Katholieke
Universiteit
Leuven

**Department of
Computer Science**

ARTIFICIAL NEURAL NETWORKS AND DEEP LEARNING

Lab Report

Chieh Fei (Jeffee) Hsiung (r0819932)

Academic year 2023–2024

Contents

Chapter 1

Supervised Learning

1.1 Section 1.3: A Small Model for a Small Dataset

1.1.1 Q1. What is the impact of the noise parameter in the example with respect to the optimization process?

Impact of noise parameters are assessed in a systematic approach. Initially with qualitative noise factorization where for each noise selection the signal-to-noise ratio (SNR) should be calculated. Subsequently, given knowledge of the SNR, noise-integrated model performance can be evaluated by analyzing the training-validation loss curve and learning curve. In this subsection, empirical small model performance with varying noise level is presented in Table 1.1.

Noise	SNR [dB]	Residual
0.1	9.7257	0.0106
0.3	0.5533	0.0879
0.9	-8.9133	0.7776
1.0	-10.1901	1.0433
1.3	-12.3810	1.7278
1.6	-14.2371	2.6492
1.9	-15.4338	3.4897
2.0	-15.9377	3.9190

Table 1.1: SNR table for various noise levels.

It is observed from Table 1.1 that a higher noise levels would lead to a harder accommodation to the noise-introduced variability in the loss landscape for the optimizer finding a clear path toward the minimum. Additioanlly, there exhibit a greater model overfitting risk to the noisy data such that the noise is captured and learned as meaningful pattern yielding inaccurate prediction and degrading generalizability.

Using LBFGS with Noisy Data: LBFGS, being a second-order optimization method, is sensitive to the quality of the gradient information as the LBFGS updates guiding Hessian approximation is highly noise-suscepted, such that the resulting updates would be varying according to noise. Implementing mechanisms such as line search strategies (like 'strong_wolfe') enables LBFGS with adaptive step size in response to the noise, attempting to ensure that each step improves the loss in a meaningful way, despite the noise.

Hypothetical Outcomes and Visualizations: The training loss over epochs would likely show more fluctuations with higher noise levels. The convergence might be slower or less smooth compared to training on less noisy data. In addition, the squared residuals, defined as the squared differences between predictions and actual values, would likely show higher values on average, indicating greater prediction error due to the noise. The experimental training and validation loss results demonstrated in Table 1.2 shows alignment with the hypothetical expectation.

Experimental Result

1.1.2 Q2. How does (vanilla) gradient descent compare with respect to its stochastic and accelerated versions?

Vanilla Gradient Descent: In the context of neural network training, Vanilla Gradient Descent refers to the simplest form of gradient descent optimization algorithm, functioning as a first-order iterative method for

	Training error	Residual	Lowest loss	Best epoch	SNR [dB]
0.3	0.054652	0.054652	0.054655	1999	5.529366
0.9	0.480521	0.480521	0.480558	1999	2.698386
1.3	1.058633	1.058633	1.058636	1999	2.283705
1.6	1.632100	1.632100	1.632197	1999	2.244913
2.0	2.524409	2.524409	2.524705	1999	1.954540

Table 1.2: Training and validation loss results for different noise levels.

minimizing a function. The primary objective of vanilla gradient descent is to reduce a loss function, which quantifies the discrepancy between the neural network's predicted output and the actual target values.

This algorithm updates all model parameters simultaneously by taking steps proportional to the negative gradient (or approximate gradient) of the loss function concerning those parameters. The update rule is defined as $\theta = \theta - \eta \nabla_\theta J(\theta)$, where θ denotes the model parameters, η represents the learning rate (a small positive hyperparameter dictating the step size), and $\nabla_\theta J(\theta)$ signifies the gradient of the loss function $J(\theta)$ with respect to the parameters.

The term "vanilla" indicates the most basic form of gradient descent, devoid of enhancements such as momentum or adaptive learning rates. This approach involves a comprehensive computation of the gradient using the entire dataset, rendering it computationally expensive and slow for large datasets. Additionally, vanilla gradient descent can be slow to converge, particularly in loss function landscapes characterized by shallow gradients, saddle points, or local minima.

Stochastic Gradient Descent (SGD): While sharing the objective of minimizing the loss function, differentiates itself by updating the SGD model parameters using the gradient computed from a randomly selected subset of the data (a mini-batch) rather than the entire dataset. This approach significantly accelerates computation and permits more frequent updates. The inherent randomness introduced by the use of mini-batches injects noise into the optimization process, which can be beneficial in escaping local minima, thereby enhancing the optimization performance.

1.1.3 Q3. How does the size of the network impact the choice of the optimizer?

The size of the neural network can indeed impact the choice of the optimizer. The design choice of optimizer is neural network dimension dependent, which concerns can be categorized as follows:

- **Memory Usage:** memory space consideration address the additional optimizer-specific parameters or states that demands supplementary memory space for supplementary, especially in the case of exponentially decaying past gradients average exhibit in optimizers such as Adam, RMSProp, and other adapive learning rate methods. This can be problematic for very large networks with limited memory.
- **Convergence Speed:** For large networks, while speed of convergence becomes influential, optimizes such as SGD with momentum or adaptive learning rate methods like Adam can converge faster than vanilla SGD that is beneficial for large networks.
- **Generalization:** large networks may benefit from the SGD optimizer with higher generalizability, while utilization for detailed adaptive methods may lead to overfitting.

In the experiment, we compare the convergence speed and generalization performance of different optimizers (e.g., SGD, Adam, RMSProp) on networks of varying sizes, which is illustrated in Table 1.3.

	Optimizer 1	Optimizer 2	Optimizer 3	Optimizer 4	Optimizer 5
10	2445	2208	1985	2500	2499
50	2475	2400	2465	2499	2500
100	2359	2460	2430	2499	2500

Table 1.3: Comparison of different optimizers and network sizes.

1.1.4 Q4. Discuss the difference between epochs and time to assess the speed of the algorithms. What can it mean to converge fast?

Both the number of epochs and the time taken address time-related resources but measure different aspects of the optimization algorithms performance in neural networks training.

An epoch is a single pass through the entire training dataset. In terms of epochs, convergence speed refers to the number of epochs required for an algorithm to reach a certain accuracy or minimize the loss function to a predefined threshold. This metric assesses the learning process efficiency in terms of dataset utilization.

Time, on the other hand, refers to the actual duration taken by the algorithm to reach convergence, measured in seconds or minutes. Here, convergence speed refers to how quickly an algorithm can achieve a specified performance level. Time serve as a practical indicator for model selection in real-world applications with limited computational resources.

Comparing epochs and time for convergence offers different perspectives on algorithm efficiency. Evaluating based on the number of epochs emphasizes the model's data learning efficiency, abstracting from computational aspects. In contrast, considering time provides a holistic view, incorporating both learning efficiency and computational cost.

1.1.5 Q5: A Bigger Model: Number of Parameters in the Model

The model's total number of parameters can be calculated by examining each layer's contribution, which formula for 2-D convolution is expressed as:

Conv2D Layers: (kernel width $\times$ kernel height $\times$ input channels $+ 1$) $\times$ number of filters

Dense Layers: (input units $+ 1$) $\times$ output units

Let's calculate the number of parameters for the provided model:

- First Conv2D Layer: $(3 \times 3 \times 1 + 1) \times 32 = 320$
- Second Conv2D Layer: $(3 \times 3 \times 32 + 1) \times 64 = 18,496$
- Dense Layer: $(7 \times 7 \times 64 + 1) \times 10 = 31,750$

The dropout layer does not add any parameters; it only acts during training by randomly setting a fraction of the input units to 0 at each update during training time to prevent overfitting.

1.1.6 Q6: A Bigger Model: Replacing the Optimizer

- **Adam to SGD:** Adam is an adaptive learning rate optimizer that combines the best properties of the AdaGrad and RMSProp algorithms. SGD maintains a single learning rate for all weight updates and the learning rate does not change during training. Changing Adam to SGD might slow down the convergence, and you might need to fine-tune the learning rate and possibly add momentum to achieve similar performance.
- **SGD to Adadelta:** Adadelta is an extension of AdaGrad aiming to reduce its aggressive, monotonically decreasing learning rate. It adapts over time and does not require a learning rate to be specified. Adadelta's performance compared to Adam or SGD can vary depending on the task and specific model architecture.

Experimental Result:

Optimizer	Test Loss	Test Accuracy
ADAM	0.025648	0.9909
SGD	0.022971	0.9917
Adadelta	0.022916	0.9918

Table 1.4: Comparison of optimizer performance on test set.

1.2 Section 1.4: A Personal Regression Exercise

1.2.1 Q1: Training and Testing Dataset Spliting and Plotting

The splitting should be done randomly to ensure that both datasets are representative of the overall data distribution. Additioanlly, having different datasets for Training and Testing is essential for model generalization evaluation. The training dataset is employed to fit the model, allowing it to learn the underlying patterns within the data. Conversely, the testing dataset, comprising unseen data, is used to assess the model's performance on new, unseen data. It is forbidden to a single dataset for both training and testing as it would

obscure whether the model has merely memorized the data (overfitting) or genuinely learned to generalize from the observed patterns during training. This distinction is essential to ensure that the model performs well on real-world, unseen data.

1.2.2 Q2: The Neural Network Architecture Hyperparameters Tuning

Model Selection: The model selection process involves experimenting with different hyperparameters to find the combination that yields the best performance on the training set. This includes:

- **Number of Layers:** Experiment with different numbers of hidden layers and units in each layer to find the architecture that best captures the underlying patterns in the data.
- **Learning Algorithm:** Try different optimization algorithms (e.g., SGD, Adam, RMSProp) and learning rates to find the one that converges most effectively.
- **Transfer Function:** Experiment with different activation functions (e.g., ReLU, Sigmoid, Tanh) to find the one that best captures the non-linear relationships in the data.

Hyperparameter	Options
Number of layers	1, 2, **3**
Units per layer	3, 10, **20**
Activation function	**tanh**, relu
Learning rate	**0.01**, 0.001, 0.0001

Table 1.5: Hyperparameters tuning for model selection

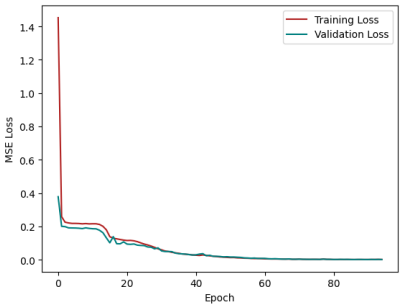


Table 1.6: Training and validation loss curve.

Validation: Separate validation sets (distinct from the training and testing sets) and cross-validation techniques assessing model performance across different subsets of the training data are commonly utilized to validate performance during training prior to testing. The approach allows for overfitting monitoring and consequenly ensuring generalizability. The following best mdoel parameters (Table 1.5) via hyperparameter tuning based on validation and its corresponding loss curve is demonstrated in Table 1.6. The parameter selection is based on grid search such that lowest validation loss expressed in bold font was opted for. For large networks, sophisticated techniques and libraries designed for hyperparameter optimization such as `keras-tuner` or `scikit-learn`'s `GridSearchCV` can be employed.

1.2.3 Q3: The Neural Network Training and Testing

Performance Evaluation: The Final Mean Squared Error (MSE) on the test set as a quantitative measure of the model's performance is empirically determined: $MSE = 0.23111077279740277$, which tields a prediction surface and delta presented in Figure 1.1 and Figure 1.2.

The optimal model configuration, consisting of 2 hidden layers with 20 units each, ReLU activation, and a learning rate of 0.01, demonstrated low training and validation errors, indicating robust generalization. Further training would likely result in overfitting, with decreasing training error but increasing validation error, thereby degrading performance on unseen data. Thus, balancing training and validation errors is essential for achieving good generalization. The current model configuration indicates strong generalization to new data, as evidenced by the low test error.

1.2.4 Q4: The Neural Network Regularization Strategies

Regularization strategies for preventing overfitting, in its basic form, includes techniques such as L1 and L2 regularization, dropout, and early stopping; in a more advance approach, the strategies includes data augmentation, batch normalization, and model ensembling.

L1 and L2 Regularization: L1 and L2 regularization involve adding a penalty to the loss function based on the magnitude of the model parameters. L1 regularization adds the absolute value of the weights to the

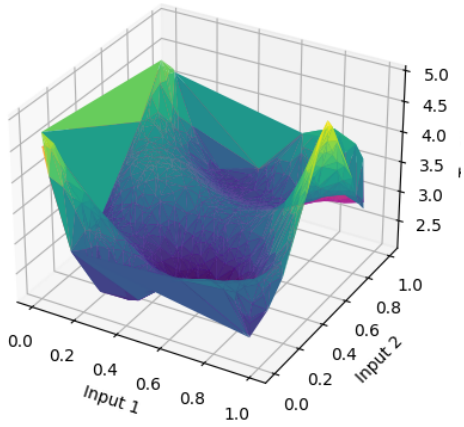


Figure 1.1: Surface of the test set and the approximation given by the network.

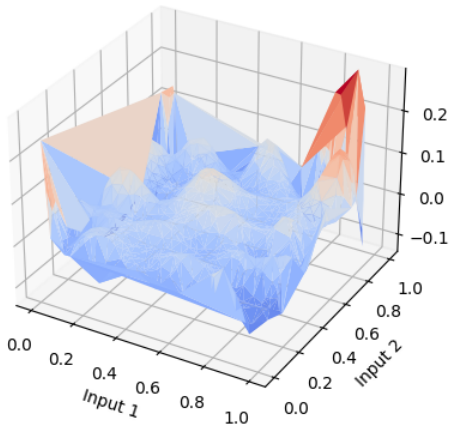


Figure 1.2: Delta between prediction and actual value.

loss function, encouraging sparsity in the model parameters, while L2 regularization adds the squared value of the weights, which tends to distribute error across all parameters, leading to more stable and generalized models.

Dropout: Dropout is a technique where randomly selected neurons are ignored during the training process. This means that their contribution to the activation of downstream neurons is temporarily removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass. Dropout prevents neurons from co-adapting too much, forcing the network to learn more robust features that are useful independently of other neurons.

Early Stopping: Early stopping is a simple yet effective regularization technique where the training process is halted as soon as the performance on a validation dataset begins to deteriorate. This prevents the model from overfitting to the training data by stopping training at the point where the model performs best on the validation data.

Data Augmentation: Data augmentation involves generating new training samples by applying random transformations such as rotation, scaling, and flipping to the existing training data. This increases the diversity of the training data, helping the model generalize better by learning from a broader range of input variations.

Batch Normalization: Batch normalization normalizes the inputs of each layer so that they have a mean of zero and a standard deviation of one. This technique helps to stabilize and accelerate the training process by reducing the internal covariate shift, which allows the network to learn more robustly and reduces the need for dropout.

Model Ensembling: Model ensembling involves training multiple models independently and then combining their predictions. This can be done by averaging the predictions (bagging) or by using a weighted average. Ensembling tends to improve the performance of models by reducing variance and leveraging the strengths of different models, thereby improving generalization.

Chapter 2

Recurrent Neural Networks

2.1 Section 2.1: Hopfield Network

2.1.1 Q1. Hopfield network with target patterns [1, 1], [-1, -1], and [1, -1] and the corresponding number of neurons.

Attractor Values For random inputs, the final states of the vectors converge to one of the target patterns ([1, 1], [-1, -1], and [1, -1]). This convergence can be understood through the energy function of the Hopfield network, given by:

$$H = -\frac{1}{2}\sum_{i,j} w_{ij} S_i S_j, \tag{2.1}$$

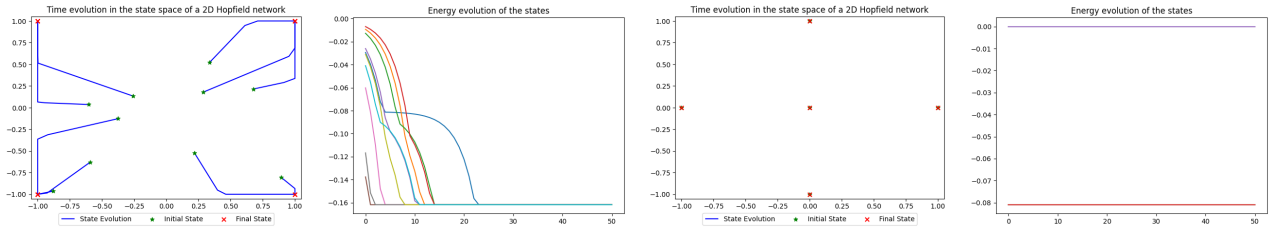
where w_{ij} are the weights, and S_i and S_j are the states of the neurons. The network evolves by updating the neurons to minimize this energy function, converging to local minima representing the stored patterns.

In some cases, the network converges to [-1, 1], which is not one of the original target patterns. This state is known as a spurious state and arises due to the symmetric nature of the Hopfield network weights, which inherently support both the target pattern and its inverse as stable states. The time evolution of these states is illustrated in Figure 2.1.

Unwanted Attractors The presence of unwanted attractors such as [-1, 1] introduces local minima due to interactions between the neurons in the network. These spurious states occur because when storing a pattern ξ, the network also stores $-\xi$ due to the symmetric weights. This is explained by the formula:

$$h_i^\nu = \xi_i^\nu + \frac{1}{N}\sum_j \sum_{\mu\neq\nu} \xi_i^\mu \xi_j^\mu \xi_j^\nu, \tag{2.2}$$

where the crosstalk term $\frac{1}{N}\sum_j \sum_{\mu\neq\nu} \xi_i^\mu \xi_j^\mu \xi_j^\nu$ can lead to the formation of spurious states if its absolute value is less than 1.



(a) Random input state evolution. (b) Energy evolution (random input). (c) Symmetric input state evolution. (d) Energy evolution (symmetric input).

Figure 2.1: Energy and time evolution in random and symmetric input states.

For symmetric inputs, such as [1, 0], [0, 1], [-1, 0], and [0, -1], the final states are not among the original target patterns but instead range between them. These attractors represent states of high symmetry and are sometimes referred to as points of unstable equilibrium in the Hopfield network, referred to as spurious states given the symmetry nature that when storing ξ^ν, automatically also $-\xi^\nu$ is stored. The network does

not converge to a specific attractor but rather to a state that is equidistant from multiple attractors. This is especially noticeable when the final state [0, 0] is exactly in the middle of all attractors, essentially representing an unstable point. The energy evolution of these symmetric states is depicted in Figure Figure 2.1.

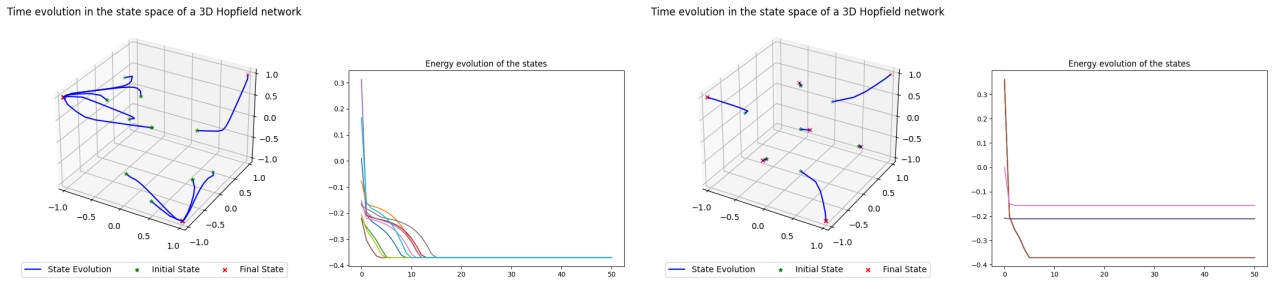
The state-dependent number of iterations for reaching an attractor exhibits a slower convergence for initial states located near the decision boundary, between basins of attraction. The stability of the attractors in the network varies based on the patterns. The attractors [1, 1], [-1, -1], and [1, -1] are stable as they are the intended patterns that the network was trained on. The spurious attractor [-1, 1] is also stable, despite being a undesired state. For symmetric inputs, as the network does not converge to the target patterns but rather to intermediate states the attractors are unstable.

2.1.2 Q2. Hopfield network with target patterns [1, 1, 1], [-1, -1, -1], [1, -1, 1] and the corresponding number of neurons.

For random inputs, the network typically converges to one of the target patterns ([1, 1, 1], [-1, -1, -1], or [1, -1, 1]), indicating effectively recalling the stored patterns from corresponding initial states. The target-pattern-matching stable final states suggest that patterns learned being stable attractors within the network's energy landscape, which states of the neurons are updated to minimize and reaching the local minima of the energy function (Equation 2.1). The energy evolution of random input states can be visualized in Figure 2.2.

For symmetric inputs, owing to the high-symmetry the converging intermediate states do not match the target patterns, which reason is similar to the high-symmetric near-boundary states positions disucessed in subsection 2.1.1. This is edivent by the appearance of states [1, 0.06045472, -0.06045472], suggesting quasi-symmetric inputs pattern convergences are highly influenced by the surrounding attractors.

The energy evolution of symmetric input states is shown in Figure 2.2.



(a) 3D random input state evolution. (b) Energy evolution (3D random input). (c) 3D symmetric input state evolution. (d) Energy evolution (3D symmetric input).

Figure 2.2: Energy and time evolution in 3D random and symmetric input states.

The network's failure to converge to the target patterns for symmetric inputs suggests the presence of unstable equilibrium points (spurious states), which existence especially in simulations with symmetric inputs, points to the presence of mixed states due to the interplay of attractors. T

The interplay-complexity dependent convergence and resulting spurious states are particularly notable high-dimensional input spaces associated with the storage capacity Equation 2.3, which worsens further if with symmetrical initial states. The concept of storage capacity is given by:

$$p_{\max} = \frac{N}{4} \log N \quad \text{(requiring perfect recall)} \tag{2.3}$$

suggesting that the probability of encountering spurious states is positively proportional to the ratio p/N.

2.1.3 Q3. Create a Higher Dimensional Hopfield Network Which Has as Attractors the Handwritten Noisy Digits from 0 to 9.

Owing to the presence of spurious states and the potential inherent complexity, noise embedded in the input would exacerbate the convergence performance of Hopfield such that the number of iterations required for the model to recover the original pattern is significant.

Higher noise levels can result in longer convergence times or even prevent the network from reaching the correct attractor such that the network is stuck in local minima or oscillate between states when the noise level is high, resulting in prolonged convergence times or failure to converge.

Conversely, low levels of noise may still allows for correct yet distorted attractor convergence. Inference of noisy input hence may lead to a spurious state or a mixed state convergence, which phenomenon can be represented by the crosstalk terms in the weight matrix as mentioned in subsection 2.1.1.

2.2 Section 2.2: Time-Series Prediction with Santa Fe Laser Dataset

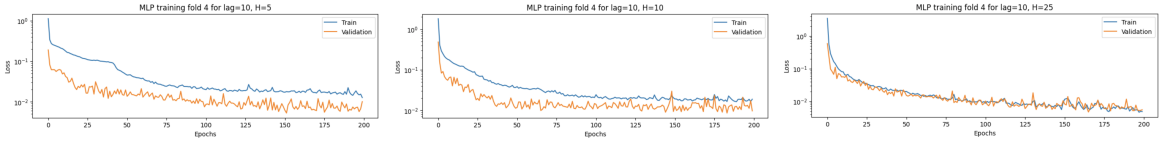
2.2.1 Q1. Implement a Time-Series Prediction MLP Network for the Santa Fe Laser Dataset

The lag parameter in a time-series prediction model determines the number of previous time steps used as input features. A higher lag value captures more historical information but can lead to overfitting and increased computational resource usage. Conversely, low lag values may provide insufficient historical context, resulting in poor prediction accuracy and generalization. An optimal lag value balances capturing relevant information and maintaining manageable model complexity, which is determined through experimentation and validation.

The validation size, specified by the *validation_size* parameter, is the fraction of the dataset set aside for validation, significantly affecting model training reliability, generalizability, and performance. The $validation\_folds$ parameter indicates the number of splits used in the cross-validation process, where more folds typically provide a more accurate estimate of model performance.

Model performance with different lags and numbers of neurons in the MLP's hidden layer was investigated through a grid search and cross-validation. The optimal combination of lag and neuron count was identified based on the lowest Mean Squared Error (MSE) on the validation set without early stopping.

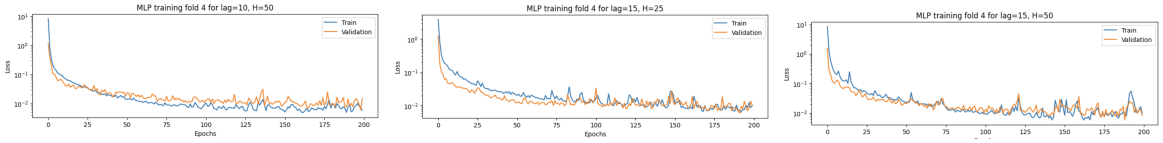
Visualizations of training and validation loss curves (Figure 2.3) were used to select the best hyperparameters. The combination of lag 10 and 25 hidden units emerged as the optimal parameter set, while higher lags tended to overfit towards the end of epochs. However, combinations of lag 10 or 15 with 25 hidden units may perform better with early stopping, which was subsequently validated.



(a) 4 validation folds, 10 lags, 5 hidden units.

(b) 4 validation folds, 10 lags, 10 hidden units.

(c) 4 validation folds, 10 lags, 25 hidden units.



(d) 4 validation folds, 10 lags, 50 hidden units.

(e) 4 validation folds, 15 lags, 25 hidden units.

(f) 4 validation folds, 15 lags, 50 hidden units.

Figure 2.3: Loss curves for different configurations.

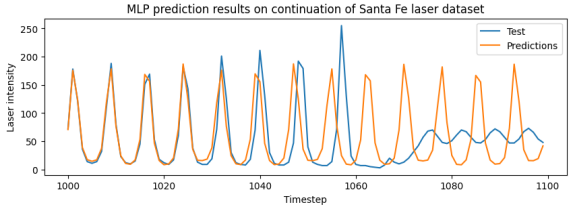
The network configured with each set of hyperparameters during tuning, gives prediction results on validation set are illustrated in Figure 2.4.

Aligning with the expectation derived from the loss curve (Figure 2.3), combination of lag 15 with 25 hidden units stands as the the optimal best model option, which could not be captured if without early stopping implementation.

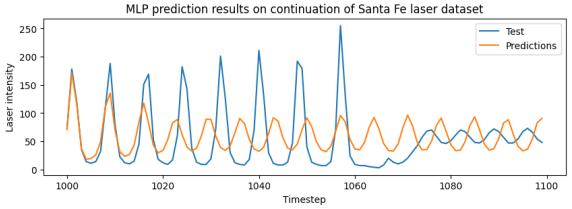
2.2.2 Q2. Implement LSTM model: explain the design process and effect of lag value.

The LSTM model, implemented to predict the Santa Fe Laser dataset, conclude its design post-hyperparameter tuning as explained in subsection 2.2.1, giving a prediction presetned in Figure 2.4d.

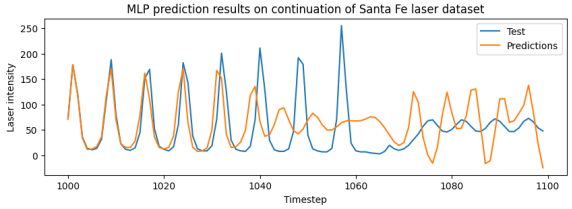
The effect of changing the lag value on the LSTM network was thoroughly examined to comprehend how the historical context affects prediction performance. The performance on the validation set was optimized by evaluating various combinations of lag values and hidden units. Specifically, lag values of 1, 5, 10, and 15, combined with hidden units of 5, 10, 25, and 50, were tested. Among these combinations, the configuration



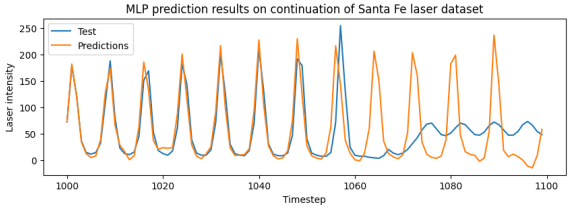
(a) MLP Lag=5 H=25 prediction results on continuation of Santa Fe laser dataset.



(b) MLP Lag=5 H=50 prediction results on continuation of Santa Fe laser dataset.



(c) MLP Lag=10 H=25 prediction results on continuation of Santa Fe laser dataset.



(d) MLP Lag=15 H=25 prediction results on continuation of Santa Fe laser dataset.

Figure 2.4: MLP prediction results on continuation of Santa Fe laser dataset for different configurations.

Lag	H=5	H=10	H=25	H=50
1	0.628	0.644	0.650	0.622
5	0.112	0.046	0.058	0.087
10	0.180	0.068	0.176	0.205
15	0.136	0.149	0.275	0.532

Table 2.1: MSE for different lags and hidden units without early stopping.

with lag 15 and 25 hidden units yielded the best performance on the cross-validation set, achieving the lowest mean squared error (MSE) of 0.110.

Extended captures specified by the lag value influence significantly the LSTM model's ability to comprehend the temporal dependencies in the data. A higher lag value within an data-dependent upper limit improves the prediction accuracy if the relevant patterns exist in the extended history as illustrated in the progressively improving MSE in Table 2.1. However, excessively high lag values can introduce unnecessary complexity and potential overfitting, where the model captures noise rather than meaningful patterns.

2.2.3 Q3. Recurrent MLP and LSTM Comparison

Both the LSTM and MLP models, using a lag of 15 and 25 hidden units, were employed to predict the continuation of a time series from the Santa Fe laser dataset, .

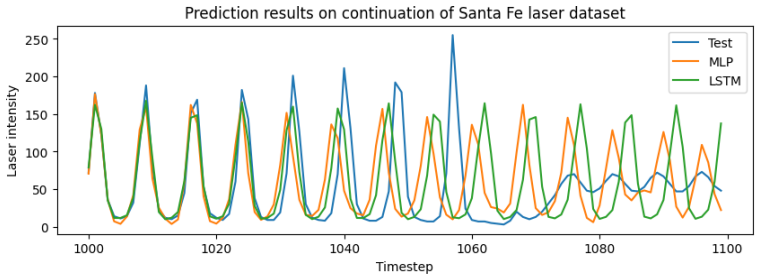


Figure 2.5: Lag=15 H=25 LSTM vs. MLP prediction results on continuation of Santa Fe laser dataset.

While both LSTM and MLP could not follow accurately the test data post timestep 1040, comparatively the LSTM model (green line) demonstrates a superior performance in closely tracking bandwidth and peak locations of the test data (blue line), effectively capturing the rhythm and magnitude of the oscillations than the results of MLP (orange line). The LSTM shows a higher degree of accuracy; conversely, the MLP model, while capturing the overall trend, struggles with the finer details of the test data's fluctuations. Post timestep 1060, the MLP's predictions deviate significantly, failing to accurately predict the peaks and troughs, resulting

in a noticeable performance decline.

Acknowledging the visual comparison, the LSTM model is preferred. LSTMs are specifically designed to handle sequences involving complex temporal patterns and potential long-term dependencies, leveraging their recurrent connections to better memorize and predict the sequence of laser intensity.

Chapter 3

Deep Feature Learning

3.1 Section 3.1: Autoencoders and Stacked Autoencoders

3.1.1 Q1. Image Reconstruction on MNIST Dataset Using an Autoencoder: Tuning Hidden Layer Neurons and Training Epochs to Improve Performance

To ensure an unbiased performance measure on unseen data, a separate validation set is used for parameter tuning, while the test set is reserved for final evaluation. The training data is split into a new training set (80%) and a validation set (20%), with the latter used for hyperparameter tuning. The model undergoes training on the adjusted training set, validation against the validation set, and final evaluation on the test set to assess its generalization capability.

Based on the validation set performance, various hyperparameters were tested, including batch size, encoding dimension, and the number of epochs. The results are summarized in the following table:

Batch Size	Encoding Dimension	Avg Val Loss (40 Epochs)	Avg Val Loss (60 Epochs)
16	32	0.0179	0.0127
16	64	0.0048	0.0054
16	128	0.0018	0.0018
32	32	0.0141	0.0139
32	64	0.0047	0.0055
32	128	0.0018	0.0018
64	32	0.0154	0.0140
64	64	0.0055	0.0053
64	**128**	**0.0018**	**0.0017**

Table 3.1: Hyperparameters tuning results for autoencoder on MNIST dataset.

The best hyperparameters configuration for the autoencoder model, based on the lowest average validation loss of 0.0017278431582575043, was found with a batch size of 64, an encoding dimension of 128, and 60 epochs. This configuration, when evaluated on the unseen data yielded a loss of 0.000227, comparable to the original test loss of 0.001470 using the initially provided parameters configuration of batch size 32, encoding dimension 32, and 20 epochs.

3.1.2 Q2. Improve MNIST Image Classification Using a Stacked Autoencoder: Network Architecture Adjustments, Fine-Tuning Results, and Benefits of Layer-Wise Pretraining

Table 3.2: Stacked Autoencoder Configuration and Test Accuracy

Batch Size	Layer-wise Epochs	Classifier Epochs	Fine-tuning Epochs	Test Accuracy
128	10	10	10	0.926
64	60	60	60	0.949
64	60	60	60 (Hidden Dim 128)	0.954

The baseline stacked autoencoder model with configuration outlined in Table 3.2 achieved an accuracy of 0.926 on the test set post fine-tuning. Nonetheless, by altering the batch size to 64 and increasing the

number of epochs to 60 for both layer-wise and classifier training, the model's test accuracy improved to 0.949. Finally, further adjustments as made by layer's hidden dimension from 256 to 128 stemming from the empirical observance, the resulting test accuracy achieved an even higher value of 0.954. by optimizing the model hyperparameters, optimal configuration was identified for this single autoencoder model.

3.2 Section 3.2: Convolutional Neural Networks

3.2.1 Q1. Answer the following questions: Consider the following 2D input matrix.

$$X = \begin{bmatrix} 2 & 5 & 4 & 1 \\ 3 & 1 & 2 & 0 \\ 4 & 5 & 7 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Q1.1. Calculate the dimensions of output of a convolution with the following 2x2 kernel with no padding and a stride of 2.

Given the input matrix X and the kernel K, with no padding and a stride of 2, the dimension of the output of a convolutional layer can be determined by deriving from the receptive field formula:

$$O = \frac{W - K + 2P}{S} + 1 \tag{3.1}$$

where O symbolize the output size (height/width), W dentoes the input size (height/width), K signifies the kernel size (height/width), P represents the padding on each side (total padding divided by 2 if it's uniform), and S is the stride.

Substituting the respective values of W, K, and P:

$$O = \frac{4 - 2 + 2 \cdot 0}{2} + 1 = 2$$

the output matrix is obtained as a 2x2 matrix with the values:

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

Q1.3. What benefits do CNNs have over regular fully connected networks One could particualrly benerfit from CNNs than from regular fully connected networks when the data can be characterized by spatial or temporal structures. Local connectivity and spatial dependencies are well-captured by CNNs, esepcailly when addressed with a layered approach, the architecture enables the network to efficiently learn the spatial and patterns hierarchical features by applying convolutional layers to small regions of the input data such that the complex patterns and relationships within the data are captured, analogy to filtering and windowed cross-correlation in digital signal processing. Additioanlly, serving as the primairy advantage of CNNs, this approach significantly reduces the number of parameters compared to fully connected networks, making CNNs more efficient. In addition to windowing captures, CNNs exhibit a parameter, or interconnected weights, sharing nature across different regions of the input, such weight-sharing mechanism enables the network to learn spatially invariant features, helping generalizing the learned features while reducing overfitting.

While CNNs are inherently translation-invariant due to the layered structure, allowing the network to recognize patterns regardless of their position in the input. Moreover, CNNs preserve the spatial structure of the input data through convolutional and pooling layers. The tranlsation-invalraitn property and the spatial features preserving nature make CNN ideal for tasks where the spatial arrangement of features matters, such as image recognition and segmentation, which spatial dependencies could not be captured in fine-details using regular fully-connected network.

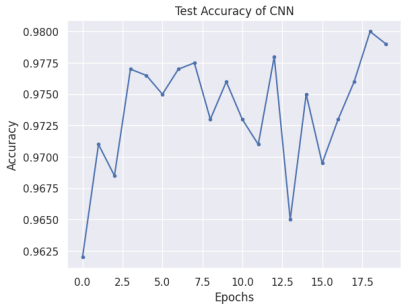


Figure 3.1: Initial CNN Architecture Baseline Test Accuracy

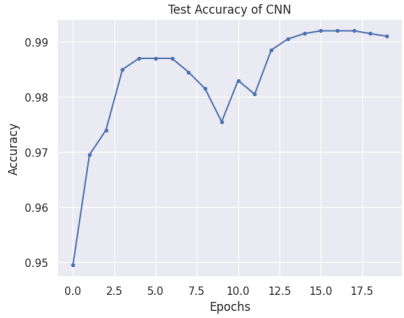


Figure 3.2: Improved CNN Architecture Test Accuracy

3.2.2 Q2. Experiment with CNN Architectures on MNIST: CNN architecture tuning

The initial baseline CNN architecture consisting of four layers with input channels set at 1, 16, 32, and 32, output channels set at 16, 32, 32, and 32, kernel sizes starting from 3 and reducing to 2, and padding of 1 achieved a test accuracy of approximately 0.98 (Figure 3.1).

Architectures of Convolutional Neural Network (CNN) including parameters such as the number of layers, types of layers (convolutional, pooling, fully connected), number of filters, and kernel sizes were experimented with. The adjustments allows for obtaining a more comprehensive understanding of how the model structure affects the network's ability to learn from the data.

In the experiment, various configurations were tested, including filter numbers of 16, 32, 64, 128, and 256, kernel sizes of 3 and 5, adding one layer of Leaky ReLU activation function, and one layer of batch normalization. Keeping the number of training epochs, batch size, and shuffle condition the same as the baseline, the test accuracy of the model with the best hyperparameters configuration was 0.991, albeit slightly better the architecture optimization through grid search still pose as a valid approach in optimizing the model structure, improving from 0.98 to 0.991.

It is empirically observed that the number of convolutional layers and number of filters are associated with the network's ability in learning more complex features. The deeper the network, the better the combine features into higher-order representations and performacne. However, overfitting or increased computational cost stand as possible adversarial effects. A different kernel sizes in the convolutional layers, is on the other hand, associated with the granularity of the features learned such that larger kernels capture more global features, whereas smaller kernels apprehend more local features.

Additionally, pooling layers such as MaxPooling incoporation post blocks of convolutional layers is observed reduce the spatial size of the representation, reducing the dimension and alleviating the inherent computation expensive process. Experiments with activation functions, such as ReLU, LeakyReLU, and Tanh, however, is observed to be applciation specific.

3.3 Section 3.3: Self-Attention and Transformers

3.3.1 Q1. Self-Attention Mechanism in NumPy and PyTorch Implementations

The self-attention mechanism relies on the correlation between queries, keys, and values, which are derived from the input data by projecting it through different weight matrices. If the input data X has dimensions $[n, d]$, where n is the number of tokens and d is the dimensionality of each token, the dimensions of the query (Q), key (K), and value (V) matrices after projection are $[n, d_k]$, $[n, d_k]$, and $[n, d_v]$, respectively. The attention scores are calculated by taking the dot product of the query matrix Wq with the key matrix Wk transpose, resulting in a matrix of dimensions $[n, n]$. These scores are then scaled (usually by $1/sqrt(d_k)$ addressing the exponential explosion of dimension) and multiplied by the value matrix Wv to produce the final attention output, which has dimensions $[n, d_v]$.

Projects, implemented using dot product, is anlaogy to operation of finding the correlation between matrix, or vectors, which resulting weighs vectors indicates how well. e.g. the corresponding keys and queries match. Queries (Q) are projections of the input data used to score the importance of each token relative to others. Keys (K) are used together with queries to compute the attention scores, determining the focus on other parts

of the input data for each token. Values (V) are projections of the input data aggregated according to the attention scores to produce the final output.

The attention mechanism, especially with multi-head attention, allows the the model to jointly attend to information from different representation subspaces at different positions of input sequence when producing each token in the output sequence, which output can be interpreted as the correlation between the input words, encoding the local and global relationship to focus-guiding weight matrices. This is crucial for tasks such as translation, where the relevance of input tokens can vary depending on the context.

3.3.2 Q2. Training Transformer on MNIST: Impact of Architecture Tuning

In the provided script, the Transformer model is trained on the MNIST dataset, and the architecture is experimented with by tuning the following hyperparameters:

- **dim**: The dimensionality of the model (default: 64).
- **depth**: The number of transformer blocks (default: 6).
- **heads**: The number of attention heads in the multi-head attention mechanism (default: 8).
- **mlp_dim**: The dimensionality of the feedforward network inside the transformer blocks (default: 128).

Model with the best hyperparameters configuration presents an average test loss: 0.0698 and an accuracy: 9850/10000(98.50%) in the 18^th training epochs. As the goal is to explore how changing the architecture size affects the model's performance on the MNIST dataset, configurations (indiated below) varying between the `dim`, `depth`, `heads`, and `mlp_dim` parameters of the `ViT` class are performed to evaluate its impact on the model's performance.

```
configurations = [
    {'dim': 64, 'depth': 6, 'heads': 8, 'mlp_dim': 128}, # Base configuration
    {'dim': 128, 'depth': 6, 'heads': 8, 'mlp_dim': 256}, # + dim, mlp
    {'dim': 32, 'depth': 6, 'heads': 8, 'mlp_dim': 64}, # - dim, mlp
    {'dim': 64, 'depth': 8, 'heads': 16, 'mlp_dim': 128}, # + depth, heads
    {'dim': 64, 'depth': 3, 'heads': 4, 'mlp_dim': 128}, # - depth, heads
    {'dim': 128, 'depth': 8, 'heads': 8, 'mlp_dim': 256}, # + dim, depth, mlp
    {'dim': 128, 'depth': 8, 'heads': 16, 'mlp_dim': 256}, # + dim, depth, heads, mlp
]
```

The performance of each configuration on the MNIST test set indicates noticeable differences between smaller and larger models. Smaller models with dimensions of 64 exhibit a steady increase in accuracy with each epoch, reaching up to 98.10% accuracy. Larger models with dimensions of 128, while initially showing slightly lower accuracy, ultimately achieve similar high accuracy levels. This suggests that both model sizes are capable of achieving high performance on the MNIST dataset, though larger models may require more epochs to converge.

Overfitting is a concern for larger models due to their higher capacity. The results indicate that larger models (e.g., dim: 128, depth: 6, heads: 8, mlp_dim: 256) maintain high accuracy and do not show significant signs of overfitting.

The loss curves during training show that larger models tend to converge slightly faster in the initial epochs but require fine-tuning over more epochs to stabilize. For instance, models with higher dimensions and depth (e.g., dim: 128, depth: 6) show rapid initial loss reduction but require more epochs to achieve optimal accuracy. Instability in training was not significantly observed across different configurations, indicating robust performance.

The computational cost is higher for larger models, as indicated by the longer training times and more complex architecture. Despite this, the performance gains in terms of accuracy justify the increased computational cost, especially when precision is critical. For instance, a model with dim: 128, depth: 6, heads: 8, mlp_dim: 256 reaches an accuracy of 98.48% after 20 epochs, exemplifying that the computational expense is offset by the accuracy benefits.

Chapter 4

Generative Models

4.1 Section 4.1: Energy-Based Models: Restricted Boltzmann Machines

4.1.1 Q1. Restricted Boltzmann machine (RBM): pseudo-likelihood?

The goal of RBM training is to maximize the log-likelihood of the training data, achieved by learning over the gradient of the log-likelihood, or logarithm of the joint distribution $P_{\text{model}}(v, h; \theta)$. The exact maximum log-likelihood, characterized by an energy function $E(v, h; \theta) = -v^T W h - b^T v - a^T h$ defining the joint configuration of visible units v and hidden units h and a partition function $Z(\theta) = \sum_v \sum_h \exp(-E(v, h; \theta))$ denoting the sum of the exponentiated negative energies over all possible configurations of the visible and hidden units, involves calculating the derivatives of the logarithm of the joint distribution $P_{\text{model}}(v, h; \theta) = \frac{1}{Z(\theta)} \exp(-E(v, h; \theta))$. This is interpreted as the expectation difference between the data-dependent expectation with respect to data distribution $P_{\text{data}}(h, v; \theta)$ and the model's expectation $P_{\text{model}}(v, h; \theta)$. However, this exact maximum likelihood learning is intractable for large models owing to the exponentially growing number of terms.

To circumvent this, the Contrastive Divergence (CD) algorithm with conditional distributions, or conditional probabilities, factorized as

$$P(h|v; \theta) = \prod_j p(h_j|v) \tag{4.1}$$

$$P(v|h; \theta) = \prod_i p(v_i|h) \tag{4.2}$$

using the sigmoid activation function, is used as an approximation. The update rule for the weight matrix W is given by

$$\Delta W = \alpha(E_{P_{\text{data}}}[vh^T] - E_{P_T}[vh^T]) \tag{4.3}$$

where P_T is the distribution obtained after running a Gibbs sampler for T steps. This alternative objective function, known as the pseudo-likelihood, is used to approximate the log-likelihood of the data, and is defined as

$$\log P(v_i|v_{-i}; \theta) = \sum_i \log P(v_i|v_{-i}; \theta) \tag{4.4}$$

where v_{-i} denotes the visible units excluding the i-th unit. The pseudo-likelihood training algorithm simplifies the computation of the log-likelihood, avoiding the intractable calculation of the partition function $Z(\theta)$, and focuses on the local dependencies between the visible units, rather than the global structure of the data. This allows for more efficient training, but at the cost of an approximation that might not capture all dependencies in the data as accurately as the true likelihood.

4.1.2 Q2. Evaluating the Impact of Component Count, Learning Rate, and Iteration Number on Performance

The effect of the number of hidden units $n_{components}$ is significant. Comparing Configurations 1 and 2 (Table 4.1), increasing the number of hidden units from 10 to 20 significantly boost the pseudo-likelihood, indicating that more hidden units allow for capturing more complex patterns in the data. Configuration 5 (Table 4.1), with 50 hidden units, shows even better performance, suggesting that a higher number of hidden units can lead to better model capacity and improved performance.

Configuration	n_components	learning_rate	n_iter
1	20	0.01	10
2	10	0.01	10
3	10	0.001	10
4	10	0.001	30
5	50	0.001	30

Table 4.1: Configuration parameters for model training.

A higher learning rate of 0.01 (Configuration 2) shows better performance compared to a lower learning rate of 0.001 (Configuration 3). The pseudo-likelihood decreases more significantly with a higher learning rate, indicating faster convergence with a limited number of iterations. However, a higher learning rate may also lead to overshooting the optimal parameters.

Regarding the number of iterations (n_{iter}), Configurations 3 and 4 (Table 4.1) demonstrate that increasing the number of iterations from 10 to 30 slightly enhance the pseudo-likelihood. This indicates that more iterations allow the model more time to converge, although the improvement might be diminishing. Configuration 5, with more hidden units, shows that the number of iterations has a significant impact, as seen in the continuous improvement in pseudo-likelihood even at 30 iterations.

The observations indicate that the number of hidden units serves as the dominant factor in improving model performance, followed by the learning rate and number of iterations. While a lower learning rate in general may suggest avoidance in overshooting the optimal parameters, it is essential that such configuration would require more training iterations allowing for optimal convergence. Without an increment in the number of hidden units, an increment in the number of iterations may not necessarily improve the model performance.

4.1.3 Q3. Impact of Varying Gibbs Sampling Steps

Gibbs sampling, a technique used in the Contrastive Divergence (CD) algorithm, involves training the model by sampling from the model distribution using Markov Chain Monte Carlo (MCMC) methods. This process iteratively updates the states of the visible and hidden units to draw samples from the joint distribution. The number of Gibbs sampling steps determines how many iterations the sampler runs to approximate the model distribution.

With 50 Gibbs sampling steps, the generated images are noisy, with digits being vaguely recognizable, indicating insufficient iterations to fully explore the state space. Increasing the steps to 100 significantly improves the quality of the generated images, reflecting a more comprehensive representation of the data distribution. Finally, with 200 Gibbs sampling steps, distinct digits are clearly generated, indicating that the sampler has learned a closer approximation to the true distribution.

4.1.4 Q4. Image Reconstruction Using RBM

The RBM attempts to fill in the missing parts of the data, characterized by variables `start_row_to_remove` and `end_row_to_remove`, based on the learned distribution from the training data, utilizing a specified number of Gibbs steps (`reconstruction_gibbs_steps`).

Initially, `end_row_to_remove` was set to `0` to empirically determine the optimal number of Gibbs steps for accurate reconstruction. It was found that `49` Gibbs steps were effective for reconstructing the missing parts accurately (Figure 4.1).

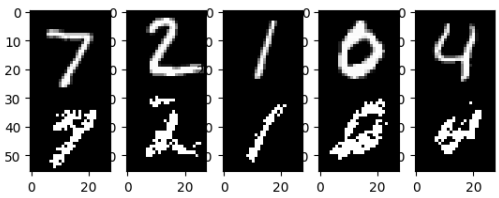


Figure 4.1: Reconstructed digits with RBM Gibbs steps of 49

Subsequently, `end_row_to_remove` was incremented stepwise to explore the RBM's ability to infer structure and patterns under a configuration of `100` hidden units, `0.01` learning rate, and `30` iterations. The model could accurately reconstruct the missing parts when up to `6` rows were removed. However, the reconstruction

quality degraded significantly beyond `7` rows and completely failed when `20` rows were removed, even with an increased `number of Gibbs steps` to `200`.

4.1.5 Q5. Impact of Row Removal Section on Network Image Reconstruction

The reconstruction results are less likely to be affected if the removed sections are less critical to the overall shape of the digit. For instance, removing four rows from the top of digits `7` or `9` still allows for fair reconstruction, as the lower parts significantly contribute to their shape. However, for digits such as `4`, `5`, or `6`, removing rows from the middle is often more detrimental due to the disruption of the digit's continuity, leading to poorer RBM inference.

4.2 Section 4.1: Energy-Based Models: Deep Boltzmann Machines

4.2.1 Q1. Analyzing Filters: Pre-trained Deep Boltzmann Machine on MNIST

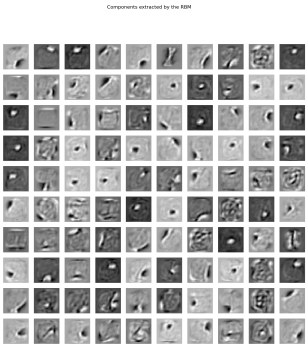


Figure 4.2: RBM Filters

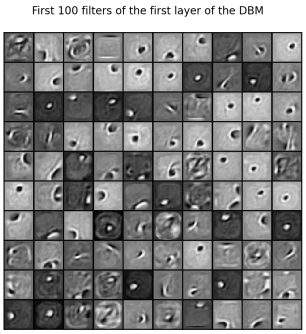


Figure 4.3: DBM First Layer Filters

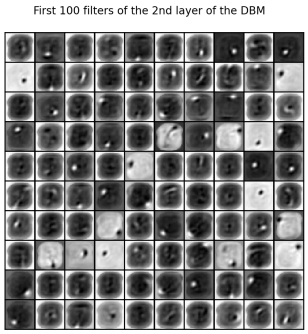


Figure 4.4: DBM Second Layer Filters

1. **Filters Extracted by the RBM and the 1st layer Filters Extracted by the DBM**: In grayscale, the filters in the DBM (Figure 4.3)exhibit features in general with higher contrast. While the intensity of each pixel corresponds to the weight value, darker shades represent negative weights, lighter shades represent positive weights, and mid-gray represents weights close to zero, the DBM filters, with comparatively more darker shades exhibited in the filter indicating high negative weights in conjunction with high-intensity features indicating high positive weights located in the filters, present a significant potential in gradient descent optimization for edge detection compared to the generally-mid-gray RBM filters(Figure 4.2). Filters in the first layer of a DBM also capture basic features but with more variation and detail compared to

2. **Filters Extracted by the 2nd Layer of the DBM**: Filters across different layers capture features at varying levels of abstraction owing to the hierarchical architecture of the model. Observed from the figures, the first layer consists of filters with a largest shade cluster lighter than the second layer filters. This indicates the function of the first layer filters is to capture localized edge- and corner-like features in the region, which distinct patterns can be directly interpreted as parts of the input images, resembling stroke-like or edge-detecting features. The second layer filters (Figure 4.4) are more abstract and complex, capturing higher-order patterns in a global extent by integrating multiple low-level features. The second layer filters exhibit more intricate and abstract patterns, with darker shades, or negative weights covering the specified round-shaped region, presumably where the digit's distinctive outline is located, with contrastive light shades covering the remaining corner of the filters. This suggests more complex broader area pattern learning and inter-local-features relationship learning are conducted at the second layer, essential for understanding the broader context and finer details within the data. Some filters in the second layer indicate a notable high-intensity circular small region in varied positions across filters, which could be interpreted as the prominent feature detection of the digit's distinctive outline.

4.2.2 Q2. Image Sampling from DBM vs RBM

The images from the DBM are of higher quality compared to those from the RBM due to the DBM's deeper architecture. This results in more distinguishable digit outlines and clearer prominent features. The DBM's multi-layer structure facilitates the learning of hierarchical data representations. The first layer captures low-level features such as edges and textures, while subsequent layers capture higher-level abstractions and complex patterns. This multi-layer approach allows the DBM to generate more realistic and detailed images compared to the single-layer RBM, which primarily captures local dependencies.

4.3 Section 4.2: Generator and Discriminator in the Ring: Generative Adversarial Networks (GANs)

4.3.1 Q1. Explain the different losses and results of the GAN framework.

In the framework of Generative Adversarial Networks (GANs), there exist two neural networks: the generator and the discriminator. These networks are trained simultaneously in a two-player minimax game, where the generator aims to produce realistic data samples and the discriminator aims to distinguish between real and generated data. The GAN's training objective is formalized as:

$$G^* = \arg\min_G \max_D v(G, D) \tag{4.5}$$

The value function $v(\theta^G, \theta^D)$ is expressed as:

$$v(\theta^G, \theta^D) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{4.6}$$

In this traditional minimax game, the generator's objective is to minimize:

$$J^{(G)}(G) = \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{4.7}$$

While the discriminator aims to maximize the probability of correctly classifying real and fake data, defined as:

$$J^{(D)}(D) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{4.8}$$

In practice, adjustments are made to mitigate vanishing gradients. The generator's objective is modified to the non-saturating GAN objective:

$$J^{(G)}(G) = -\mathbb{E}_{z\sim p_z}[\log D(G(z))] \tag{4.9}$$

With the corresponding loss function for the generator

$$L_G = -\left(\mathbb{E}_{z\sim p_z}[\log D(G(z))]\right) \tag{4.10}$$

Here, the generator tries to maximize $\log D(G(z))$, which is equivalent to minimizing $-\log D(G(z))$, such that $D(G(z))$ should be as large as possible, interpreted as the discriminator will classify the generated data $G(z)$ as real with high probability. This modification stabilizes training by ensuring the generator receives stronger gradients even when the discriminator performs well.

The loss function for the discriminator:

$$L_D = -\left(\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]\right) \tag{4.11}$$

Breaking down the discriminator's loss according to Goodfellow et al. (2014):
The loss for the real samples:

$$L_{D,\text{real}} = -\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] \tag{4.12}$$

The loss for the fake samples:

$$L_{D,\text{fake}} = -\mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{4.13}$$

The total discriminator loss is:

$$L_D = L_{D,\text{real}} + L_{D,\text{fake}} \tag{4.14}$$

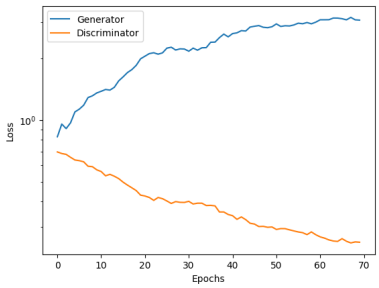


Figure 4.5: GAN loss curve

In the context of the discriminator's objectives, both $L_{D,\text{real}}$ and $L_{D,\text{fake}}$ should be minimized and approach zero as training progresses. From the perspective of the discriminator, $D(G(z))$ should be as close to zero as possible, indicating that the generated data is classified as fake. Conversely, $D(x)$ should be as close to one as possible, indicating that the real data is accurately classified as real. During training, the generator's loss should increase over time, while the discriminator's loss should decrease, reflecting the generator's improving ability to deceive the discriminator, making it more challenging to differentiate between real and fake data, as illustrated in Figure 4.5. This indicates stable training without significant oscillations or divergence, a common issue in GAN training. After sufficient training, the generator's distribution p_G becomes indistinguishable from the real data distribution p_{data} such that $p_{\text{G}} = p_{\text{data}}$. At equilibrium, the discriminator cannot differentiate between real and fake samples, resulting in $D(x) = \frac{1}{2}$ for any sample x. This indicates that the generator has learned to generate realistic data samples.

4.3.2 Q2. Effects of a Superior Discriminator in GANs

If the discriminator performs significantly better than the generator, several issues can arise, affecting the training dynamics and the quality of the generated data. The discriminator's performance can be considered too strong when it can easily distinguish between real and fake data with high confidence. This imbalance in performance indicates a classification accuracy with high confidence of $D(x) \approx 1$ for real data and $D(G(z)) \approx 0$ for generated data, leading to diminishing gradients for the generator. Suppose a batch of m generator updates by descending its stochastic gradient:

$$\nabla_{\theta^G} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D(G(z^{(i)}))\right) \tag{4.15}$$

If $D(G(z)) \approx 0$, such vanishing gradients for the generator yield the gradients degrade, leading to the inability of the generator to learn and improve, adversely enforcing constant poor-quality samples generation.

4.3.3 Q3. Discuss and illustrate the convergence and stability of GANs.

Configured with a latent dimension of 20, a batch size of 512, and a learning rate of 1×10^{-3}, the empirical results detail the average losses for the generator and discriminator over 70 epochs. Key observations include the theoretical illustration of GAN convergence.

Near convergence, the generator's distribution p_G approximates the real data distribution p_{data}, and the discriminator D achieves partial accuracy in distinguishing real from fake data. The discriminator is optimized to differentiate real and fake samples, converging to:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} \tag{4.16}$$

The generator updates based on the discriminator's feedback, producing samples $G(z)$ that are increasingly realistic. At equilibrium, $p_G \approx p_{\text{data}}$, and the discriminator's output for any sample x is:

$$D(x) = \frac{1}{2} \tag{4.17}$$

In the empirical training process, the provided GAN implementation includes key elements such as network architecture and training procedures. The generator comprises fully connected layers with Leaky ReLU and

Tanh activations, while the discriminator consists of fully connected layers with Leaky ReLU and Dropout, followed by a sigmoid activation. During training, the discriminator updates are performed using the gradient:

$$\nabla_{\theta^D} \frac{1}{m} \sum_{i=1}^{m} \left[\log D(x^{(i)}) + \log(1 - D(G(z^{(i)}))) \right] \tag{4.18}$$

The generator updates are carried out with the gradient:

$$\nabla_{\theta^G} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))) \tag{4.19}$$

Empirical Results

Stage	Epoch	Average Generator Loss	Average Discriminator Loss
Initial	1	0.829	0.698
Initial	2	0.956	0.686
Mid	10	1.36	0.571
Mid	20	1.99	0.429
Later	30	2.22	0.396
Later	50	2.86	0.3
Later	70	3.08	0.253

The convergence analysis indicates that both the generator and discriminator losses move towards equilibrium, aligning with the theoretical expectation $p_G \approx p_{\text{data}}$. The discriminator's loss decreases, showing improved classification capability, while the generator's loss increases but stabilizes, indicating enhanced realism in generated samples. The smooth progression of losses without significant oscillations or divergence, as shown in Figure 4.5, suggests a stable training process. Adjustments to learning rates and regularization techniques contribute to this stability.

4.3.4 Q4. Explore the latent space and discuss.

Three aspects of the latent space are explored: latent dimension, interpolation in latent space (traversing latent space), and the sampling and visualizing of multiple latent vectors.

Latent dimension was first explored empirically by comparing `latent_dim = 10` and `latent_dim = 20` with both `num_epochs = 70`, results tabulated below:

Latent Dimension	Epoch 1 Generator Loss	Epoch 1 Discriminator Loss	Epoch 25 Generator Loss	Epoch 25 Discriminator Loss	Epoch 50 Generator Loss	Epoch 50 Discriminator Loss
10	0.818	0.705	1.76	0.463	2.46	0.354
20	0.829	0.698	2.13	0.413	2.86	0.3

This reveals that an increment in latent dimension from `10` to `20` results in a higher generator loss, indicating that the generator is producing more realistic samples. The discriminator loss also decreases, suggesting that the discriminator is finding it more challenging to differentiate between real and fake data, which is a positive sign of the generator's improvement.

Interpolation in latent space, or discussed in the lecture as Traversing the latent space, was conducted by linearly interpolating between two latent vectors with parameter λ. The generated images are illustrated in Fig. 4.7, showing a smooth transition between numbers `5` and `3`. The interpolation indicates a semantically coherent and realistic morphing, or transition capability, of the generator of GAN in between latent vectors in the latent space.

Sampling multiple latent vectors, or discussed in the lecture as Sampling from the latent space, was conducted by randomly sampling multiple latent vectors and accordingly generating corresponding fake images. The generated images are illustrated in Fig. 4.6, showing the ability of the generator to produce relatively diverse and higher-quality samples and demonstrating zero memory dependency on the training set for new image generation. However, the latent code inference limitation of the GAN model, constrained by focusing on the models of the real-data distribution, is inevitable if compared to Variational Autoencoders (VAEs) or plain Autoencoders, which have an encoder.

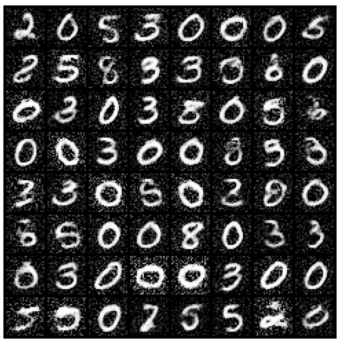


Figure 4.6: Latent vectors Sampling (GANs)

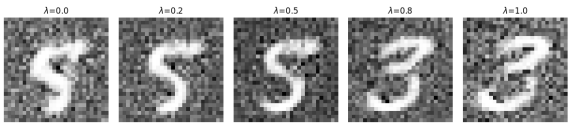


Figure 4.7: Latent vectors interpolation (GANs)

4.3.5 Q5. Try the CNN-based backbone and discuss.

CNN-based GANs, or Deep convolutional GAN (DCGAN), compared to the fully connected GANs, as discussed in the lecture, demonstrate a more realistic and diverse image generation capability well-suited to the application of Large-scale Scene Understanding. As the DCGAN generator leverages a hierarchical structure of convolutional layers, it captures the distribution projected small spatial extent convolutional representations with feature maps, recognizing local patterns, textures, and dependencies. Exploiting the aforementioned properties of convolutional networks, the extracted feature maps are then upsampled, deconvoluted, expanding images from features to generate high-quality images. Conversely, the discriminator performs an inverse operation, downsampling the input images to extract features and classifying the images as real or fake.

4.3.6 Q6. Pros and Cons of GANs vs. Other Generative Models like Autoencoders and Diffusion Models

When comparing GANs with other prominent generative models such as autoencoders and diffusion models, several factors including conceptual aspects, quality of results, and training considerations must be taken into account.

GANs offer several advantages. The adversarial architecture guides the generator implicitly through feedback from the discriminator, facilitating realistic data generation without explicitly modeling the data distribution. This approach often results in high-quality outputs, with GANs producing highly detailed and high-fidelity images that frequently surpass those of other generative models. Additionally, GANs have inspired a diverse range of architectures, such as DCGAN, Conditional GAN, Auxiliary Classifier GAN, InfoGAN, CycleGAN, and StyleGAN, leveraging the spatial pattern and texture learning advantages of convolutional layers.

However, GANs also exhibit notable disadvantages. Training instability is a significant issue, with GANs often suffering from vanishing gradients, requiring careful tuning of hyperparameters to stabilize the training process. Furthermore, training GANs is resource-intensive, demanding substantial computational power and large amounts of data to effectively learn the distribution. Another challenge is the lack of underlying data distribution information. Since GANs do not model the data density explicitly, evaluating the likelihood of the generated samples becomes difficult, complicating the assessment of the model's performance.

Autoencoders, particularly Variational Autoencoders (VAEs), provide a structured probabilistic framework. VAEs enable stable training processes due to the regularization of the latent space, which ensures more diversified sample generation and better latent space interpolation and interpretability. This structured latent space, enhanced by the KL divergence regularization term, offers a well-defined and interpretable feature, making VAEs suitable for various applications.

Diffusion models are another class of generative models that excel in producing high-fidelity outputs. The iterative nature of diffusion models allows for the generation of highly detailed images, often surpassing the performance of GANs. However, the computational expense and time-consuming nature of the iterative steps in diffusion models are significant drawbacks, making them less efficient in terms of training and inference.

A summary comparison of these generative models is provided in the table below:

This comparison highlights the strengths and weaknesses of each generative model, offering insights into their applicability based on specific requirements such as efficiency, sample quality, latent space behavior, and likelihood estimation.

Model	Efficient	Sample Quality	Coverage	Well-behaved Latent Space	Disentangled Latent Space	Efficient Likelihood
GANs	✓	✓	×	✓	?	n/a
VAEs	✓	×	?	✓	?	×
Flows	✓	×	?	✓	?	✓
Diffusion	×	✓	?	×	×	×

Table 4.2: Comparison of Generative Models

4.4 Section 4.3: An Auto-Encoder with a Touch: Variational Auto-Encoders (VAEs)

4.4.1 Q1. Practical Metrics for Model Optimization

Variational Autoencoders (VAEs), instead of maximizing the log-likelihood which is computationally intractable for complex models, capitalize on parameterizing the variational distributions (latent variables) and approximating them using Jensen's inequality:

$$\ln p(x) = \ln \int p(x|z)p(z)dz \geq \mathbb{E}_{z\sim q_\phi(z|x)}\left[\ln \frac{p(x|z)p(z)}{q_\phi(z|x)}\right] \tag{4.20}$$

Considering an amortized variational posterior $q_\phi(z|x)$ for each x, the VAE model maximizes the Evidence Lower Bound (ELBO) instead of the log-likelihood, which is defined as:

$$\ln p(x) \geq \text{ELBO} = \mathbb{E}_{z\sim q_\phi(z|x)}[\ln p(x|z)] - \mathbb{E}_{z\sim q_\phi(z|x)}(\ln q_\phi(z|x) - \ln p(z)) \tag{4.21}$$

Here, $\mathbb{E}_{z\sim q_\phi(z|x)}[\ln p(x|z)]$ is the expected log-likelihood of the data given the latent variables, corresponding to the reconstruction error term measuring the model's ability to reconstruct the input data.

$\mathbb{E}_{z\sim q_\phi(z|x)}(\ln q_\phi(z|x) - \ln p(z))$ is the Kullback-Leibler (KL) divergence $\text{KL}(q_\phi(z|x)\|p(z))$ between the approximate posterior $q_\phi(z|x)$ and the prior $p(z)$, acting as a regularization term ensuring the approximate posterior is close to the prior.

The lower bound of the log-likelihood is referred to as Evidence Lower Bound (ELBO), which is maximized during training. The ELBO balances the reconstruction error and the regularization term, facilitating an accurate representation of the latent space while stimulating a generalized model construction.

In implementation, the $q_\phi(z|x)$ can be reparameterized by sampling from a common distribution such as a Gaussian distribution, characterized by μ_ϕ mean, σ_ϕ standard deviation, and σ_ϕ^2 variance, enabling the backpropagation of gradients in stochastic gradient descent training for ELBO maximization.

4.4.2 Q2. Comparing Stacked Autoencoder and Current Model

Both VAEs and stacked autoencoders use an encoder-decoder architecture where the encoder maps input data to a lower-dimensional latent space while the decoder reconstructs the data from this latent space, and both models aim to minimize the reconstruction error matching the original input. However, there are key differences in the objective functions and the latent space regularization between VAEs and stacked autoencoders.

In VAEs, the encoder maps input x to parameters $\mu(x)$ and $\sigma(x)^2$ of a Gaussian distribution $q_\phi(z|x)$. The latent variable z is then sampled from this distribution during training, introducing randomness and regularization. A regularization term is added to the loss function to ensure the learned latent distribution $q_\phi(z|x)$ is close to the prior distribution $p(z)$ (usually a standard Gaussian), measured using the Kullback-Leibler (KL) divergence: $\text{KL}(q_\phi(z|x)\|p(z))$. The VAE objective function aims to maximize the Evidence Lower Bound (ELBO), balancing the reconstruction accuracy and the regularization of the latent space distribution:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)] - \text{KL}(q_\phi(z|x)\|p(z)) \tag{4.22}$$

The first term, $\mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)]$, ensures the reconstructed output $\hat{x}$ is similar to the input x. For continuous data, this is typically implemented using Mean Squared Error (MSE) loss: $\mathbb{E}_{z\sim q_\phi(z|x)}[-\|x - \hat{x}\|^2]$. For binary data, binary cross-entropy loss $\mathbb{E}_{z\sim q_\phi(z|x)}[-(x \log \hat{x} + (1 - x) \log(1 - \hat{x}))]$ is used. In contrast, stacked autoencoders employ a deterministic mapping from input x to a latent space z, without enforcing

Feature	VAE	Stacked Autoencoder
Architecture	Encoder-decoder	Encoder-decoder
Latent Space Regularization	Probabilistic framework with latent space defined by a distribution (e.g., Gaussian). Encoder outputs mean and variance, with sampling from this distribution. Regularization via KL divergence.	Deterministic mapping from input to latent space without enforced distribution over latent variables.
Objective Function	Maximizes the Evidence Lower Bound (ELBO): $\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z\|x)}[\ln p_\theta(x\|z)] - \mathrm{KL}(q_\phi(z\|x)\|p(z))$	Minimizes reconstruction error (e.g., Mean Squared Error).
Reconstruction Error Metric	Part of ELBO, measured as expected log-likelihood of data given latent variables: $\mathbb{E}_{z\sim q_\phi(z\|x)}[\ln p(x\|z)]$. Uses MSE for continuous data or binary cross-entropy for binary data.	Typically uses Mean Squared Error (MSE) loss for continuous data.
Mathematical Formula	ELBO: $\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z\|x)}[\ln p_\theta(x\|z)] - \mathrm{KL}(q_\phi(z\|x)\|p(z))$	Reconstruction Error: $\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$

Table 4.3: Comparison of VAEs and Stacked Autoencoders

a distribution over the latent variables. In the case of images $x \in \{0, 1, ..., 255\}^D$, one cannot use a normal distribution, rather, a categorical distribution:

$$p_\theta(x|z) = \text{Categorical}(x|\theta(z)) \tag{4.23}$$

using a Neural Network (NN) for $\theta(z) = \text{softmax}(\text{NN}(z))$.

The objective is to minimize the reconstruction error, ensuring the output $\hat{x}$ is as close as possible to the input x:

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2 \tag{4.24}$$

The key difference lies in the probabilistic framework and regularization of the latent space in VAEs, which is absent in stacked autoencoders. The probabilistic nature of VAEs allows for decomposing the error into reconstruction error, $\mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)] = \sum_{n=1}^{N}\{\ln \text{Categorical}(x_n|\theta(z_{\phi,n}))\}$, and KL divergence, $\mathrm{KL}(q_\phi(z|x)\|p(z)) = \frac{1}{2}\sum_{n=1}^{N}\{1 + \ln(\sigma^2_{\phi,n}) - \mu^2_{\phi,n} - \sigma^2_{\phi,n}\}$, providing a more structured and interpretable latent space. In contrast, stacked autoencoders focus solely on minimizing the reconstruction error without probabilistic constraints, leading to a more deterministic latent space.

4.4.3 Q3. Exploring the Latent Space

With a configuration of `batch_size = 3000`, `latent_dim = 600`, `middle_dim = 300`, `learning_rate = 1e-3`, and `max_epochs = 100`, the empirical results of the latent space explorations samples of the VAE model are illustrated in Figure 4.8, whereas the interpolations in the latent space are depicted in Figure 4.9.

Figure 4.8: Latent vectors Sampling (VAEs)

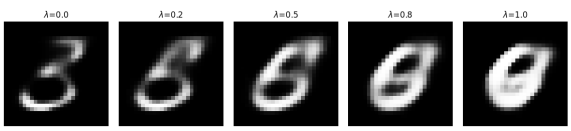


Figure 4.9: Latent vectors interpolation (VAEs)

Fig. 4.8 shows the latent space exploration samples of the VAE model, which are more diverse compared to those generated by GANs, as seen in Fig. 4.6 with the same MNIST dataset. This diversity captures a more holistic distribution of the latent space due to the probabilistic nature of the VAE model. However, the fidelity and level of detail in the VAE-generated digits are notably less intricate compared to those generated

by GANs, as VAEs focus on input data reconstruction rather than discriminator-guided new data sample generation as in GANs.

Fig. 4.9 demonstrates a strong coherence with high-level variations transforming between digits `3`, `5`, `6`, and almost `0`. This variance indicates a well-represented latent space where the VAE model can generate continuously changing data with more levels of variance than GANs, maintaining a semantically meaningful and interpretable change between latent vectors in the latent space.

4.4.4 Q4. Comparing Generation Mechanisms: GANs vs Current Model

The generation mechanisms of VAEs and GANs differ fundamentally in the architecture and objective functions. VAEs utilize a probabilistic framework to model the data distribution. The encoder maps the input x to a latent space z, which is defined by a Gaussian distribution with parameters mean μ and variance σ^2. The latent variable z is sampled from this distribution, and the decoder reconstructs the data $\hat{x}$ from z. The objective function of VAEs aims to maximize the Evidence Lower Bound (ELBO), comprising a reconstruction term and a Kullback-Leibler (KL) divergence term, given by:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\ln p_\theta(x|z)] - \mathrm{KL}(q_\phi(z|x)\|p(z)) \tag{4.25}$$

In contrast, GANs operate on an adversarial framework involving two neural networks: a generator G and a discriminator D. The generator maps a random noise vector z to the data space, producing fake samples, while the discriminator attempts to distinguish between real and fake samples. The training process is a minimax game where the generator tries to fool the discriminator, and the discriminator aims to correctly classify real versus fake samples. The objective functions for GANs are twofold: one for the generator, given by:

$$J^{(G)}(G) = -\mathbb{E}_{z\sim p_z}[\log D(G(z))] \tag{4.26}$$

and one for the discriminator, expressed as:

$$J^{(D)}(D) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{4.27}$$

The comparison indicates advantages and disadvatnges imposed by respective framework and objective functions of VAEs and GANs, which is summarized in Table 4.4.

Aspect	**VAEs**	**GANs**
Training Stability	Generally more stable due to probabilistic framework. The loss function is well-defined and gradients are usually smooth.	Training can be unstable due to the adversarial nature. It often requires careful tuning of hyperparameters and may suffer from mode collapse.
Latent Space Structure	Provides a well-defined and interpretable latent space, thanks to the regularization term (KL divergence). This leads to a smoother and more continuous latent space.	The latent space may not be as well-structured or interpretable. There is no explicit regularization enforcing a distribution over the latent space.
Sample Quality	Generated samples might be of lower quality compared to GANs because the decoder needs to model the entire data distribution.	GANs often produce high-quality, sharp images because the generator is directly trained to fool the discriminator, which enforces realism in the samples.
Mode Coverage	VAEs tend to cover the entire data distribution, generating diverse samples, but might be blurry.	GANs might suffer from mode collapse, where the generator produces a limited variety of samples, focusing on modes that can easily fool the discriminator.
Reconstruction Ability	Good reconstruction ability due to the explicit reconstruction term in the loss function.	GANs do not focus on reconstruction; they focus on generating realistic samples that can fool the discriminator.
Computational Efficiency	Training is generally computationally efficient and straightforward due to the absence of an adversarial component.	Training can be computationally intensive due to the need to optimize two networks simultaneously and the potential instability.

Table 4.4: Comparison of VAEs and GANs

Use of ChatGPT (or any other AI writing assistance tool)

Form to be completed

Student name: **Chieh Fei (Jeffee) Hsiung** .....

Student number: **R0819932** .....

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: Aritificial Neural Networks and Deep Learning .....

Course number: .....

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

Title Master thesis: .....

Promotor: .....

Please indicate with "X":

O **I did not use** ChatGPT or any other AI writing assistance tool.

X I did use AI Writing Assistance. In this case **specify which one** (e.g. ChatGPT/GPT4/...):

ChatGPT.....

Please indicate with "X" (possibly multiple times) in which way you were using it:

X Assistance purely with the language of the paper

- *Code of conduct*: This use is similar to using a spelling checker

X As a search engine to learn on a particular topic

- *Code of conduct*: This use is similar to e.g. a google search or checking Wikipedia. Be aware that the output of Chatbot evolves and may change over time.

O **For literature search**

- *Code of conduct*: This use is comparable to e.g. a google scholar search. However, be aware that some AI writing assistance 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.

X For short-form input assistance

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

O **To let generate programming code**

- *Code of conduct*: Correctly mention the use of ChatGPT (or other AI writing assistance tool) and cite it. You can also ask ChatGPT how to cite it.

O **To let 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 let generate blocks of text**

- *Code of conduct*: Inserting blocks of text without quotes from ChatGPT (or other AI writing assistance tool) to 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. In case it is really needed to insert a block of text from ChatGPT (or other AI writing assistance tool), mention it as a citation by using quotes. But this should be kept to an absolute minimum.

O **Other**

- *Code of conduct*: Contact the professor of the course or the promotor 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 other AI tool: ….

Further important guidelines and remarks

- ChatGPT cannot be used related **to data or subjects under NDA agreement.**
- ChatGPT cannot be used related **to sensitive or personal data due to privacy issues**.
- **Take a scientific and critical attitude** when interacting with ChatGPT (or other AI writing assistance tool)and interpreting its output. Don't become emotionally connected to AI tools.
- As a student you are responsible to comply with Article 84 of the exam regulations: your report or thesis should reflect your own knowledge. Be aware that plagiarism rules also apply to the use of ChatGPT or any other AI 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."
- **ChatGPT suggestion about citation**: "Citing and referencing ChatGPT output is essential to maintain academic integrity and avoid plagiarism. Here are some guidelines on how to correctly cite and reference ChatGPT in your Master's thesis: 1. Citing ChatGPT: Whenever you use a direct quote or paraphrase from ChatGPT, you should include an in-text citation that indicates the source. For example: (ChatGPT, 2023). 2. Referencing ChatGPT: In the reference list at the end of your thesis, you should include a full citation for ChatGPT. This should include the title of the AI language model, the year it was published or trained, the name of the institution or organization that developed it, and the URL or DOI (if available). For example: OpenAI. (2021). GPT-3 Language Model. https://openai.com/blog/gpt-3-apps/ 3. Describing the use of ChatGPT: You may also want to describe how you used ChatGPT in your research methodology section. This could include details on how you accessed ChatGPT, the specific parameters you used, and any other relevant information related to your use of the AI language model. Remember, it is important to adhere to your institution's specific guidelines for citing and referencing sources in your Master's thesis. If you are unsure about how to correctly cite and reference ChatGPT or any other source, consult with your thesis advisor or a librarian for guidance."

Additional reading

ACL 2023 Policy on AI Writing Assistance: https://2023.aclweb.org/blog/ACL-2023-policy/

KU Leuven guidelines on citing and referencing Generative AI tools, and other information:

https://www.kuleuven.be/english/education/student/educational-tools/generative-artificial-intelligence

Dit formulier werd opgesteld voor studenten in de Master of Artificial intelligence. Ze bevat een code of conduct, die we bij universiteitsbrede communicatie rond onderwijs verder wensen te hanteren.

Deze code of conduct schept een kader voor academiejaar 2023-2024, aanpassingen kunnen gebeuren in functie van nieuwe evoluties.