguous energy landscape. This is illustrated in the energy plots - the left plot below shows the energy evolution of the states with a noise level of 1.5 and converges after less than 10 iterations, while the right plot has a noise level of 8 and converges after a much higher number of iterations, around 35:





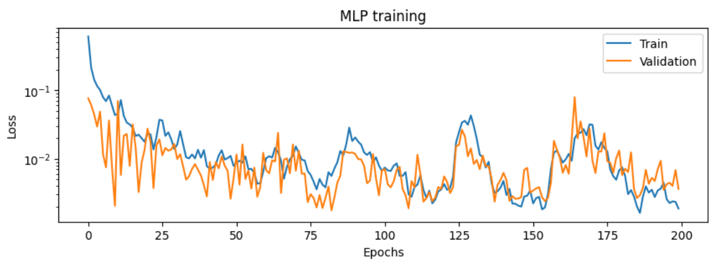
2.2.1.

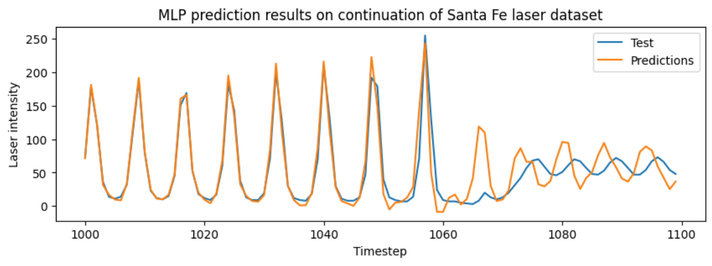
A Multilayer Perceptron (MLP) with one hidden layer was evaluated on the Santa Fe laser dataset across various combinations of lag values, hidden layer sizes (number of neurons), validation sizes, and numbers of cross-validation folds. Model performance was evaluated using Mean Squared Error (MSE) on the test set. Hyperparameters were tuned via grid search to identify the best-performing configuration. It is important to note that results varied depending on the random seed used for training, which significantly affected convergence behavior and final accuracy.

Lag	Neurons	Val Size	Folds	MSE
2	20	100	4	3139.49
20	100	100	4	4223.14
50	100	100	4	4400.48
50	100	150	5	2681.01
50	**100**	**50**	**5**	**620.25**
100	20	100	4	3383.46
100	100	100	4	3017.782
100	100	150	5	854.34

Larger lag values provide a broader view of the time series, allowing the model to learn more complex temporal patterns. In contrast, small lag values, such as 2 or 20, result in a window that is too narrow to capture meaningful dependencies, leading to poor generalization and high MSE on the test set. Increasing the number of neurons generally improves training performance but does not always lead to better test performance. When the lag is too small, a high number of neurons can result in overfitting due to a lack of sufficient temporal context. For MLP models, all temporal memory is encoded within the input window defined by the lag. The hidden layer must be large enough to process this input effectively. However, overly large lag values may introduce irrelevant or noisy historical information, potentially degrading performance. As such, moderate lag sizes, such as 50, tend to yield the best results. Increasing the number of validation folds generally improves generalization, while validation size influences stability. However, larger validation sizes can reduce performance in MLPs, which tend to memorize training data.

The best empirical performance was achieved with a lag of 50, 100 neurons, a validation size of 50, and 5-fold cross-validation, resulting in a test MSE of 620.25. As shown in the plots below, training and validation losses for this model were comparable, indicating no significant underfitting or overfitting, but the model showed high variance across runs, making it not stable. Despite this, predictions closely matched the test data.





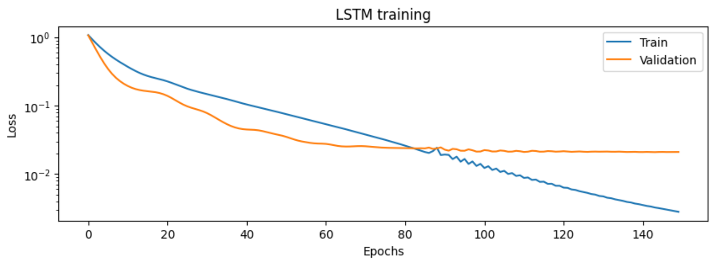
2.2.2.

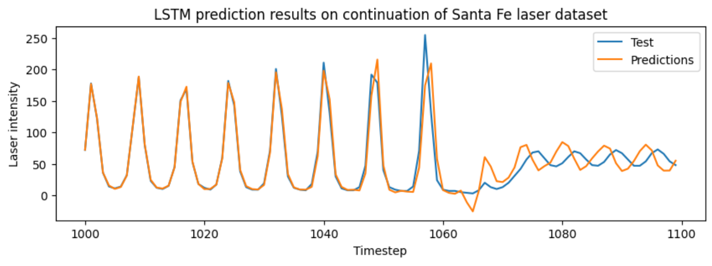
Hyperparameters for the Long Short-Term Memory (LSTM) model were tuned using a similar grid search strategy, exploring the same range of lag values, neuron counts, validation sizes, and cross-validation folds, with some additional configurations. The MSE on the test set was again used as the primary evaluation metric.

Lag	Neurons	Val Size	Folds	MSE
2	20	100	4	3808.38
20	100	100	4	1867.53
20	60	100	4	638.79
25	100	150	5	620.90
50	100	100	4	365.94
50	**100**	**50**	**5**	**343.398**
100	20	100	4	354.15
100	100	100	4	474.50
100	100	150	5	452.33
100	20	150	5	499.33
100	100	150	5	344.46

The LSTM architecture includes input, output, and forget gates, which allow it to maintain and update an internal memory state. This enables the model to learn temporal dependencies effectively without requiring large input windows. However, extremely short lag values of about 2 still provide insufficient context, resulting in high error. When the number of neurons is adequate, even relatively short lags, e.g. 20, can yield good performance. In contrast, extending the lag to 100 sometimes increased error slightly, likely due to the inclusion of irrelevant or noisy inputs from the distant past. The best results were achieved with a moderate lag size of around 50 and rather a high number of neurons, e.g. 100, particularly when paired with more folds for better generalization. A high number of neurons allowed the model to fit the training data effectively while maintaining generalization, thanks to LSTM's gating mechanisms that mitigate overfitting by filtering redundant signals.

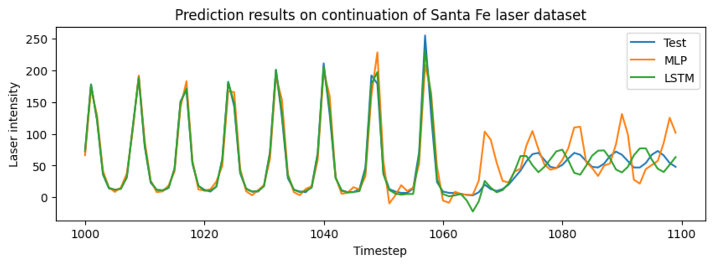
The best LSTM model in terms of performance on the test set used the same configuration that worked well for MLP - a lag of 50, 100 neurons, a validation size of 50, and 5-fold cross-validation. This model achieved a test MSE of 343. Compared to MLP, the training process for LSTM showed a much lower variance across epochs, demonstrating greater stability. For this best model, the validation loss closely followed the training loss, with only slight overfitting after about 90 epochs and no significant degradation thereafter. Predictions from this model closely matched the test data, confirming both the accuracy and good generalization of LSTM in this time series prediction task.





2.2.3.

Across all tested configurations, the LSTM consistently outperformed the MLP in terms of both accuracy and training stability. LSTM achieved significantly lower test MSE and showed better generalization with less sensitivity to training seed or validation layout. Its architecture allows it to learn long-term dependencies with less reliance on long input windows, unlike MLP which require larger lag values to compensate for their lack of memory. Thus, MLP require longer input windows to perform well, which may be impractical for certain applications. While MLP is a computationally faster model due to its simpler structure, this speed advantage is contrasted by its instability.



The plot above gives a comparison of predition of best model of LSTM and MLP. The best LSTM model achieved a test MSE of 343, compared to the best MLP at 620. In addition, LSTM was more robust across runs and folds. Therefore, LSTM is the preferred model for this time series task as it provides better accuracy and reliability. Nonetheless, MLP may still be considered a lightweight and interpretable alternative when computational efficiency is critical, provided that validation performance is carefully monitored.

Exercise 3

3.1.1

Several architectures were tested on MNIST using an stacked autoencoder under consistent hyperparameters: 20 epochs for layer-wise pre-training, 10 epochs for classifier training, and 30 epochs for fine-tuning, with a batch size of 128. The table below summarizes the results:

Input	Architecture	Accuracy before fine-tuning	Accuracy after fine-tuning
784→	256→64	0.277	0.928
784→	256→10	0.19	**0.932**
784→	512→128	0.308	0.928
784→	512→256→64	0.097	0.65
784→	512	0.908	0.929

The size and depth of the architecture affects the results. While deeper networks in principle offer more representational power, they may overfit or require more data and epochs to train effectively. In our gridsearch, the 512 → 256 → 64 stack model architecture underperformed, probably due to its complexity and insufficient training. Thus, the addition of a third hidden layer may only help if accompanied by additional optimisation efforts, such as a lower learning rate, more epochs, or batch normalisation. In contrast, a wide, shallow model with one layer of 512 neurons learns almost an identity map during pre-training and achieves an impressive 90.8% accuracy, but gains little from fine-tuning because there is not much to refine. In our training settings, narrow, two-layer models perform best, compressing more aggressively, looking weak at first but catching up as supervised error signals flow through all layers, all achieving a reasonable result of around 93% with very comparable small differences between them. In particular, the 256 → 10 network performs best.

Fine-tuning consistently improves classification accuracy, often significantly. The networks that contain a true bottleneck of less than or equal to 128 neurons sit at 19-31% accuracy before fine-tuning, but after 30 epochs of end-to-end training they all converge around 93%. The exception is the deeper stack of 3 layers, whose accuracy starts at 9.7% and reaches 65%, still a significant gain.

Pre-training gives each hidden layer a head start by forcing it to reconstruct its input, so that the network starts fine-tuning in a part of the weight space where activations are well scaled and gradients are stable. The advantage of this is that each layer is initialised with useful representations rather than random weights, especially for deeper networks. This often leads to better convergence during fine tuning and reduces the risk of getting stuck in bad local minima. This is also the reason why the 256 → 10 stack - starting from a modest 19% - can outperform the over-complete model in the fully trained state.

3.2.1

The dimensionality of the output of a convolutional layer is determinated as follows:

$$n_{out} = \left(\frac{n_{in} + 2p - k}{s}\right) + 1,$$

where n_{out} is the output size, n_{in} is the input size, p is padding size, k is kernel size and s is stride. Thus, for an input of height H_{in} and width W_{in}, kernel size of height k_h and width k_w, padding p, stride s:

$$H_{out} = \left(\frac{H_{in} + 2p - k_h}{s}\right) + 1 \; and \; W_{out} = \left(\frac{W_{in} + 2p - k_w}{s}\right) + 1.$$

In our case then:

$$H = \left(\frac{4-2}{2}\right) + 1 = 2 \; and \; W = \left(\frac{4-2}{2}\right) + 1 = 2.$$

Thus, an output is 2 by 2.

$$\begin{bmatrix} 2 & 5 \\ 3 & 1 \end{bmatrix} \rightarrow 2*1 + 5*0 + 3*0 + 1*1 = 3 = Y_{(0,0)},$$

$$\begin{bmatrix} 4 & 1 \\ 2 & 0 \end{bmatrix} \rightarrow 4*1 = 4 = Y_{(0,1)},$$

$$\begin{bmatrix} 4 & 5 \\ 1 & 2 \end{bmatrix} \rightarrow 4*1 + 2*1 = 6 = Y_{(1,0)},$$

$$\begin{bmatrix} 7 & 1 \\ 3 & 4 \end{bmatrix} \rightarrow 7*1 + 4*1 = 11 = Y_{(1,1)}.$$

Thus an ouput Y is as follows:

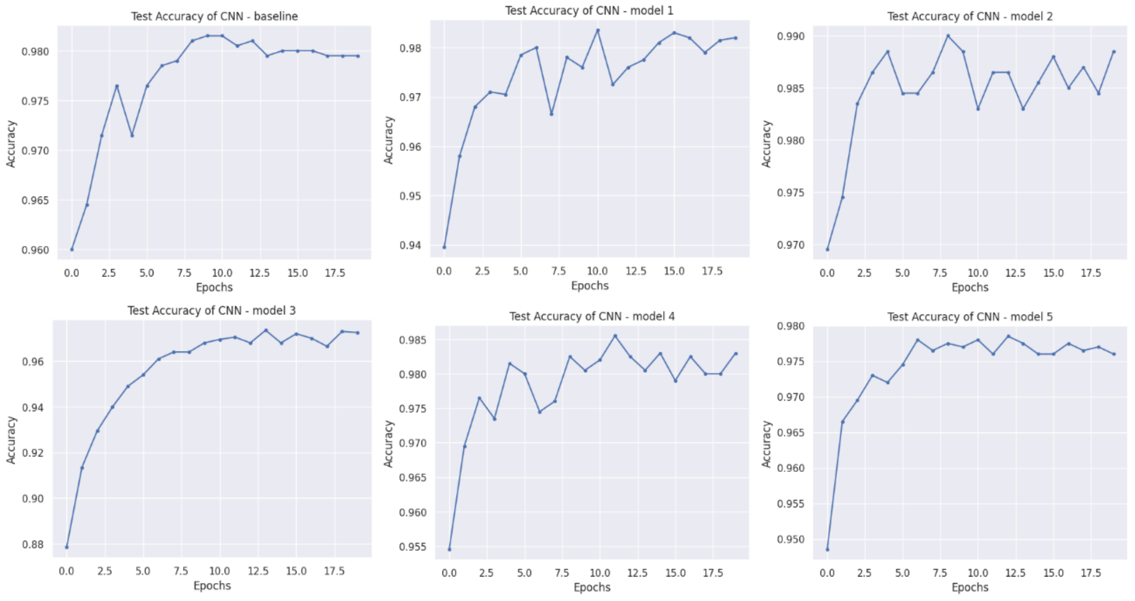
$$Y = \begin{bmatrix} 3 & 4 \\ 6 & 11 \end{bmatrix}.$$

Different from the fully connected layers where all nodes from subsequent layers are connected, in convolutional layer, neurons that are close to each other are likely to be more related than neurons that are further away. Due to this localized nature of convolution, CNNs are inherently better at detecting patterns regardless of their position in the input. A feature identified in one region can be detected just as effectively elsewhere, making CNNs robust to shifts and translations in the input. While fully connected layers ignore the spatial structure of data, CNNs explicitly leverage it, which makes them particularly well-suited for image-related tasks. Kernels in CNNs scan small local neighborhoods, aligning well with natural-image statistics where nearby pixels are highly correlated. Deeper layers then build on these local features to recognize more complex, global structures. In addition, CNNs drastically reduces the number of trainable parameters. This not only conserves memory and computation but also mitigates the risk of overfitting, especially when compared to dense, fully connected layers.

3.2.2

CNNs were trained on MNIST data using different architectures. For training, the number of epochs used is 20 and the batch size is 100. The final results are as follows:

Model	**Architecture**	**Trainable parameters**	**Test accuracy (vs. baseline)**
Baseline	4 convolution layers: 16-3×3→32-3×3→32-3×3→64-3×3, +Batch Normalization+ ReLU activation+ no pooling on every convolution layer→ 2 fully connected layers (100, 10)	~49 thousands	97.95%
Model 1	2 convolution layers: 16-3×3→32-3×3 +AdaptiveAvgPool 4×4→ 2 fully connected layers: 100→10	~57 thousands	98.2% (+0.25 pp)
Model 2	3 convolution layers: 32-3×3→64-3×3+MaxPool (2×2)+ Dropout(0.25)→128-3×3+MaxPool (2×2)+Dropout(0.25)→ 2 fully connected layers: 128→10	~897 thousands	**98.85% (+0.9 pp)**
Model 3	2 convolution layers: 8-3×3+MaxPool (2×2)→16-3×3→ 1 fully connected layer: 10	~11 thousands	97.25% (-0.7 pp)
Model 4	Model 2 + BatchNorm on every convolution layer	~897 thousands	98.30% (+0.35 pp)
Model 5	Baseline model + AdaptiveMaxPool 1×1	~30 thousands	97.6% (-0.35 pp)



Although Model 1 employs only two convolutional layers, making it the shallowest feature extractor, it surprisingly uses more parameters than the baseline. This is due to replacing the strided convolutions with an adaptive average pool that outputs a fixed 4×4 feature map before the classifier. It slightly outperforms the baseline at 98.2% test accuracy versus 97.95%, and converges smoothly by epoch 10. This demonstrates that decoupling resolution reduction from feature extraction improves optimization, stabilizes gradients, and accelerates training, while the shallower depth helps mitigate vanishing-gradient and overfitting issues.

Model 2, with its expanded capacity from tripled channel counts and dropout applied around both max-pooling layers, achieves the highest accuracy at 98.85%. It converges quickly due to its representational depth, showing how capacity combined with dropout fosters both strong learning and regularization. Model 4 builds on this by adding Batch Normalization after each convolution. While this results in slightly smoother training curves, it achieves only 98.30%, suggesting that in this case, dropout delivers more regularization benefit than Batch Normalization, which may introduce noise on datasets like MNIST.

Smaller networks can still perform competitively. Model 3, with just two narrow convolutional layers, pooling, and a single dense output layer, achieves 97.25% accuracy with only 11k parameters. It proves that aggressive pooling and minimal capacity can still extract sufficient digit structure, making it ideal for low-resource environments. Moreover, its training curve is notably stable with low risk of overfitting. Model 5 retains the baseline's four convolutional layers but replaces the 64×2×2 flattening step with an AdaptiveMaxPool2d(1×1), reducing the parameter count to about 30 thousands. It achieves 97.60%, hitting above 97.5% as early as epoch 6 and maintaining stable performance. This shows that global pooling is a highly efficient strategy for reducing fully connected layer size without compromising feature richness, offering a solid trade-off between capacity, regularization, and robustness.

In conclusion, Model 2 offers the best raw performance, but it comes with a high computational cost. If efficiency is the priority, Model 5 or Model 3 delivers respectable accuracy with minimal risk of overfitting and significantly smaller models. For a quick improvement over the baseline with little additional complexity, Model 1's adaptive pooling strategy can also be considered.

3.3.1

In our implementation, the NumPy version uses a batch of size 1 treating each of the samples as independent single token sequences, whereas the PyTorch version uses a batch size of 32 with sequence length 20. In general, the input is denoted by:

$$X \in \mathbb{R}^{B*N*d_{model}},$$

where B is number of samples in a batch, N is the sequence length for each sample, d_{model} is the dimension of each item in the sample sequence - embedding dimension. Regarding Queries Q, Keys K and Values V in a matrix form denoted as:

$$Q = XW_Q, K = XW_K, V = XW_V,$$

with $W_Q, W_k \in \mathbb{R}^{d_{model}*d_k}, W_V \in \mathbb{R}^{d_{model}*d_v}$. Hence, $Q, K \in \mathbb{R}^{B*N*d_k}, V \in \mathbb{R}^{B*N*d_v}$, where d_k is the dimensionality of each key (and query) vector and d_v is the dimensionality of each value vector. Attention raw score is calculated as:

$$S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{B*N*N},$$

The softmax function is applied independently to every row of its argument:

$$A_{b,m,n} = softmax(S_{b,m,n}) = \frac{\exp(S_{b,m,n})}{\sum_{n'=1}^{N} \exp(S_{b,m,n})},$$

yielding $A \in \mathbb{R}^{B*N*N}$. Attention output is denoted as:

$$Y = AV = AXW_V \in \mathbb{R}^{B*N*d_v}$$

Query vectors represent the questions that each position in the sequence is asking of other positions. Keys vectors represent the answers that each position provides to queries. Value vectors contain the actual content from each position.

3.3.1

The ViT models are trained on MNIST data using 20 epochs, batch size of 128 and Adam optimizer with an initial learning rate of 0.001.

Model	**dim**	**depth**	**heads**	**mlp_dim**	**final test accuracy**	**best epoch test accuracy**
baseline	64	6	8	128	98.38%	98.38%
smaller	32	4	4	64	97.66%	97.9%
wider	128	6	8	256	98.01%	98.22%
deeper	64	8	8	128	98.4%	98.48%
best	128	8	16	512	**98.58%**	98.6%

The performance results across different architectures highlight clear trade-offs in the size of the architecture. The baseline model of model dimension of 64, depth of 6 layers, 8 heads and MLP dimension of 128 achieves a solid 98.38% accuracy on test set, indicating a good performance and a strong reference point. The smaller model of lower dim, depth, heads and mlp_dim - 32, 4, 4, 64 respectively - shows a noticeable performance drop to 97.66% accuracy on test set, indicating that adequate model capacity remains important to achieve good performance. The wider model of higher model dimension of 128 and higher MLP dimension of 256 (with the baseline's depth and heads) performs slightly worse than the baseline achieving 98.01% accuracy on test set, suggesting that by only increasing representational width without complementary changes to depth and heads may not be beneficial and could result in underfitting. In contrast, the deeper model of 8 depth layers (with the baseline's dim, heads and mlp_dim) achieves 98.40%, indicating better performance, as additional transformer layers increase model capacity and better capture relevant patterns in the data, while maintaining good generalization. However, it's good to keep in mind that too complex models risk overfitting. The combination approach of increasing model dimension, depth, number of heads and MLP dimension compared to the baseline model (128 dim, 8 depth, 16 heads, 512 mlp_dim) yields the highest accuracy at 98.58%. We could further exploit more complex models with higher hyperparameters via tuning optimization methods based on validation set to prevent overfitting. However, it's worth noting that these models already achieve excellent performance, and the potential benefit from more complex models that are computationally more costly comes with a risk of overfitting and may offer limited additional gains on this dataset.

Exercise 4
4.1.
For Restricted Boltzmann Machines (RBM):

$$\frac{1}{N}\sum_{n=1}^{N}\frac{\partial \log P\left(v^{(n)};\theta\right)}{\partial w_{ij}} = E_{P_{data}}[v_i h_j] - E_{P_{model}}[v_i h_j],$$

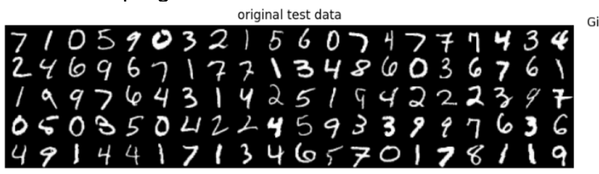
exact maximum likelihood learning is intractable because the expectation with respect to the model $E_{Pmodel}[\cdot]$ would require enumerating all configurations of v and h. Therefore, during training we sidestep the intractable expectation term with Contrastive Divergence (CD) algorithm:

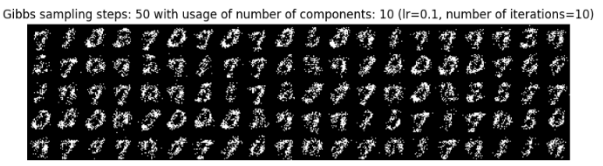
$$\Delta W = \alpha(E_{P_{data}}[vh^T] - E_{P_T}[vh^T]),$$

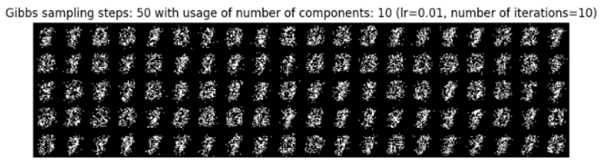
with α learning rate and P_T a distribution defined by running a Gibbs chain initialized at the data for T full steps. The Gibbs sample acts as a proxy for the model expectation. Because this is only an approximation, the algorithm is not guaranteed to follow the true ML gradient - yet in practice CD-1 is often sufficient to learn useful filters on images and other high-dimensional data. A tractable surrogate, the pseudo-likelihood, is used only to monitor whether learning is progressing, not for updating the parameters. For CD-1:

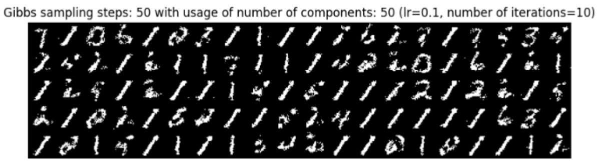
$$\Delta W \propto \left(v_n h^{(1)^T} - v^{(2)} h^{(2)^T}\right)$$

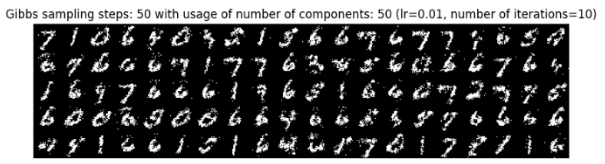
Hence the training loop tracks a surrogate objective, and the printed values should be interpreted as relative indicators of progress rather than exact measures of the model's probability of the data.

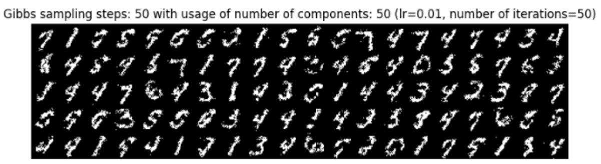

original test data


Gibbs sampling steps: 50 with usage of number of components: 10 (lr=0.1, number of iterations=10)

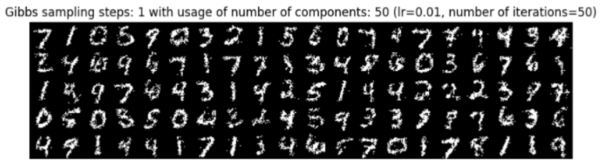

Gibbs sampling steps: 50 with usage of number of components: 10 (lr=0.01, number of iterations=10)

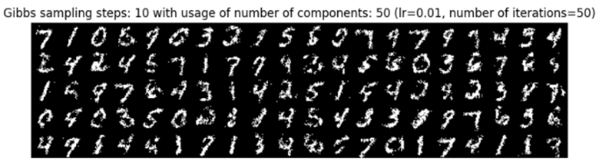

Gibbs sampling steps: 50 with usage of number of components: 50 (lr=0.1, number of iterations=10)

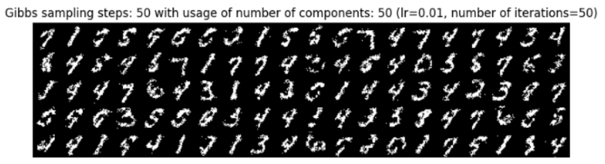

Gibbs sampling steps: 50 with usage of number of components: 50 (lr=0.01, number of iterations=10)

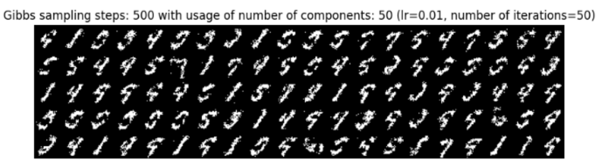

Gibbs sampling steps: 50 with usage of number of components: 50 (lr=0.01, number of iterations=50)

The number of components in an RBM determines how many latent feature detectors the model can allocate to the data, thereby defining its capacity. With only 10 components, the RBM lacks expressive power — reconstructed digits appear blurry, missing fine strokes and looking like averaged shapes. Increasing the number of components to 50, for example, gives the model enough capacity to capture loops, hooks, and cross-bars, resulting in sharper and more varied generated digits. However, having too many components can cause the model to memorise noise, slow down the sampling process, and overfit. The learning rate determines the size of the step in the CD gradient update. A large value, such as 0.1, enables the RBM to make significant jumps in parameter space and converge quickly. This results in sharp reconstructions, but the weights may overshoot optimal regions, causing digits to appear as speckles or thin vertical strokes. Reducing the learning rate to 0.01 produces gentler updates, enabling the model to refine itself smoothly and stably. The number of iterations corresponds to the number of training epochs - the number of times the model is allowed to pass over the data. With only 10 epochs, many digits still have missing strokes or resemble the mean digit. With 50 epochs, the strokes fill in and the contrast improves significantly. However, after the pseudo-likelihood plateaus, further training mostly increases the risk of overfitting, so monitoring this metric is essential.


Gibbs sampling steps: 1 with usage of number of components: 50 (lr=0.01, number of iterations=50)

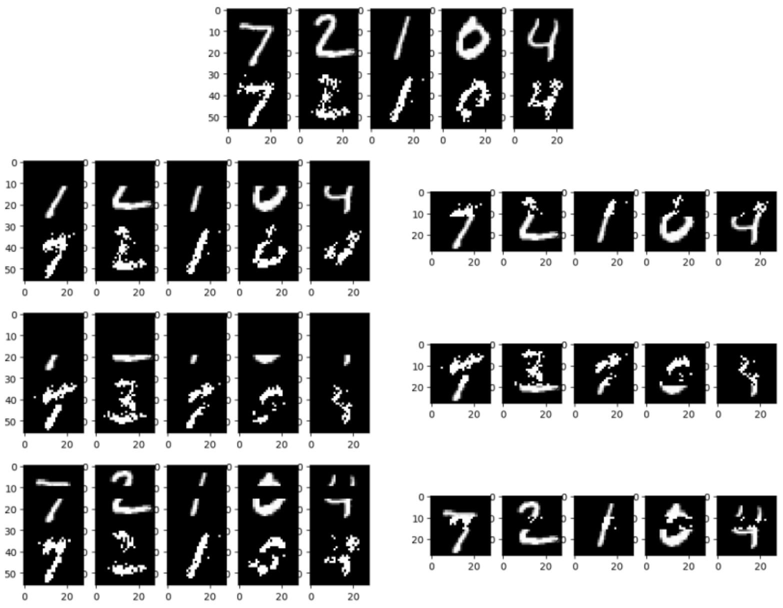

Gibbs sampling steps: 10 with usage of number of components: 50 (lr=0.01, number of iterations=50)


Gibbs sampling steps: 50 with usage of number of components: 50 (lr=0.01, number of iterations=50)

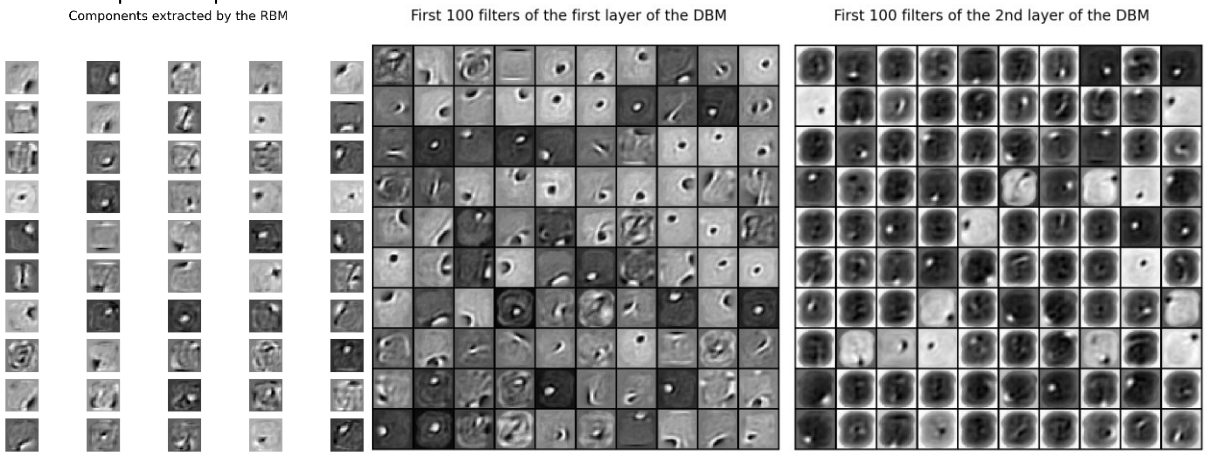

Gibbs sampling steps: 500 with usage of number of components: 50 (lr=0.01, number of iterations=50)

The number of Gibbs sampling steps is a post-training sampling parameter that determines how long we let the Markov chain run before the visible layer is inspected, thereby determining how far the sample can deviate from its initial state towards the RBM's equilibrium distribution (CD-k uses the same approach during training). With just one step, the hidden layer is toggled once and the visibles are resampled once, so most information from the starting

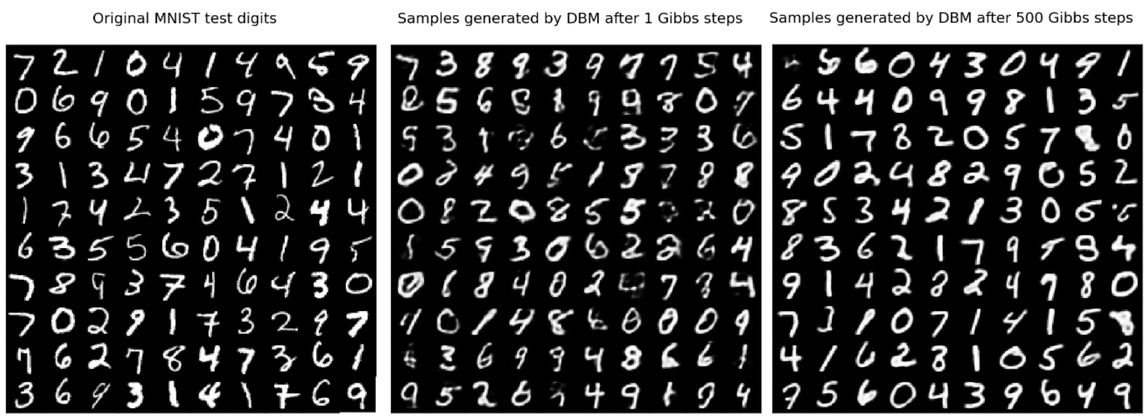
digit is retained and the images resemble slightly blurred reconstructions of the originals. Hundreds of steps, e.g. 500, push the chain much closer to the stationary distribution that the RBM has learned. In our case, this distribution is dominated by digits resembling 1 or 9, meaning almost every sample collapses to these shapes. The issue lies not with the sampling, but with the model's tendency to capture only a few strong modes while assigning too little probability to the rest. A few Gibbs steps produce a near-reconstruction of the input, whereas many steps reveal the RBM’s internal biases. With a moderate number of steps, e.g. 10 or 50, the chain begins to forget its starting point and gravitates towards regions of high probability under the model. If the model is well-trained and expressive enough, long chains produce diverse, realistic digits. If the model is underpowered or overfits certain patterns, however, long chains make that bias obvious. In our configuration, with 50 hidden components after 50 epochs, the model captures only a few strong modes, so very long chains are biased. After around 10 or 50 steps, well-shaped digits emerge that are noticeably different from the input, yet still retain the good shape of the original test data digits.
The reconstructions of the test data digits below, including the removal of pixel rows, were performed using the trained RBM model with 50 hidden components with a learning rate of 0.01 after 50 epochs:



The RBM can reconstruct a digit fairly well when the visible part contains enough of a distinctive shape to determine the correct class. However, if the visible fragment is consistent with several digits - for instance, a single vertical stroke that could be a 1 or a 7 - the model cannot make a principled choice, so the reconstruction may collapse into the wrong digit. Removing more rows makes the fragment increasingly ambiguous, resulting in incorrect digits or noisy patches in the reconstruction. Therefore, while an RBM can fill in missing regions by exploiting learned pixel correlations, the quality of the result depends heavily on how informative the observed portion is, as well as on the model’s expressive power.


Components extracted by the RBM
First 100 filters of the first layer of the DBM
First 100 filters of the 2nd layer of the DBM

The filters (interconnection weights) extracted from the previously trained RBM and by the Deep Boltzmann Machine (DBM) look similar only in the first layer of the DBM. In the RBM the filters are classic local shape detectors - edges, short diagonals, tiny loops. In the DBM’s first layer we see the same kinds of primitive patterns, but they are noticeably sharper and more contrasting, because top-down feedback from higher layers forces them to serve as reliable building blocks for abstract representations, thereby discouraging spurious patterns. The second DBM layer never sees raw pixels - instead, it observes the activation patterns of the first hidden layer. Its filters therefore differ from those of the RBM, capturing co-occurrence patterns of lower-level shapes, such as frequent sub-templates or larger parts of digits, rather than individual edges.


Original MNIST test digits
Samples generated by DBM after 1 Gibbs steps
Samples generated by DBM after 500 Gibbs steps

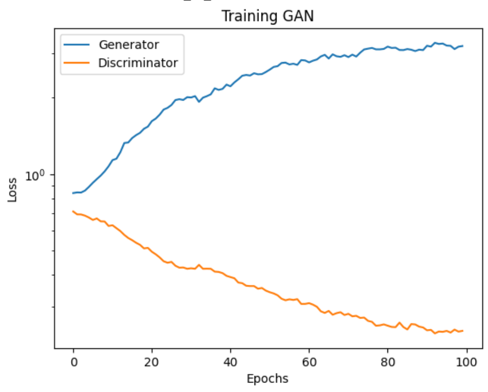
The DBM learns a hierarchical representation, with the first layer capturing primitive shapes and the deeper layers representing larger parts of digits. Thanks to this richer set of features, it produces cleaner and more diverse samples than a single-layer RBM. After just one Gibbs step, the DBM yields sharp, recognisable digits, and even after 500 steps, the images remain clear and diverse. By contrast, RBM samples are blurrier, and long chains often collapse to just a few shapes. The extra depth gives the DBM greater expressive power, shaping an energy landscape in which each Gibbs update can make broader, meaningful moves. This avoids falling into local traps and improves sample quality.

4.2

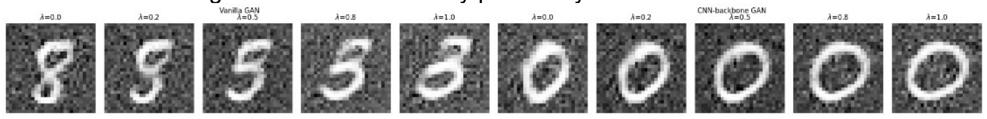
Generative Adversarial Network (GAN) trains two competing models in a zero-sum game: Generator G, which generates fake output examples from random noise, and Discriminator D, which discriminates between fake examples and real examples. During training each player maximises its own payoff, so at convergence:

$$G^* = \arg\min_G \max_D v(G, D),\ \ v\left(\theta^{(G)}, \theta^{(D)}\right) = E_{x_r \sim p_{data}} log D(x) + E_{z \sim p_z} log\left(1 - D\left(G(z)\right)\right).$$

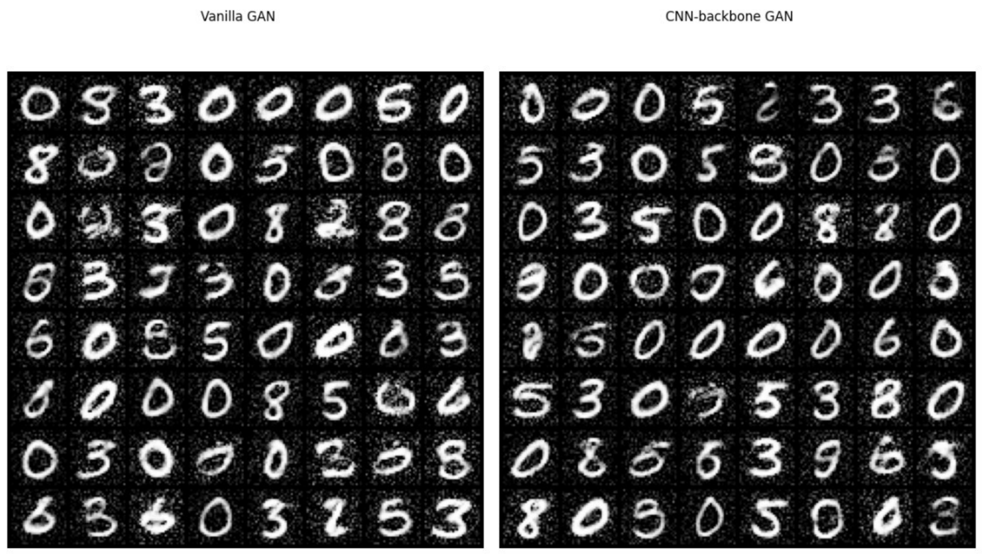
Because v is optimised in alternating steps, two separate losses appear. Discriminator loss, $\frac{1}{2}(BCE(D(x), 1) + BCE(D(G(z)), 0))$ (where BCE stands for bianry cross-entropy), which falls when D gets better at separating real from fake. Generator loss, $BCE(D(G(z)), 1) = -log\left(D\left(G(z)\right)\right)$, which falls only when the fakes look real according to D - non-saturating form is used to avoid vanishing gradients in the areas where D(x) is flat.



Each loss is minimised independently, so the corresponding curves must be read separately. In our run the discriminator curve plateaus low, the generator curve plateaus high (final averages ≈ 0.241 vs 3.19). D therefore out-performs G, so we expect less diversity and visible salt-and-pepper artefacts, i.e. repetitive digits on noisy backgrounds. The losses pattern signals the game has stabilised in a sub-optimal regime, where D dominates too much. Remedies such as using Wasserstein GAN may potentially further be considered to reduce this imbalance.

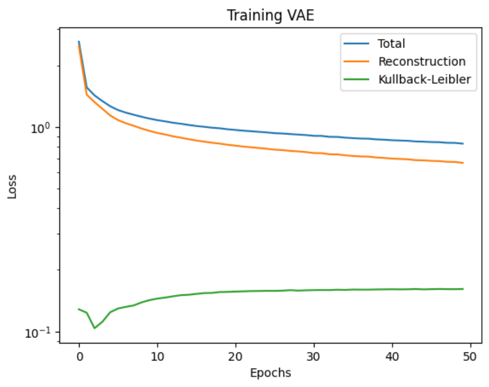


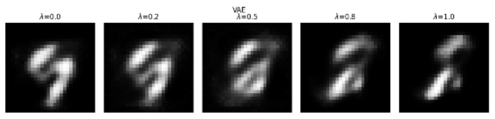
The latent space shows the interpolation of the two latent vectors and how λ blends them. In the vanilla GAN, the foreground digit morphs from an '8'-like shape to a '3'-like shape as λ increases, while the background noise barely changes. This means that the digit's identity and the background texture lie in largely orthogonal directions. Moreover, with the CNN backbone, the digit and background evolve smoothly together - the edges stay sharp and the speckle disappears, indicating a better organised manifold. Furthermore, the sample grid shows that the CNN-backbone GAN generates slightly more diverse digits than the vanilla GAN, confirming that the extra convolutional capacity helps the generator avoid mode collapse. Additionally, the samples are clean shapes with a small amount of background noise.

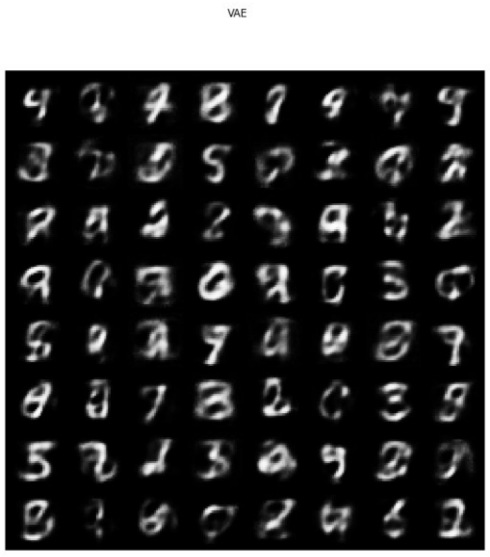


GAN-models offer the fastest inference thanks to one-pass inference and and a rich latent space. Their weaknesses are training fragility and the risk of mode collapse as keeping generator and discriminator in balance often requires hyper-parameter tuning, or regularization. VAEs train robustly and provide a tractable likelihood that is useful for anomaly detection, but reconstructions tend to be much more blurred than those produced by GANs. Diffusion models are almost as stable as VAEs and achieve good reconstructions, but they remain computationally heavy.

4.3







For a Variational Auto-Encoder (VAE), the intractable log-likelihood is replaced by the Evidence Lower BOund (ELBO). Applying Jensen's inequality gives:

$$\ln p(x) \geq E_{z\sim q_{\varphi}(z|x)}(\ln p(x|z)) - E_{z\sim q_{\varphi}(z|x)}\big(\ln q_{\varphi}(z|x) - \ln p(z)\big).$$

The first term is the (negative) reconstruction error, while the second is a regularizer corresponding to Kullback-Leibler $KL(q_{\varphi}(z|x)||p(z))$ that pulls the posterior $q_{\varphi}(z|x)$ toward the Gaussian prior $p(z)$. Maximising the ELBO therefore tightens this bound and indirectly increases the true log-likelihood.

Both VAE and the stacked autoencoder compress data through an encoder–decoder pair and use a pixel-level reconstruction loss. The stacked auto-encoder produces a deterministic code and optimises only that loss. In contrast, the VAE outputs a Gaussian distribution $N(\mu_{\varphi}, \sigma_{\varphi}^2)$ a and augments the reconstruction term (here, binary cross-entropy) with the KL penalty. This yields a principled latent space and enables sampling.

The resulting latent space of VAE shows a smooth transition, with no sudden jumps. That indicates the decoder learned a good manifold with the background noise barely changes. Furthermore, plotting random samples reveals good digit diversity, with all ten classes appearing, but the patterns lack sharp edges.

Thus, VAE samples are blurrier with little background noise but missing sharp edges. However, VAE samples display greater mode coverage and no collapse. GAN images are sharper, but they can repeat patterns or collapse to specific patterns without careful tuning. VAE training is single-objective and highly stable, whereas GAN training is fragile because the generator and discriminator influence must remain roughly balanced.

GenAI code of conduct for students (2024-2025)

*Generative AI (GenAI) assistance tools can be used to generate text, image, code, video, music or combinations of these***.** *It includes typical tools like (but this list is not limited to): ChatGPT, Google Gemini, MS Copilot, Midjourney, Claude.ai, Perplexity.ai, Dall-E, …*

Student name:	Bartosz Bogucki
Student number:	r1032166

Please indicate with "X" whether it relates to a course assignment or to the master thesis:

X This form is related to a **course assignment**.

Course name:	Artificial Neural Networks and Deep Learning
Course number:	H02C4a

O This form is related to **my Master thesis**.

Title Master thesis: ……………………………………………………………………………………………………

Promoter: ………

Daily supervisor: …………………………………………………………………………………………………

Please indicate with "X":

O **I did not use** any GenAI assistance tool.

X **I did use** GenAI Assistance. In this case **specify which ones** (e.g. ChatGPT, ...):

	Name of the GenAI tool used *Please indicate with "X" (possibly multiple times) in which way you were using GenAI:*				
GenAI assistance used as/for:	ChatGPT	DeepL Write			
Language assistance	X	X			
Search engine					
Literature search					
Short-form input assistance					
Generating programming code	X				
Generating new research ideas					
Generating blocks of text					
Other (specify):					

X **As a language assistant for reviewing or improving texts I wrote myself**

- *Code of conduct*: This use is similar to using spelling and grammar check tools, you do not have to refer to use of GenAI in the text.
  Be careful:
  - Using GenAI tools on texts you did not write yourself to cover up plagiarism (by paraphrasing original texts) is not allowed.

O **As a search engine to get information on a topic or to search for existing research on the topic**

- *Code of conduct*: This use is similar to e.g. a Google search or checking Wikipedia. If you then write your own text based on this information, you do not have to refer to use of GenAI in the text.
  Be careful:
  - Be aware that the output of the GenAI tool cannot be guaranteed as a 100% reliable source of information. The output may not be entirely correct and be limited due to the databases it uses. Knowledge evolves and may change over time, it may be that the database of the GenAI tool is not up to date.

O **For literature search**

- *Code of conduct*: This use is comparable to e.g. a Google Scholar search.
  Be careful:
  - Be aware that the output is restricted to the database it is built on. After this initial search, look for scientific sources and conduct your own analysis of the source documents. Interpret, analyse and process the information you obtained; verify it and don't just copy-paste it.
  - Be aware that some GenAI tools (like ChatGPT) may output no or wrong references. As a student you are responsible for further checking and verifying the absence or correctness of references; don't just copy-paste it.

  If you then write your own text based on this information, you do not have to refer to use of GenAI in the text.

O **For short-form input assistance**

- *Code of conduct*: This use is similar to e.g. Google docs powered by generative language models

X **To generate programming code**

- *Code of conduct*: Use of GenAI for coding should be explicitly allowed by the teacher. If used for coding, correctly mention the use of GenAI assistance and cite it.

O **To generate new research ideas**

- *Code of conduct*: Further verify in this case whether the idea is novel or not. It is likely that it is related to existing work, which should be referenced then.

O **To generate blocks of text**

- *Code of conduct*: Inserting blocks of text without quotes and a reference to GenAI assistance in your report or thesis is not allowed. According to Article 84 of the exam regulations in evaluating your work one should be able to correctly judge on your own knowledge, understanding and skills. In case it is really needed to insert a block of text from a GenAI tool, mention it as a citation by using quotes. But this should be kept to an absolute minimum. When you literally copy elements from a conversation with a GenAI tool: Quote between quotation marks and refer according to the specified reference style or as a personal communication within the text itself. Describe the use of the GenAI-tool (tool name, version, date, ...) in the method section (if there is one) and optionally add the (link to the) full conversation as an attachment.

O **Other**

- *Code of conduct*: Contact the professor of the course or the promoter of the thesis. Inform also the program director. Motivate how you comply with Article 84 of the exam regulations. Explain the use and the added value of ChatGPT or another AI tool: ….

Further important guidelines and remarks

- GenAI assistance cannot be used related **to data or subjects under Non-Disclosure Agreement.**
- GenAI assistance cannot be used related **to sensitive or personal data due to privacy issues**.
- **Take a scientific and critical attitude** when interacting with GenAI assistance and interpreting its output. Don't become emotionally connected to AI tools.
- As a student you are responsible for complying with Article 84 of the exam regulations: your report or thesis should reflect your own knowledge, understanding and skills. Be aware that plagiarism rules also apply to (work that is the result of) the use of GenAI assistance tools.
- **Exam regulations Article 84**: "Every conduct individual students display with which they (partially) inhibit or attempt to inhibit a correct judgement of their own knowledge, understanding and/or skills or those of other students, is considered an irregularity which may result in a suitable penalty. A special type of irregularity is plagiarism, i.e. copying the work (ideas, texts, structures, designs, images, plans, codes , ...) of others or prior personal work in an exact or slightly modified way without adequately acknowledging the sources. Every possession of prohibited resources during an examination (see article 65) is considered an irregularity."
- In order to maintain academic integrity and avoid plagiarism **more information about being transparent on the use of GenAI assistance and about citing and referencing GenAI** can be found on this website for students (Dutch/English).
- **Additional reading : KU Leuven guidelines on responsible use of Generative AI tools, and other information** (Dutch/English)

A few final words

If you are uncertain whether or not you should declare your use of AI tools, we suggest that you discuss the matter with your teacher or promoter. It is safer to declare AI use when it is not needed than to withhold that declaration when it is required.

Finally, remember that advanced AI tools are new and that they can do things they could not do, up until recently – so we do not have all the answers about how to use them responsibly yet. It is important to follow-up on most recent evolutions in AI technologies, to be a bit cautious, to communicate with teachers, teachings assistants, supervisors, promoters and peers with open minds, to be as transparent as we can, and to learn together as we move along.

This code of conduct can be used as a framework for academic year 2024-2025, changes will be made based on new evolutions.