TÉCNICO LISBOA

Stochastic Undirected Neural Networks

Ricardo Manuel Madeira Fraga Fernandes Simões

Thesis to obtain the Master of Science Degree in

Data Science and Engineering

Supervisors: Prof. André Filipe Torres Martins
Prof. Mário Alexandre Teles de Figueiredo

Examination Committee

Chairperson: Prof. Bruno Emanuel Da Graça Martins
Supervisor: Prof. André Filipe Torres Martins
Member of the Committee: Prof. Vlad Niculae

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I wish to extend my heartfelt appreciation to my supervisors: Prof. André Martins and Prof. Mário Figueiredo. Their comprehensive guidance, discussions, reviews, and support over the past months have been invaluable. Collaborating and learning with such exceptional professors has been a great honour.

I also owe a lot to my parents, Fernando and Maria, for always supporting me in this journey. This work is dedicated to them. I also thank my grandparents, siblings, and family for their continuous encouragement.

Lastly, I want to thank my friends. The last five years had its ups and downs, and I couldn't have gotten through it without them. They've been there for me every step of the way, making it memorable.

Abstract

Recent developments in the fields of machine learning (ML) and deep learning (DL) have led to the discovery of new models capable of outperforming older models in various tasks, such as classification, object detection, and even sentiment analysis. This work presents a proof of concept for a type of neural network called stochastic undirected neural network (SUNN), which demonstrates the ability to perform multiple tasks, such as classifying an image or generating an image conditioned on its class, due to its property of not having a specific direction for the flow of internal information within the network. Additionally, this model also exhibits generative properties and can represent uncertainty due to the incorporation of stochastic nodes in the network. Both properties are explored simultaneously on a dataset commonly used as a benchmark, revealing promising results for specific tasks, when compared to some reference models.

The focus of this work is on classification and prototype generation tasks. To achieve this, we use SUNNs with fully connected layers and convolutional layers, which are compared to non-stochastic and undirected neural networks, as well as other well-known neural network models for similar applications. The architectures of stochastic undirected neural network models were optimized for these tasks, aiming to achieve the best possible results.

Keywords

Neural Networks, Deep Learning, Undirected, Stochasticity, Classification, Prototype Generation

Resumo

Desenvolvimentos recentes nas áreas de aprendizagem automática e aprendizagem profunda têm conduzido à descoberta de novos modelos capazes de superar modelos mais antigos em diversas tarefas, como classificação, deteção de objetos ou até análise de sentimento. Este trabalho, apresenta uma *Proof of Concept* para um tipo de rede neural apelidada de stochastic undirected neural network (SUNN), que demonstra a capacidade de executar multiplas tarefas, como classificação de imagens ou geração de imagens condicionadas a uma classe, devido à sua propriedade de não possuir uma direção específica para o fluxo de informação interna na rede. Adicionalmente, este modelo também exibe propriedades generativas e é capaz de representar incerteza devido à incorporação de nós estocásticos na rede. Ambas as propriedades são exploradas simultaneamente em um dataset comumente utilizado como referência, revelando resultados promissores para tarefas específicas, em comparação com alguns modelos de referência.

O foco deste trabalho está nas tarefas de classificação e geração de protótipos. Para isso, utilizamos SUNNs com camadas totalmente conectadas e camadas convolucionais, que são comparadas a redes neuronais não-estocásticas e não-direcionadas, e a outras conhecidas redes neuronais para aplicações semelhantes. As arquiteturas dos modelos stochastic undirected neural network foram otimizadas para essas tarefas, visando alcançar os melhores resultados possíveis.

Palavras Chave

Redes Neuronais, Aprendizagem Profunda, não-Direcionalidade, Estocasticidade, Classificação, Geração de Protótipos

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AU	area under
BNN	bayesian neural network
CNN	convolutional neural network
DL	deep learning
EBM	energy based model
ELBO	evidence lower bound
FFNN	feed-forward neural network
FG	factor graph
FPR	false positive rate
HN	Hopfield network
ID	in domain
JSD	Jensen–Shannon divergence
KL	Kullback-Leibler
MC	Monte-Carlo
MI	mutual information
ML	machine learning
NN	neural network
PM	perturb-and-map
PR	precision-recall
OOD	out of domain
BM	Boltzmann machine
ROC	receiver operating characteristic

SGD	stochastic gradient descent
SUNN	stochastic undirected neural network
TPR	true positive rate
UNN	undirected neural network
VAE	variational auto encoder
VI	variational inference

1

Introduction

Contents

1.1 Motivation

In recent times, the subjects of machine learning (ML) and deep learning (DL) have witnessed an array of remarkable advancements and pivotal breakthroughs, largely catalyzed by the innovative utilization of neural network (NN) architectures (see Chapter 5 in [3] and Chapters 1, 5 and 6 in [4]). These architectures have demonstrated their power across a diverse spectrum of tasks, encompassing various domains. However, many models adopt a monolithic approach, mapping inputs to outputs via a predetermined sequence of calculations. The challenges inherent in this paradigm were highlighted in the insightful work seen in [1], prompting the conception of the undirected neural network (UNN) model as a potential solution.

This conceptual innovation has the ability to encompass and unify well-established NN-based frameworks. The convolutional neural network (CNN) (see Chapter 9 in [4]) for tasks like image classification and visualization, or the biaffine model for dependency parsing tasks are just two examples of the many possible applications of this concept (see [5] and [6]). Nevertheless, the UNN model, while commendably tackling the aforementioned concerns, remains poised for further refinement in terms of introducing **generative properties** and establishing a mechanism for **quantifying uncertainty**. These are two distinct but pivotal challenges, that continue to captivate researchers within the dynamic domain of ML, stimulating a relentless pursuit of novel solutions.

Therefore, within the confines of this work, we introduce the new concept of stochastic undirected neural network (SUNN), an evolved version of the UNN framework, thoughtfully designed to possess the essential attributes for surmounting the dual challenges outlined earlier. Subsequent sections delve into the theoretical fundamentals of both the UNN and SUNN models. It is imperative to bear in mind that both models empower computations within a NN to unfold in a predefined manner, facilitating the flow of information across layers, unrestricted by direction and frequency. In fact, enhanced by the stochastic element intrinsic to SUNNs, each iteration introduces an element of randomness into the network's dynamics. This stochastic infusion equips the model to confront the two aforementioned predicaments, setting the stage for an innovative approach to address these challenges.

1.2 Applications

It is not hard to think about scenarios where it is important to measure uncertainty and scenarios where generating data is also valuable. A good example of the importance of uncertainty quantification can be found in Section 1.1 in [7], wherein a scenario unfolds of a passenger that commits to a generous tip for the taxi driver if the journey culminates within a 25-minute span. In such a situation, a navigation system that proffers a choice between two routes based on absolute time projection pales in utility compared to a probabilistic one. The latter one, rooted in probabilistic inference, produces results based

on a distribution of arrival times. This dynamic approach not only furnishes a refined decision-making foundation but also enables the calculation of the probability associated with securing the previously mentioned tip.

The field of data generation also holds immense potential, spanning a spectrum that ranges from enhancing training data through augmentation to crafting synthetic samples for exploration and analysis. This unveils a novel era characterized by data-driven creativity and profound insights. Notably, certain domains exhibit a rare disparity between different classes. For instance, in financial transactions, non-fraudulent occurrences are more prevalent than fraudulent ones, as elaborated in [8]. Imbalanced datasets can introduce bias towards the most frequent class; in these situations, it struggles to correctly classify examples of the minority class. This is where data generation comes into play, enabling the creation of new data samples to achieve a more balanced distribution of samples per class. This approach helps prevent the machine learning model from becoming overly biased towards predicting the majority class.

While the landscape of potential applications sprawls extensively, we consciously opt to narrow our purview for the purposes of this MSc thesis. Our focus gravitates towards a specific dataset of paramount significance, one that has achieved widespread adoption across various ML tasks, the *MNIST* [9], which serves as an established benchmark. We use it for our research, in the realms of uncertainty measurement and data generation tasks.

1.3 Contributions

The present work makes contributions to both the field of uncertainty quantification and the subject of data generation. In particular, we introduce a novel model that exhibits both commonalities and distinctions when compared to classical approaches within these domains. This innovation not only expands the horizons of potential hybrid solutions but also underscores its applicability across both areas. Our contributions can be synthesized as:

- An overview of classical models capable of data generation and other models suitable for uncertainty quantification.
- An introduction of our novel model, accompanied by a direct comparison to the aforementioned approaches.
- An application of the proposed model on the *MNIST* dataset, demonstrating its effectiveness through data synthesis and uncertainty assessment in predictions.
- An open-sourced implementation in *Pytorch*, offering the research community access to our experiment code for future investigations, available in [10].

- A vast amount of ideas of future work to develop SUNN models in terms of applications and training alternatives.

1.4 Organization of the Document

This dissertation is organized in the following way:

Chapter 2 provides some background knowledge, which is essentially for understanding the fundamentals of the contributions presented in this dissertation. It contains a review of factor graphs (FGs) and graphical models, some energy based models (EBMs), uncertainty estimation methods, and the reparametrization trick.

Chapter 3 proposes a stochastic and undirected NN-based model, titled SUNN, with the potential to measure uncertainty in its predictions and generate new data. Comparison with other models as well as the training function is presented.

Chapter 4 presents the experiments and their results, with emphasis on the model's performance when compared with other approaches, using relevant metrics.

Chapter 5 presents the main conclusions of this dissertation and some suggestions for future work and experiments.

2

Literature Review and Background

Contents

This chapter provides fundamental background for the proposed model, in order to understand its structure, algorithm, applicability, and contributions that can be seen in Chapters 3 and 4. This chapter starts with Section 2.1, with an overview of the importance of FGs for representing UNN and SUNN architectures. After this, the idea behind EBMs, is introduced in Section 2.2, since the proposed approach is also energy-based. The section starts with an overview of two specific and well-known EBMs, Hopfield networks (HNs), in sub-Section 2.2.1, and Boltzmann machines (BMs), in sub-Section 2.2.2, where the latter one has generative properties. Still, in this section an overview of the recently proposed UNN model is shown in sub-Section 2.2.3, followed by an outline of the perturb-and-map (PM), a popular sampling technique, in sub-Section 2.2.4. In Section 2.3 three frameworks that are capable of measuring uncertainty in ML models are shown, more specifically in sub-Sections 2.3.1, 2.3.2 and 2.3.3. They will be compared with the SUNN model in the uncertainty estimation front. Finally in Section 2.4 the classic reparametrization trick approach is outlined, which has ties to the way stochasticity is introduced in the SUNN model.

2.1 Factor Graphs and Graphical Models

Factor Graphs (FGs) are a type of bipartite graphs that were first introduced in [11]. This type of graphical model has many connections with other types of models like Tanner graphs, Markov random fields and Bayesian networks. They all serve a common purpose, which is to model probabilistic relationships between variables. Although there are similarities among them, there are also key differences in their graphical structure and the types of relationships they encode.

Tanner graphs are considered the predecessor of FGs, widely applied in the coding theory field. They are characterized by variable nodes and check nodes which represent the individual bits of the codeword bits and parity-check equations, respectively (see [12] and Section 11.6 in [13]). As for Bayesian networks, they are a type of directed graphical model, where nodes in the graph represent variables, and directed edges represent direct dependencies or causal relationships between variables using a directed acyclic graph. They allow the representation of the joint distribution of a graph as a product of conditional distributions. They encode conditional independence using the concept of d-separation and inference is performed using techniques like, for example, belief propagation (see Sections 8.1 and 8.2 in [14]). Markov random fields are very similar to Bayesian networks, but the graph is undirected. Using the concept of a *clique* and energy-functions, they define the joint probability distribution of the graph (see Section 8.3 in [14]). Finally, FGs are very similar to Tanner graphs, but in FGs, the variable nodes represent variables in the probabilistic model, and factor nodes represent the factors that capture the relationships between variables (and they represent functions or potentials). The edges in FGs connect variables to the factors they depend on. The general purpose is to represent factorized joint probability

distributions in probabilistic graphical models (like Bayesian networks and Markov random fields) and opposed to Tanner graphs that focus on error-correcting codes and their decoding. The FGs distinguish themselves from Bayesian networks and Markov random fields due to some additional nodes (generally represented by squares) for each factor in the joint distribution. These factors allow the representation of the joint probability distribution of the graph as a product of functions defined over them (see Section 8.4 in [14]). For our proposed model, we leverage FGs, which can be useful for the explicit representation of modular components in NN architectures that we further present.

We are now in conditions of defining a FG. We use the same notation across this work and define a FG as $\mathcal{G}$, constituted by a set of variable nodes $V = \{X_1, ..., X_N\}$ and a set of factor nodes F, where each factor node $f_s \in F$ is linked to a subset of variable nodes. Each variable node $X \in V$ is associated with a representation vector $x \in \mathbb{R}^{d_X}$. Let $X_s \subseteq V$ be a subset of nodes that are linked through a particular factor f_s. We can write the joint distribution, $p(x_1, ..., x_N)$, as a product of factors, f_s:

$$p(x_1, ..., x_N) = \prod_s f_s(x_s). \tag{2.1}$$

A minimalist example can be seen in Figure 2.1. In this example we have three nodes, therefore $V = \{X_1, X_2, X_3\}$ with, representation vectors x_1, x_2 and x_3, respectively. The nodes X_1 and X_2 are connected via f_a and f_b, X_2 and X_3 via f_c and we have a single factor f_d connected to node X_3. The joint distribution is given by

$$p(x_1, x_2, x_3) = f_a(x_1, x_2) f_b(x_1, x_2) f_c(x_2, x_3) f_d(x_3). \tag{2.2}$$

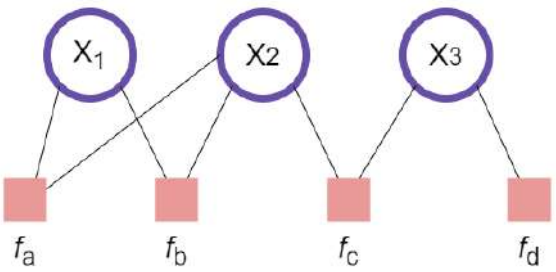


Figure 2.1: Example of a FG with three nodes and four factors.

This factorization of the joint probability distribution is also done in Bayesian networks and Markov random fields, although done differently using the techniques mentioned above. Additionally, they go a step further and do a connection between the joint probability distribution and an energy function, which

is something commonly done in energy-based models. For example, in Markov random fields we have the factorization

$$p\left(x_1, ..., x_N\right) = \frac{1}{Z} \prod_C \psi_C(x_C). \tag{2.3}$$

In the previous equation $\psi_C(x_C)$ is a potential function defined for a particular maximal *clique* C and the set of variables in that *clique* is given by x_C. A *clique* is defined as a subset of nodes such that there exists a link between all pairs of nodes in that subset. If it is not possible to include any other nodes from the graph in the set without it ceasing to be a *clique*, then we have a maximal *clique*. The term Z is a normalization term, given by

$$Z = \sum_x \prod_C \psi_C(x_C). \tag{2.4}$$

This potential function is then linked with an energy function, $E(x_C)$ defined over the variables of the *clique*, by

$$\psi_C(x_C) = \exp(-E(x_C)). \tag{2.5}$$

The exponential representation seen in Equation (2.5) is commonly called as the *Boltzmann Distribution*, which has roots in statistical mechanics (see Section 9.4 in [15]). It is commonly used in the statistics and ML fields and used as the sampling distribution of stochastic NNs such as the restricted BM [16]. FGs do not directly employ the concept of energy potentials nor the negative exponential of energy as done in Markov random fields representations. They solely represent factorized joint probability distributions by explicitly modelling the factors as defined in Equation (2.1) and, therefore, they are more general than Markov random fields. In [1], FGs are used to express the modularity of many NN architecture models and for each FG there is an underlying energy function which is essential for the propagation of information throughout the network. However, it is not connected to a probability distribution defined over the nodes in the FG, as it will be seen more in-depth in sub-Section 2.2.3. In this work, a similar approach is going to be used for the SUNN model.

2.2 Energy-Based Models

Energy-based models (EBMs) are a group of probabilistic models used for many different machine learning tasks such as classification, memory recalling, optimization, generative modelling, etc. For

these types of models, we have an energy function, and its minimization process allows the network to capture dependencies between variables, as seen in sub-Section 2.2.1 and Section 2.2.3 for HNs and UNNs, respectively. In BMs, which we can see in sub-Section 2.2.2, this is also the case. However, with their stochastic component, the global energy can increase sometimes in order to escape noisy low-energy configurations.

2.2.1 Hopfield Network

The Hopfield network (HN) [17] was one of the first proposed energy-based models being similar to a simple neural network, but with its own special features. In general, the network can be seen as a group of N nodes, written as $V = \{x_1, ..., x_1\}$, and it is made up of two-way connections between every pair of nodes, forming a feedback loop. These connections are characterized by a weights' matrix of size $N \times N$, $W = \{w_{ij}\}_{i,j=1}^{N}$, where w_{ij} is the weight going from the i^{th} to the j^{th} node. It has equal weights in both directions ($w_{ij} = w_{ji}$), and it does not connect back to itself ($w_{ii} = 0$). Each neuron's output is called "activity", and the overall energy formula is

$$E(V) = -\frac{1}{2}\sum_{i,j} x_i x_j w_{ij}. \tag{2.6}$$

In the context of the binary HN, the variables are denoted as $x_i = \pm 1, \forall i \in \{1, ..., N\}$. However, beyond the binary network, alternative variations such as the continuous HN exist, where $x_i \in \mathbb{R}, \forall i \in \{1, ..., N\}$. The selection between these variants hinges on the inherent nature of the problem and the desired characteristics of the network. In [3], Chapter 42, the author demonstrates the utility of the binary HN for memory retrieval and the application of the continuous HN for tackling optimization problems. For the purpose of our analysis, we concentrate on the binary version.

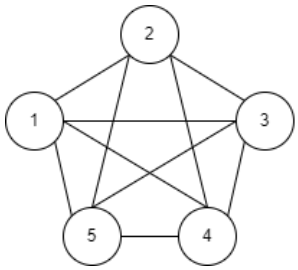
The values assigned to each node within the network undergo updates guided by specific rules. These updates can occur either synchronously or asynchronously. In the case of synchronous updates, all neurons compute their activations in unison. The update expression of x_i, is given by a_i, defined by:

$$a_i = \sum_{j=1}^{n} w_{ij} x_j. \tag{2.7}$$

The updates are then done simultaneously using a discrete update rule, as stated by

$$x_i = \begin{cases} 1, & \text{if } a_i \geq 0 \\ -1, & \text{if } a_i \leq 0. \end{cases} \tag{2.8}$$

If the updates are asynchronous, the activations and updates are made in a sequential way, and the order in which the updates are made is defined by the user. Under the condition mentioned above, the energy function defined in Equation (2.6) is a Lyapunov function, which decreases its value under the evolution of the system (see Section 42.4 in [3]). Therefore, memories that the net stores internally are actually minimum energy values where the network converges to, for any initial condition (which can be a corrupted memory we wish to recover). However, note that the HN can store points of local minimum energy which correspond to spurious states – patterns that were not explicitly stored. Below, in Figure 2.2, we present a common way to represent a HN, alongside its weights' matrix when $N = 5$.



$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{bmatrix} = \begin{bmatrix} 0 & w_{21} & w_{31} & w_{41} & w_{51} \\ w_{21} & 0 & w_{32} & w_{42} & w_{52} \\ w_{31} & w_{32} & 0 & w_{43} & w_{53} \\ w_{41} & w_{42} & w_{43} & 0 & w_{54} \\ w_{51} & w_{52} & w_{53} & w_{54} & 0 \end{bmatrix}$$

Figure 2.2: Example of the graphical representation of an HN (left) and its weights matrix (right).

All the connections seen in Figure 2.2 represent the weights of the network, which have to be specified. A common way to specify the weights is to use the Hebb learning rule. It is based on the remark that "neurons which fire together, wire together". This quote suggests that the more frequently two neurons are active, the larger their associative strength becomes: this is what the Hebb rule does. When we present the K target memories $\{x_1^t, ..., x_K^t\}$, weights are adjusted based on the correlation between the activity of the neurons. If two neurons have the same sign more often than not during this phase, their connection weight is increased, and the opposite is also valid, as seen in:

$$w_{ij} = \frac{1}{K} \sum_{t=1}^{K} x_i^t x_j^t. \tag{2.9}$$

This method contrasts with backpropagation (see Section 6.5 in [4]), which is the conventional way of learning the values of the weights in many other NN models, something that usually requires multiple thousands of examples and computational resources. As for the Hebb rule, each weight requires only one calculation, making it much computationally simpler. The most interesting fact regarding HN is that regardless of the initial state we input to the network, this energy function is guaranteed to decrease and converge to a minimal energy value, under some particular conditions such as weights' symmetry and asynchronous updates (see Section. 42.4 in [3]). A synthesized algorithm of this model can be seen below.

Algorithm 1 Binary Hopfield Network - training and inference

1: Set the weights using your set of target memories T: Use, for example, the Hebb rule in Equation (2.9);
2: Input a sample: Set the values of the network neurons according to the values of the input;
3: Update the weights, synchronously or asynchronously using Equation (2.7), and Equation (2.8);
4: Repeat step 3 until convergence or stopping criterion:

2.2.2 Boltzmann Machines

The Boltzmann machine (BM) [16], also known as stochastic HN is another energy-based NN that shares many similarities to a HN, being one of the first EBMs with stochasticity. While BMs were theoretically defined, they are rarely used in practice due to their computational complexity (see Chap 27.7 of [18]). Instead, a restricted version of the network is used, named restricted BM. This particular model is motivated by the idea that it can be a NN that models a particular type of distribution function, which is the Boltzmann distribution function presented in Equation (2.5). More specifically, the objective of a BM is to find the optimal parameters that minimize the difference between the model's estimated distribution and the true distribution of the data.

The architecture of this model is defined by nodes - visible (v) and hidden (h), which are connected to each other, with some restrictions. The number of visible nodes in a BM is given by the dimensionality of the input data. For example, if we are working with tabular data, each column of the dataset represents a visible node. On the other hand, if we are working with images of size 28×28 pixels, then we would treat every pixel as a feature, leading to 784 visible nodes in the network. The number of hidden nodes in a BM is usually a hyperparameter that needs to be specified and tuned. Naturally, this hyperparameter affects the network's ability to learn complex patterns in the data. For previous works on finding the appropriate number of hidden nodes see, for example, [19] and [20]. The energy function of this model depends on the nature of the data being studied. For example, if we consider a binary (or Bernoulli) restricted BM with D visible and K hidden binary nodes, define $\mathcal{V} = \{0,1\}^D$ as the set of visible vectors and $\mathcal{H} = \{0,1\}^K$ as the set of hidden vectors, the energy expression is given by:

$$E(v,h) = -\sum_{i=1}^{D}\sum_{j=1}^{K} w_{ij} v_i h_j - \sum_{i}^{D} b_i^V v_i - \sum_{j}^{K} b_j^H h_j. \tag{2.10}$$

In the previous equation, w_{ij} is the weight that connects the i–th visible unit and the j–th hidden unit, b_i^V and b_j^H are biases associated with visible unit i and hidden unit j, respectively. Other variants for categorical and real-valued data can be seen in Table 27.2 in [18]. Restricted BMs lack directed connections and possess an energy function. Thus, they can be represented as a Markov random field, as discussed in Section 2.1. Their joint distribution function is defined using potential functions, as shown in Equation (2.3). Having the network architecture defined, it is necessary to fine-tune the parameters for a

specific dataset. During training, the weights and biases are adjusted to reduce the energy of observed samples, bringing their energies closer to that of training data. This process allows the network to capture dependencies and patterns in the data, enabling it to, for example, generate new data samples or perform dimensionality reduction. This is done using a specific algorithm, like the constrastive divergence [21]. The fact that the network has some restrictions in the connections simplifies the modelling and learning processes, making it more efficient than a fully connected BM. In inference time, we initiate this process by beginning with a basic pixelized image, and the goal is to gradually refine this image over multiple Gibbs sampling [22] steps to make it look more like actual handwritten numbers. At each step, the algorithm updates the pixels, guided by the statistical relationships learned from the provided training data. Gibbs sampling is used to obtain approximated samples from complex probability distributions and is widely used in statistics and ML applications. This method is described in Algorithm 2.

Algorithm 2 Inference in restricted Boltzmann Machine using Gibbs sampling.

Initialization: Start with initial visible unit states $v^0 = \{v_i^0\}_{i=1}^D$ and initial hidden unit states $h^0 = \{h_i^0\}_{i=1}^K$
repeat:
for $j = 1$ **to** K **do**
    $P(h_j^i = 1|v^{i-1}) = \sigma\left(b_j^H + \sum_{n=1}^D v_n^{i-1} w_{ij}\right)$
    Sample hidden state h_j^i:
    $h_j^i = \begin{cases} 1 & \text{with probability } P(h_j = 1|v^{i-1}) \\ 0 & \text{otherwise} \end{cases}$
end for
Sampling Visible States:
for $n = 1$ **to** D **do**
    $P(v_n^i = 1|h^i) = \sigma\left(b_n^V + \sum_{j=1}^K h_j^i w_{ij}\right)$
    Sample visible state v_n^i:
    $v_n^i = \begin{cases} 1 & \text{with probability } P(v_n^i = 1|h^i) \\ 0 & \text{otherwise} \end{cases}$
end for

The restricted BM is, like the HN, an energy minimization procedure. It is supposed that the distribution of the visible and hidden nodes of the network can be modelled by an expression closely similar to the one employed in Markov random fields, defined in Equation (2.5), which is:

$$\begin{aligned} p(v,h) &= \frac{1}{Z} exp(-E(v,h)), \\ \text{where } Z &= \sum_{v' \in \mathcal{V}} \sum_{h' \in \mathcal{H}} exp(-E(v',h')). \end{aligned} \tag{2.11}$$

The formula shows that states with low energy are linked to a higher probability of occurrence, while high-energy states are linked to low probability values. Now, the connections' weights in BMs determine the energy associated with different configurations. The learning process involves adjusting these

weights to minimize the energy of observed data and increase the energy of unobserved or unwanted patterns. It is common to consider an approach of minimizing the negative log-likelihood for a set of N training examples $\{v_1, ..., v_N\}$, as:

$$l(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log p(v_i|\theta). \tag{2.12}$$

If we define W as the set of weights between nodes, B^V as the set of bias terms of the visible nodes, and B^H as the set of bias terms of the hidden nodes, it can be shown (see sub-Section 27.7.2 in [18] and [23]) that the derivation of this function with respect to the set of model parameters $\theta = \{W, B^V, B^H\}$ yields:

$$\nabla_\theta l(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta F(v_n) - \sum_{v' \in \mathcal{V}} P(v')\nabla_\theta F(v'), \tag{2.13}$$

where

$$F(v) = -v^T b^V - \sum_{i=1}^{K} \text{soft}_+(W_{i.}v + b_i^H), \tag{2.14}$$

$$\nabla_W F(v) = \mathbb{E}(h|v)v^T = -\widehat{h}(v)v^T, \tag{2.15}$$

$$\nabla_{B^H} F(v) = \mathbb{E}(h|v) = -\widehat{h}(v), \tag{2.16}$$

$$\nabla_{B^V} F(v) = -v. \tag{2.17}$$

In the previous equations, $\widehat{h}(v) = \sigma(Wv + B^H)$, where $\sigma(x) = \frac{1}{1+e^{-x}}$, and $\text{soft}_+(x) = \ln(1 + e^x)$. The updates in the model's parameters can be done by coupling this procedure with a gradient-based approach. The intuition, as mentioned in [23], is that the first term in Equation (2.13) increases the probability of examples provided, while the second term decreases the probability of examples generated by the model. In inference time, the restricted BM can start on a given initial state and do successive Gibbs sampling updates, tending to energy minima points learned during training time. These minima states do represent the patterns or features that are consistent with the training data distribution. A small experience with Bernoulli restricted BMs can be done in the *MNIST* dataset, where we explore the process of generating images that resemble handwritten digits. The two examples, seen in Figure 2.3, show that the network can actually generate meaningful representations. However, there are certain disadvantages and limitations associated with this approach to image generation.

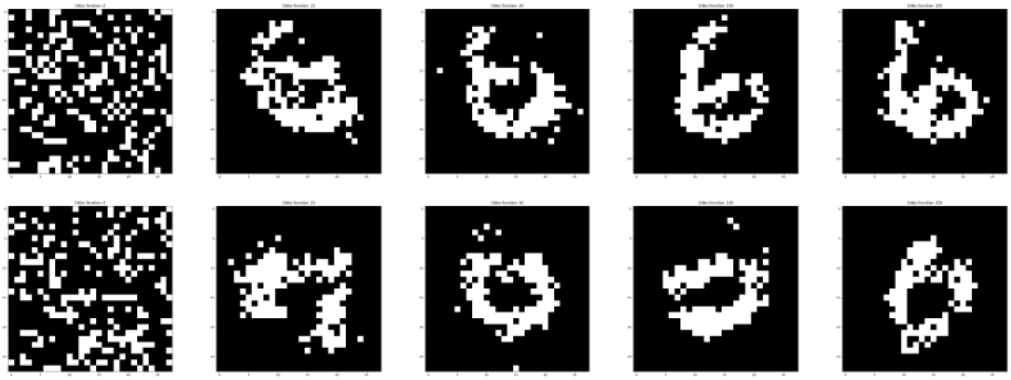


Figure 2.3: Top row: Evolution of images from the first experiment with Bernoulli restricted BM model (left to right: Initial image, 25th iteration, 50th iteration, 100th iteration, 200th iteration). Bottom row: Evolution of images from the second experiment with Bernoulli restricted BM model (left to right: Initial image, 25th iteration, 50th iteration, 100th iteration, 200th iteration).

We can see that the quality of the generated images is sensitive to the initial pixelized image used as a starting point. Poor initialization might lead to slow convergence or less diverse outcomes. We also verified that sometimes Gibbs sampling can be slow to converge to high-quality samples, and the bottom image generated is proof of that. In other words, a substantial number of iterations may be required to generate images that are visually appealing and diverse. Performing a large number of Gibbs sampling iterations for high-resolution images can be computationally expensive, making the image generation process time-consuming. The previous results also show that the first example had converged to a stable state around the 30^{th} iteration, where the estimate of the likelihood (the pseudo-likelihood, whose implementation can be seen in Equation (18.20) in [4]) achieved a maximum value, as seen in Figure 2.4.

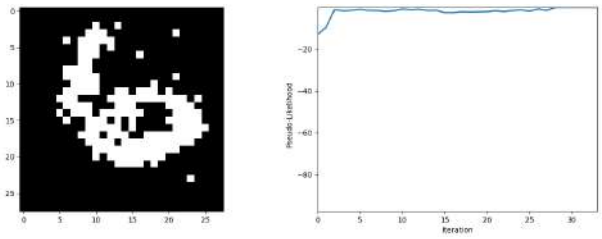


Figure 2.4: Pseudo-likelihood evolution with successive Gibbs sampling iterations, of the first example in Figure 2.3.

The image in the previous figure shows that the initial states have a very low (pseudo) likelihood since they do not resemble a learned pattern because the initial point is random noise. However, with successive iterations the network evolves to a pattern it learns during training. This implementation was done using *sklearn* [24] functions.

2.2.3 Undirected Neural Networks

The main building block of this thesis is the undirected neural network (UNN) model, proposed in [1]. In this small overview over the UNN, the same notation as in the original work is used. This notation will also be used in Chapter 3, when we introduce the SUNN model. The original work of the UNN model combines NNs with FGs. On one hand, we have NNs, who are powerful estimators used in many different tasks and often achieve state-of-the-art results. On the other hand, we have FGs, which emphasize the problem's modularity [25], and allows the user to visualize complex joint probability distributions as a product of local conditional probability distributions, which captures the relationship between a subset of variables in the graph, commonly denoted as factors [11]. Their combination offers a flexible framework, where the order of the computations involving the factors of the network can be arbitrary – hence, the term undirected. A particular choice for a FG allows the UNN to represent a specific NN architecture. Just like it was shown in [1], UNNs can subsume many well-known NN architectures like feed-forward, recurrent, self-attention networks, auto-encoders, etc). In Figure 2.5 we see some examples of FGs, $\mathcal{G} = \{V, F\}$ where the node variables, V, are represented by circles and the factors, F, are represented by squares.

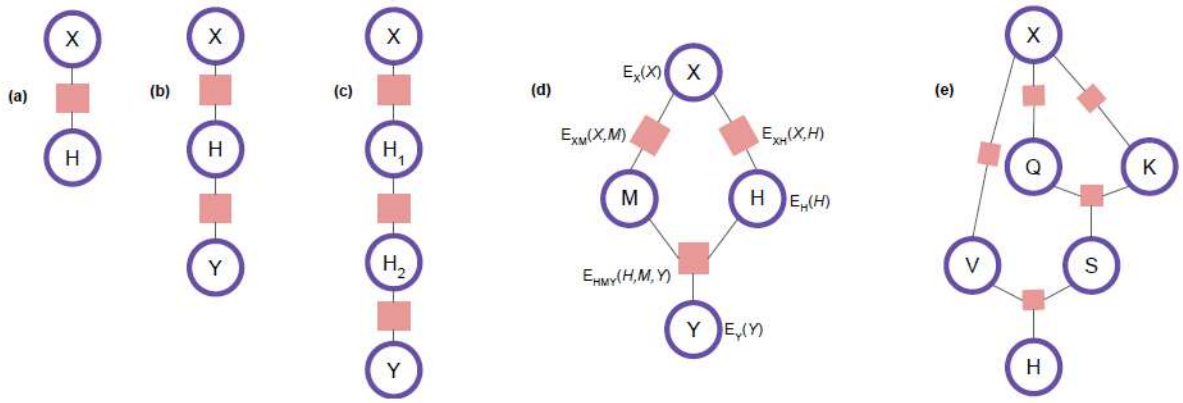


Figure 2.5: FGs for different undirected networks. From left to right: zero (**a**), one (**b**), and two (**c**) intermediate layers, for the biaffine dependency parser model (**d**), for the self-attention mechanism (**e**). Energy labels were omitted for brevity with the exception of (**d**). Adapted from [1].

For example, due to the arbitrary computation order, a UNN that subsumes a feed-forward neural network (FFNN) allows it to work both as a classifier and a prototype generator. For UNNs, outputs are not obtained by evaluating a composition of functions in a specific order, but rather by minimizing the overall energy of the system, via updating its factor terms in a specific order. This large difference between UNNs and other common regular NN is that now we do updates on the energy terms, and this characterizes the flux of information through the network, which can happen in an arbitrary direction. Since the network can do two tasks, a dual loss function can be minimized with standard deep learning optimizers, like stochastic gradient descent (SGD) or *Adam*. Regarding the energy function of UNN

models, it is composed of single and high-order energy terms. The single energy terms are defined for each node and the high-order energy terms refer to the factors, where two or more nodes may be linked. The overall energy is then:

$$E(x_1, ..., x_N) = \sum_i E_{X_i}(x_i) + \sum_f E_f(x_f). \tag{2.18}$$

In the previous equation, we are considering N arbitrary nodes, $V = \{X_1, ..., X_N\}$, therefore the sum is defined over all these N nodes plus all the factors that may exist. Every node X_i is associated with a representation vector $x_i \in \mathbb{R}^{d_{X_i}}$ The authors restrict the energy equations to the form:

$$\begin{aligned} E_{X_i}(x_i) &= \Psi_{X_i}(x_i) - \langle b_{X_i}, x_i \rangle \\ E_f(x_f) &= -\langle W_f, \bigotimes_{X_j \in f} x_f \rangle. \end{aligned} \tag{2.19}$$

In the previous equation, b_{X_i} refers to the bias term associated with the node X_i and W_f to the weights associated with a specific factor. The term Ψ_{X_i} refers to a strictly convex regularizer. It happens that with these conditions, a learning rate free coordinate descent algorithm guarantees the global energy to decrease at every iteration [26], updating each representation x_i in a predefined order, leaving the remaining representations fixed. Moreover, the update equations result in a chain of affine transformations followed by non-linear activations that yield the traditional computations that happen in regular NN-based models. In general, there can be many nodes linked to a factor in the network. In this work, we will focus solely on pairwise factors $f = \{X_i, X_j\}$, since this will be the scenario in the examples we will address. For the more general expression see Appendix A in [1]. For the case where the bias term and the representation vector are vectors, we have the update expression:

$$\begin{aligned} (x_i)_* &= \arg\min_{x_i} E_{X_i}(x_i) + \sum_{f=\{X_i, X_j\} \in F(X_i)} E_f(x_f) \\ &= \arg\min_{x_i} \Psi_{X_i}(x_i) \underbrace{-b_i^T x_i - \sum_{f=\{X_i, X_j\} \in F(X_i)} \langle W_f, \bigotimes_{j \in f} x_j \rangle}_{-z_i^T x_i}. \end{aligned} \tag{2.20}$$

The Fenchel conjugate [27] of a function $\Psi : \mathbb{R}^d \to \mathbb{R}$ is given by $\Psi^*(t) = sup_{x \in \mathbb{R}^d} \langle x, t \rangle - \Psi(x)$. Due to the convexity of Ψ,Ψ^* is differentiable and $(\nabla \Psi^*)(t)$ is the unique maximizer $\arg\max_{x \in \mathbb{R}^d} \langle x, t \rangle - \Psi(x)$. Then

$$\begin{aligned} (x_i)_* &= (\nabla \Psi^*_{X_i})(z_i), \\ z_i &= \sum_{f=\{X_i, X_j\} \in F(X_i)} \rho_i(W_f) x_j + b_i. \end{aligned} \tag{2.21}$$

In the former equation, ρ_i is either the identity or the transpose operator. In essence, the update of the node X_i is an activation function $(\nabla \Psi^*_{X_i})$, generally non-linear, applied over an affine transformation

that involves the neighbours of the node X_i, and the bias term linked to the node X_i. Below, Table 2.1 shows some common options of regularizers and the corresponding activation functions.

Table 2.1: Regularizers $\Psi(h)$, with corresponding activation functions $\nabla\Psi^*(t)$, where $\phi(t) = t\log(t)$.

$\Psi(h)$	$(\nabla\Psi^*)(t)$
$\frac{1}{2}\|\|h\|\|^2$	t
$\frac{1}{2}\|\|h\|\|^2 + 1_{\mathbb{R}_+}(h)$	$relu(t)$
$\sum_j (\phi(h_j) + \phi(1-h_j)) + 1_{[0,1]^d}(h)$	$sigmoid(t)$
$\sum_j (\phi(\frac{1+h_j}{2}) + \phi(\frac{1-h_j}{2})) + 1_{[-1,1]^d}(h)$	$tanh(t)$
$-\mathcal{H}(h) + 1_{\Delta}(h)$	$softmax(t)$

Below, we present the equations for a specific UNN model, which is the undirected FFNN with only one hidden layer. This model can be graphically represented using the second (b) FG seen in Figure 2.5. This FG is composed of a set of nodes $V = \{X, H, Y\}$, that represent the input intermediate and one output layer, respectively. The adjacent layers are connected via factors, therefore $F = \{XH, HY\}$. If provided with training data from the *MNIST* dataset, this setup can correspond to a NN that works both as a digit classifier and a prototype generator, as seen below in Figure 2.6.

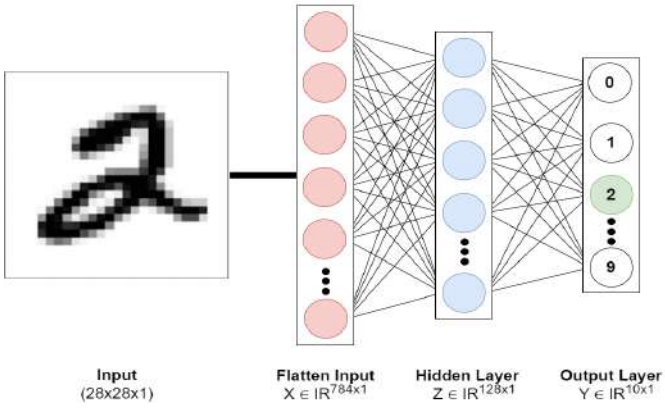


Figure 2.6: Example of an undirected FFNN with one hidden layer.

In this example, our total energy function is given by:

$$\begin{aligned} E(x,y,z) &= \sum_i E_{X_i}(x_i) + \sum_f E_f(x_f) \\ &= E_X(x) + E_H(h) + E_Y(y) + E_{XH}(x,h) + E_{HY}(h,y) \\ &= -\langle b_X, x\rangle + \Psi_X(x) - \langle b_H, h\rangle + \Psi_H(h) \\ &\quad - \langle b_Y, y\rangle + \Psi_Y(y) - \langle h, Wx\rangle - \langle y, Vh\rangle. \end{aligned} \tag{2.22}$$

The updates of each node variable are given below in Equation (2.23):

$$
\begin{aligned}
X_{k+1} &= (\bigtriangledown \Psi_X^*)(W^T H_{k+1} + b_X), \\
H_{k+1} &= (\bigtriangledown \Psi_{H_k}^*)(Wx + V^T y_k + b_H), \\
Y_{k+1} &= (\bigtriangledown \Psi_Y^*)(V H_{k+1} + b_Y).
\end{aligned} \tag{2.23}
$$

Note that the term b_X is usually not considered, as bias terms are generally not used in the input layer. Upon analysis, it becomes apparent that the update functions bear striking resemblance to computations conducted in a usual FFNN. Note that the node H has two neighbour nodes, X and Y, and therefore they influence its update function. Nodes X and Y only have one neighbour, which is H for both, and therefore they are influenced only by this node. For a task where an input digit X is given, and we want to obtain a class prediction of that digit, we can now update H and Y terms as many times as desired, and pick the Y representation as the final prediction of the given input. For the opposite task where a label Y is provided, we can generate a digit representation X by updating the nodes X and H as many times as desired.

2.2.4 Perturb-and-MAP

In sub-Section 2.2.2, we discussed the incorporation of stochasticity into EBMs through methods like Gibbs sampling and contrastive divergence. Yet, there are alternative techniques for this purpose, one of which is the Perturb-and-MAP (PM) method, as detailed in [28]. This method offers a different avenue for introducing randomness into EBMs. Designed as an alternative to conventional sampling techniques such as Markov-chain Monte Carlo (MCMC)—which can sometimes be computationally intensive or problematic—the PM is employed for approximate inference in Gibbs Markov random fields. These graphs hold significance in numerous applications, primarily because of their capacity to represent local interactions and dependencies between variables. For example, in image segmentation tasks, a pixel's label is frequently determined by the labels of its adjacent pixels. In this context, our emphasis is on the relationship between the PM and EBMs, the mechanism of noise introduction, and the rule for parameter learning.

At the core of the PM method, there is an energy function defined over discrete labels. This energy function quantifies a "cost" associated with each possible label configuration. In the original work, they focus their approach on energy functions defined over discrete labels, although the method can be generalized to continuous ones. In general, the energy function $E(\mathbf{x}, \theta)$, where $\mathbf{x}$ refers to the discrete vector of nodes, ϕ refers to a potential and θ refers to the model's parameters, has the form

$$
E(\mathbf{x}, \theta) = \langle \theta, \phi(\mathbf{x}) \rangle. \tag{2.24}
$$

After this, we perturb the parameters of the energy function using samples ϵ coming from a known

noise distribution $f_\epsilon(\epsilon)$. In the original work, the authors mention the Gumbel distribution, which has important properties for the design of the method. This noise is summed to the model parameters, as seen in Equation (2.25). The density $f_\epsilon(\epsilon)$ must be designed in such a way that it closely approximates the standard Gibbs Markov random field, whose probability is $f(\mathbf{x}, \theta) \propto \exp(-E(\mathbf{x}, \theta))$. Thus,

$$\widetilde{\theta} = \theta + \epsilon, \quad \epsilon \sim f_\epsilon(\epsilon). \tag{2.25}$$

Finally, there is the optimization part which allows us to obtain a sample $\widetilde{\mathbf{x}}$, where the spin configuration which minimizes the perturbed energy is found,

$$\widetilde{\mathbf{x}} = \arg\min_{\mathbf{x}} E(\mathbf{x}, \theta + \epsilon). \tag{2.26}$$

Therefore, the PM method facilitates efficient sampling, being particularly beneficial in scenarios where traditional sampling methods pose computational challenges. While the PM method presents clear advantages, it is essential to understand its intricacies, especially concerning the design and injection of noise processes. In a Gibbs Markov random field, configurations with lower energy typically have higher probabilities. However, this intuitive relationship does not always hold in the PM model. A state with a lower energy might not necessarily be the most probable state in the PM model. Consequently, the selection of an appropriate perturbation density becomes crucial, to ensure that perturbations align with the structural properties of the Gibbs Markov random field. For more references on this see Section 4 in the original work [28].

2.3 Uncertainty Estimation Methods

This section delves into three classical methods of quantifying uncertainty, presented in interconnected sub-Sections: 2.3.1, 2.3.2, and 2.3.3. By examining the principles behind *softmax* classifiers, bayesian neural networks (BNNs) with Monte-Carlo (MC) dropout, and BNNs with variational inference (VI), we aim to establish a strong theoretical foundation on how this measure is quantified and to, further ahead, compare them with our proposed model.

2.3.1 Softmax Classifier

In classification-type tasks, a simple way to measure uncertainty is to consider a NN architecture with a *softmax* activation function in the output layer. For a binary classification problem with two classes, the output layer should have 1 node. This single node can represent the probability of the input belonging to the positive class, and the complementary probability represents the negative class. The activation function used in this case is often the *sigmoid* function, which ensures that the predicted outputs are

probabilities summing up to 1. For multiclass classification problems with three or more classes, the output layer should have a number of nodes equal to the number of classes. Each node in the output layer represents the probability of the input belonging to a specific class. The predicted probabilities across all nodes should sum up to 1, and they can be obtained using the *softmax* activation function. Training is done using the cross-entropy loss function which is the negative of the sum of the log of each class probability, weighted by the same probability value. In a scenario with N samples and C classes, the term $p_{i,c}$ refers to the predicted probability for sample i in class c and $y_{i,c}$ is a binary indicator that takes the value 1 if class label c is the correct classification for observation i and 0 otherwise. The loss expression is

$$\text{Cross Entropy Loss} = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c}). \tag{2.27}$$

This is an advantageous method when compared to non-probabilistic models like, for example, the K-nearest neighbours (KNN) classifier (see sub-Section 1.4.2. in [18]). After a model produces probabilities, uncertainty can be measured using various statistical metrics and techniques. These methods use the predicted probabilities and quantify, for example, the dispersion or spread of predicted probabilities and assess the model's confidence in its predictions. Not that the UNN model belongs to the category of *softmax* classifier methods.

2.3.2 Bayesian Neural Network via Variational Inference Approximation

Bayesian Neural Networks (BNNs) are proposed as an alternative to regular NNs as better uncertainty estimators, which tend to overfit and become overconfident in their predictions (see sub-Sections 1.3.3 - 1.3.5 in [7]). In Chapter 1, we saw an example of why this measurement is crucial, but other examples can be seen in [29] for medical image segmentation and in [30] for internet traffic classification. The deterministic behaviour of the weights of a regular NN model is a key factor that contributes to its overconfident predictions, leading the model to be unable to express a range of potential outcomes in its predictions. This is where BNNs introduce the main difference with respect to the regular NNs via the integration of Bayesian principles into the NN framework. The main difference is, therefore, the introduction of stochastic weights, which allow for an expression of a spectrum of values for each weight in the network, rather than a single deterministic one. This empowers BNNs to provide richer insights into the inherent uncertainty within the data and consequently makes more informed and cautious predictions. Therefore, we consider that we have a prior distribution $p(\theta)$ defined for the set of weights θ, *i.e.*, we define a spectrum of values that we are willing to consider for the model's parameters. Additionally, we introduce the concept of the posterior distribution $p(\theta|D)$, which captures the updated beliefs about the model's parameters after observing the data $D = \{(x_i, y_i)\}_{i=1}^{n}$. The posterior distribution takes into

account both the prior distribution $p(\theta)$ and the likelihood of the observed data given the parameters. Using the Bayes rule (see Section 3.11 in [4]), we have that:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}. \tag{2.28}$$

We note that the Bayesian rule provides a systematic way to update our beliefs about model parameters based on observed data, incorporating both our prior beliefs and the evidence from the data. In practice, computing the exact posterior distribution can be challenging or an intractable task. We can see that because the normalization factor $p(D)$ is equal to:

$$p(D) = \int_{\theta} p(D|\theta)\ d\theta. \tag{2.29}$$

If θ is of high dimensionality, evaluating the integral across all dimensions can become very complex to solve analytically. This is where approximation techniques like VI or Markov chain Monte Carlo (MCMC) methods come into play. In fact, there are many possible ways to proceed and the one presented below is a VI-type approach.

In the case of VI, the objective is to approximate the posterior distribution $p(\theta|D)$ by a simple distribution denoted as the variational distribution $q_\lambda(\theta)$, usually a Gaussian one. VI is used to find the set of parameters $\lambda = \{(\mu_1, \sigma_1), ..., (\mu_K, \sigma_K)\}$, where K is the number of parameters in the variational distribution, such that the latter one is close to the posterior one. The distance between probability distributions is usually measured using the Kullback-Leibler (KL) divergence, and therefore we calculate it over the variational distribution and the posterior one:

$$\mathsf{KL}\left[q_\lambda(\theta)\ ||\ p(\theta|D)\right] = \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{p(\theta|D)}. \tag{2.30}$$

It can be proved (see sub-Section 8.2.1 in [7]) that minimizing this KLs divergence can be done by minimizing the following equation with respect to λ:

$$(\lambda)_* = \arg\min\ \{\mathsf{KL}[q_\lambda(\theta)\,||\,p(\theta)] - \mathbf{E}_{\theta\sim\mathbf{q}_\lambda}[\log(p(D|\theta))]\}. \tag{2.31}$$

We usually are interested in computing what we call the predictive distribution $p(y_{test}|x_{test}, D)$ for a given test sample x_{test}, which can be approximated by the variational distribution:

$$\begin{aligned} p(y_{test}|x_{test}, D) &= \int_{\theta} p(y_{test}|x_{test}, \theta)p(\theta|D)\ d\theta \\ &\approx \int_{\theta} p(y_{test}|x, \theta)q_\lambda(\theta)\ d\theta. \end{aligned} \tag{2.32}$$

After finding the optimal configuration of the parameters, the model can be set for testing purposes. To do it, we can now run our model multiple times for a sample x_{test}, say K times, using different samples $\theta_i \in \Theta = \{\theta_i\}_{i=1}^{K}$ for having different predictions $F_{\theta_i}(x)$. After that, we do an average model prediction to give the relative probability of each class, as stated in Equation (2.33),

$$\hat{p} = \frac{1}{|\Theta|} \sum_{i=1}^{K} F_{\theta_i}(x_{test}), \tag{2.33}$$

followed by taking the class with the highest probability as the predicted label, as stated in Equation (2.34),

$$\arg\max_{i} p_i \in \hat{p}. \tag{2.34}$$

2.3.3 Bayesian Neural Network via MC Dropout

The *dropout* technique is predominantly utilized as a regularization method. Its primary function is to randomly deactivate certain neurons in the hidden layers during training. This prevents neurons from co-adapting and ensures they actively contribute to learning input representations. While dropout is typically deactivated during testing in conventional neural networks, retaining it transforms the process into a Bayesian approach. This method, termed MC *dropout* [31], aligns with a VI-type approach where the variational distribution is defined for each weight matrix, as:

$$\begin{aligned} z_{i,j} &\sim \text{Bernoulli}(p_i), \\ \mathbf{W_i} &= \mathbf{M_i} \cdot \text{diag}(\mathbf{z_i}). \end{aligned} \tag{2.35}$$

In the aforementioned equation, $z_{i,j}$ denotes a sample from a Bernoulli distribution, where p_i is the dropout probability of the i-th layer. The index j corresponds to the j-th node within the i-th layer. Typically, this dropout probability is user-defined and can be optimized during training. The matrix $\mathbf{M_i}$ represents the weight matrix prior to the application of dropout, while $\mathbf{W_i}$ is the resultant weight matrix post-dropout. The term $diag(\mathbf{z_i})$ is a diagonal matrix constructed using the random activation coefficients z_i. In practice, during testing, executing multiple passes yields a prediction distribution for a specific test sample. This distribution facilitates the computation of uncertainty measures.

2.4 Reparametrization Trick

In many generative models like variational auto encoders (VAEs) or Generative Adversarial Networks (GANs), it is common to find some form of stochasticity in the network. Stochasticity refers to the randomness generation process of the model, which is introduced by the presence of some latent variables or random noise sources. In VAEs the latent variables follow a probability distribution, typically a multivariate Gaussian distribution, represented by $\mathbf{z}$ in the left image of Figure 2.7. These latent variables capture the underlying factors of variation in the data and introduce randomness during the generation process. By sampling from the latent variable distribution, different samples can be generated, resulting in diverse outputs. However, in order to train the model using backpropagation, we need to propagate the gradients through the network, which is not possible to do if we have a stochastic node. In fact, the differentiation process requires a deterministic mapping between inputs and outputs. Therefore, this problem is tackled using a reparametrization trick that decouples the stochasticity introduced by the latent variables from the parameters of the model. It involves reformulating the sampling of latent variables as a differentiable transformation, allowing the gradients to flow through the network, denoted by $g(\phi, \mathbf{x}, \epsilon)$ in Figure 2.7.

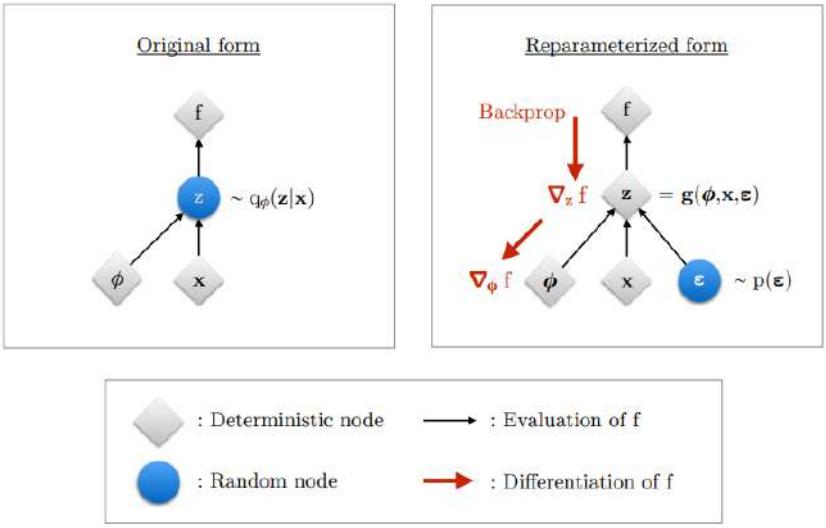


Figure 2.7: Illustration of the reparameterization trick. On the left, gradients cannot be directly backpropagated through the random variable $\mathbf{z}$. On the right, the randomness in $\mathbf{z}$ is externalized by reparameterizing the variable as a deterministic and differentiable function g which allows for backpropagation through $\mathbf{z}$. Figure taken from [2].

To visualize this problem using mathematical notation. we can recall the common differentiable loss function, named evidence lower bound (ELBO), that is used in the context of VAEs. In VAEs, the primary goal is to model and generate data effectively. To achieve this, we need to understand the underlying structure of the data, which is captured by the latent variables $\mathbf{z}$. The true posterior distribution $p_\theta(\mathbf{z} \mid \mathbf{x})$

provides insight into how these latent variables are distributed given a specifically observed data point x. However, computing the true posterior is often intractable. Therefore, we approximate this posterior with another distribution $q_\phi(\mathbf{z}, \mathbf{x})$, by learning its parameters ϕ using an optimization procedure. Therefore we use the KL divergence to minimize the difference between the true and the approximate distributions, and it can be shown that minimizing the KL is achieved by maximizing the ELBO function (see Chapter 2 in [32] for more detail in this derivation). The loss function $\mathcal{L}_{\theta,\phi}(\mathbf{x})$ is defined as:

$$\mathcal{L}_{\theta,\phi}(\mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\ p_\theta(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log\ \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})}\right], \tag{2.36}$$

where θ refers to the decoder parameters and ϕ to the encoder parameter. It contains two components: an expectation error reconstruction loss that ensures that the data reconstructed from the latent space is as close as possible to the original data, and a KL divergence term between the encoder's output distribution and a prior distribution to prevent the encoder from mapping input data to arbitrary locations in the latent space. The term $p_\theta(\mathbf{x}|\mathbf{z})$ refers to the conditional probability of the data $\mathbf{x}$ given a particular latent variable $\mathbf{z}$. This term represents the likelihood of reconstructing the original data $\mathbf{x}$ from the latent variable $\mathbf{z}$, modelled by the decoder. The term $q_\phi(\mathbf{z}|\mathbf{x})$ is the variational distribution over latent variables $\mathbf{z}$ given the input data $\mathbf{x}$. This distribution is parameterized by the encoder network in VAEs and it learns to map the input data $\mathbf{x}$ to a distribution in the latent space. The term $p_\theta(\mathbf{z})$ is the prior distribution over the latent variable $\mathbf{z}$, often assumed to be a standard Gaussian, representing our prior knowledge about the distribution of the latent variable. Maximizing this function leads to a model that aims to both accurately reconstruct the input data from the latent vector and encourage the approximate posterior distribution to align with the prior distribution, promoting regularization and learning of meaningful latent representations. The problem lies in estimating the gradient of the loss with respect to the model parameters, more specifically, the encoder ones. We can first simplify the loss expression as:

$$\begin{aligned}
\mathcal{L}_{\theta,\phi}(\mathbf{x}) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})}\right] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z}) - \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p_\theta(\mathbf{z})}\right] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}) + \log p_\theta(\mathbf{z})\right] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}, \mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right].
\end{aligned} \tag{2.37}$$

Now we separately take the derivative with respect to the encoder parameters (ϕ) and decoder parameters (θ). These gradients are essential for training the model using optimization algorithms such as SGD or other optimizer functions. Gradients are used to update the model's parameters iteratively, allowing the VAE to learn and improve its performance. For the decoder ones we simply apply the Leibniz Integral rule (see Equation 18.14 in [4]) and then approximate the last expectation with a Monte Carlo estimator, as:

$$
\begin{aligned}
\nabla_\theta \mathcal{L}_{\theta,\phi}(\mathbf{x}) &= \nabla_\theta \mathbb{E}_{q\phi(\mathbf{z}|\mathbf{x})} \left[\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right] \\
&= \mathbb{E}_{q\phi(\mathbf{z}|\mathbf{x})} \left[\nabla_\theta(\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}))\right] \\
&\simeq \nabla_\theta(\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})) \\
&= \nabla_\theta \left(\log p_\theta(\mathbf{x},\mathbf{z})\right).
\end{aligned}
\tag{2.38}
$$

.

However, in the derivation computed with respect to the encoder parameters, the derivative can not be moved inside expectation. It happens that the expectation is taken with respect to the distribution $q_\phi(\mathbf{z}|\mathbf{x})$, so in general:

$$
\begin{aligned}
\nabla_\phi \, \mathcal{L}_{\theta,\phi}(\mathbf{x}) &= \nabla\phi \mathbb{E} q_\phi(\mathbf{z}|\mathbf{x}) \left[\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right] \\
&\neq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \left[\nabla_\phi(\log p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x}))\right].
\end{aligned}
\tag{2.39}
$$

It turns out that this problem can be overcome using the reparameterization trick, a change of variable and the application of the *law of the unconscious statistician* (see Section 3.8 in [4]). The change of variable allows us to rewrite a sample from an arbitrary (and more complex) distribution as a transformation function applied over a base (and simpler) distribution. This transformation function depends, of course, on the parameters of the more complex distribution. For a Gaussian random variable example, we have:

$$
\begin{aligned}
p_\theta(x) = N(x;\mu,\sigma), \text{ where } \theta = \{\mu,\sigma\}, \\
x = \mu + \sigma \cdot \epsilon \quad \text{where} \quad \epsilon \sim N(0,1).
\end{aligned}
\tag{2.40}
$$

The *law of the unconscious statistician* (LOTUS) allows calculating the expected value of a measurable function of a random variable X, without actually knowing the density function of that transformed random variable, but only the distribution of X, the "base" random variable, as:

$$
\mathbb{E}[f(X)] = \int_{\mathbb{R}} f(x) p_\theta(x)\, dx. \tag{2.41}
$$

Therefore, for the ELBO loss function we have:

$$
\left.
\begin{aligned}
\epsilon &\sim p(\epsilon) \\
\mathbf{z} &= g(\phi,\mathbf{x},\epsilon)
\end{aligned}
\right\} \text{Change of variable,} \tag{2.42}
$$

$$
\mathcal{L}_{\theta,\phi} = \underbrace{\mathbb{E}_{p_{(\epsilon)}}}_{\text{LOTUS}} \left[\log \; p_\theta(\mathbf{x},\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right]. \tag{2.43}
$$

Finally, we can compute the following gradient and transpose it inside the expectation, solving the differentiability problem we had back in Equation (2.39):

$$\nabla_{\phi}\mathcal{L}_{\theta,\phi}(\mathbf{z}) = \nabla_{\phi}\mathbb{E}_{p_{(\epsilon)}}\left[\log\ p_{\theta}(\mathbf{x},\mathbf{z}) - \log q_{\phi}(\mathbf{z}|\mathbf{x})\right]. \tag{2.44}$$

This illustrates the critical consideration of effectively managing stochastic elements within NN architectures, a factor of particular relevance in SUNN models, as they inherently incorporate stochastic nodes in their design. In the subsequent section, we will revisit the previously mentioned models and frameworks, conducting a comparative analysis with our proposed model. This analysis will highlight both the commonalities and distinctions inherent in these structures.

3

Stochastic and Undirected Neural Networks

Contents

Having introduced the necessary background, we now proceed to present the details of the model we propose, which is an EBM with stochastic terms added. We called it SUNN - a generalization of UNN where we now add stochastic nodes to the architecture. In Section 3.1, we cover these aspects more in-depth. Next, in Section 3.2, we establish comparisons with some of the models presented in Chapter 2 and finally, in Section 3.3, we present an approach used for training the model.

3.1 Architecture Design

As it was presented before, the UNN cannot be used as a generative model, unless under it has random initializations in the network, although they do not necessarily have to be present since there are other types of non-random initializations. The way we introduce stochasticity is by introducing extra nodes in the FG. In consequence, the FG is now composed by a set of deterministic nodes $V = \{X_1, ..., X_M\}$ with respective representation vectors $\{x_1, ..., x_M\}$, where $x_i \in \mathbb{R}^{d_{X_i}}$, and stochastic nodes $W = \{N_1, ..., N_K\}$ with respective representation vectors $\{n_1, ..., n_K\}$, where $n_i \in \mathbb{R}^{d_{N_i}}$, and whose joint probability distribution is given by $p(n_1, ..., n_K)$. In general, an iteration in a SUNN model is given by:

$$\begin{aligned} (n_1, ..., n_K) &\sim p(n_1, ..., n_K), \\ (x_\pi)_* &= \arg\min_{x_\pi} E(x_1, \dots, x_M, n_1, \dots, n_K), \end{aligned} \tag{3.1}$$

where π refers to a particular order of updating the energy values of the non-stochastic terms. In this work, we apply some restrictions over the FG and the stochastic terms. First and foremost, we also consider one stochastic term for each hidden and output layer of the network, which means $K = M - 1$. Lastly, we assume that the global energy function is composed of three types of energy terms: the unary energy potentials and the pairwise energy terms which are assumed to be the same ones adopted for UNNs, seen in Equation (2.19), and a linear energy term for the interaction of the stochastic terms with the nodes, where the stochastic terms are assumed to be independent and standard Gaussian distributed with known σ^2 variance, given by:

$$\begin{aligned} E_{X_i N_{X_i}}(x_i) = -\langle n_{X_i}, x_i \rangle, \quad n_{X_i} \sim N(0, \sigma^2 I_{d_{X_i}}), \quad \sigma \text{ known}, \\ n_{X_i} \perp n_{X_j} \forall\, i \neq j. \end{aligned} \tag{3.2}$$

In Section 2.4, the importance of handling stochastic nodes within a NN structure was discussed. In the proposed model, both the sampling procedure and the noise introduction utilize an approach similar to the final result of applying the reparametrization trick, avoiding compromising the differentiation process in the network. It is crucial to emphasize that, based on our assumptions, the network does not have any stochastic nodes inside the network (between layers). Instead, and as depicted in the example

seen in the second image of Figure 3.1, the stochasticity is parallel to the network. Therefore, we have some stochasticity added to a deterministic component, which is similar to the result of applying the reparametrization trick to VAEs, as seen on the right-side image of Figure 2.7. With this approach, the backpropagation in the SUNNs model is still "forcing" low energy to align with the desired behaviour by adjusting the model's parameters, just like it happened with the UNN. It uses gradient information to navigate the energy landscape and create configurations of states that correspond to low-energy regions for correct predictions and high-energy regions for incorrect ones. This optimization process ensures that the energy function captures the relationships and constraints necessary for successful task execution. Another perspective is to recognize that the stochastic elements have shifted from the random nodes directly to the bias term present in the initial unary energy potential function, as:

$$\begin{aligned} E_{X_i}(x_i) + E_{X_i N_{X_i}}(x_i) &= -\langle b_{X_i}, x_i\rangle + \Psi_{X_i}(x_i) - \langle n_{X_i}, x_i\rangle \quad n_{X_i} \sim N(0, \sigma^2 I_{d_{X_i}}), \quad \sigma \text{ known}, \\ &= -\langle b_{X_i} + n_{X_i}, x_i\rangle + \Psi_{X_i}(x_i). \end{aligned} \tag{3.3}$$

In the previous equation, b_i refers to the bias term of a general node X_i and $\Psi_{X_i}(x_i)$ to the regularizer applied to the same node. Introducing stochasticity inside the network when adding stochastic nodes in the FG is equivalent to directly injecting noise in the bias term of the unary potentials. We can now derive the mean value and variance of the sum of energy terms $E_{X_i}(x_i) + E_{X_i N_{X_i}}(x_i)$, when b_i, x_i, n_{X_i} are vectors:

$$\begin{aligned} \mathbb{E}[E_{X_i}(x_i) + E_{X_i N_{X_i}}(x_i)] &= \mathbb{E}[-\langle b_{X_i}, x_i\rangle + \Psi_{X_i}(x_i) - \langle n_{X_i}, x_i\rangle] \quad n_{X_i} \sim N(0, \sigma^2 I_{d_{X_i}}), \quad \sigma \text{ known}, \\ &= -\langle b_{X_i}, x_i\rangle + \Psi_{X_i}(x_i) \\ &= -b_{X_i}^T x_i + \Psi_{X_i}(x_i), \end{aligned} \tag{3.4}$$

$$\begin{aligned} \mathbb{V}[E_{X_i}(x_i) + E_{X_i N_{X_i}}(x_i)] &= \mathbb{V}\left[-\langle b_{X_i}, x_i\rangle + \Psi_{X_i}(x_i) - \langle n_{X_i}, x_i\rangle\right] \quad n_{X_i} \sim N(0, \sigma^2 I_{d_{X_i}}), \quad \sigma \text{ known}, \\ &= \mathbb{V}\left[-\langle n_{X_i}, x_i\rangle\right] = \sigma^2 \sum_{i=1}^{d_{X_i}} x_i^2. \end{aligned} \tag{3.5}$$

This leads us to conclude that $E_{X_i}(x_i) + E_{X_i N_{X_i}}(x_i) \sim N(-x_i^T b_{X_i} + \Psi_{X_i}(x_i), \sigma^2 \sum_{i=1}^{d_{X_i}} x_i^2)$. The assumptions initially proposed are equivalent to defining unary stochastic terms with the previous mean and variance expressions, as seen in Equations (3.4) and (3.5), and as illustrated in the right-most image of Figure 3.1. Regarding the update expression, it will be very close to Equation (2.20), but now with the added stochastic component, as presented in:

$$
\begin{aligned}
(x_i)_* &= \arg\min_{x_i} E_{X_i}(x_i) + \sum_{f=\{X_i,X_j\}\in F(X_i)} E_f(x_f) + E_{X_i N_{X_i}}(x_i) \\
&= \arg\min_{x_i} \Psi_{X_i}(x_i) - b_{X_i}^T x_i - \sum_{f=\{X_i,X_j\}\in F(X_i)} \langle W_f, \underset{j\in f}{\otimes} x_j \rangle - n_{X_i}^T x_i \\
&= \arg\min_{x_i} \Psi_{X_i}(x_i) \underbrace{-(b_{X_i} + n_{X_i})^T x_i - \sum_{f=\{X_i,X_j\}\in F(X_i)} \langle W_f, \underset{j\in f}{\otimes} x_j \rangle}_{-z_i^T x_i} \\
&= (\nabla \Psi^*_{X_i})(z_i), \\
\text{where} \quad z_i &= \sum_{f=\{X_i,X_j\}\in F(X_i)} \rho_i(W_f) x_j + (b_{X_i} + n_{X_i}).
\end{aligned}
\tag{3.6}
$$

In the last equation, the term ρ_i refers either to the identity or the transpose operator, depending on the position of the adjacent nodes with respect to the node X_i. In the illustrative example of Figure 3.1, we see a FG with this step up where we have three non-stochastic terms and two stochastic ones.

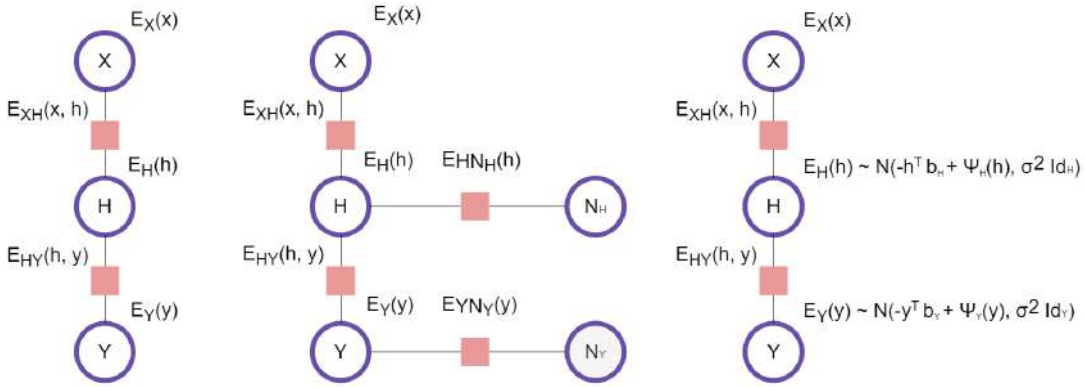


Figure 3.1: From left to right, FG for: network with one intermediate layer and no stochastic nodes, network with one intermediate layer and one stochastic node for each node in $V = X, H, Y$, network with redefined energy potentials.

From the first to the second image, we see the incorporation of the stochastic terms in the network, and from the second to the third image we see two equivalent approaches of introducing stochasticity in the network. In the examples further ahead presented, we will consider the equations of the second image, *i.e.* the framework with deterministic unary and pairwise interaction energy terms, plus the stochastic energy terms. Another important remark is that the equations (3.2) to (3.5) can be adjusted to suit the specific operations within the network. These adaptations should ensure that the dimensions of the nodes and weights are appropriately accounted for. This flexibility in equation adaptation is particularly relevant in the context of the setup discussed in Section 4.5, as we will explore in more detail later.

3.2 Comparative Analysis with other Approaches

Building upon the foundational concepts of our proposed model, this subsection delineates the parallels and distinctions between our model and those elaborated in Chapter 2.

In sub-Section 2.2.1, we highlighted the intrinsic weight learning mechanism of classic HN through the Hebbian learning rule, facilitating pattern reconstruction. This method necessitates the entirety of the data for weight computation prior to reconstruction, a stark contrast to the training rule of the SUNN. While our discussion centred on the traditional Hebbian learning for the HN, alternative methodologies do exist as referenced in [33], [34], and [35]. The commonality with SUNN models emerges in the post-learning energy minimization phase: the HN employs internal operations to minimize the global energy function for pattern reconstruction, just like the SUNN leverages this for sample generation or label classification. One difference relies on the stochastic component, which exists in the SUNN but not in the HN. The latter is a deterministic system: given the same initial state and the same sequence of neuron update orders, they will always produce the same final state, which is clearly different from the SUNN. As for the restricted BM, this model is seen as the incorporation of stochasticity to the deterministic HN. The training is done via Gibbs sampling or contrastive divergence methods, which brings it closer to traditional ML models, especially because they are also gradient-based approaches. This training method utilizes a cost function, like the negative log-likelihood function, which is optimized using a gradient-based approach — a characteristic shared with the SUNN and the UNN models. During the training phase, the restricted BM adjusts the network's weights to favour training samples by assigning them higher probabilities, corresponding to low-energy states, and reducing the probability for samples generated by the model. In the inference phase, the network seeks configurations with the lowest energy. Similarly, during testing for the SUNN, given initial conditions on the input or output layer, the network undergoes operations to minimize the global energy, aiming to generate representations in the network's remaining layers. The BNN trained using VI learns the parameters of a Gaussian distribution that captures the characteristics of the true posterior, respects a specified Gaussian prior distribution, and aligns with the observed training data. In particular, the training rule is more closely related to the SUNN one (and the UNN one) since it has two terms controlling two distinct but fundamental aspects: on one hand, the negative log-likelihood term ensures that the model's predictions match the observed data as closely as possible, which encourages the model to fit the training data for prediction purposes. The lower the negative log-likelihood, the better the model's predictions align with the training data. This term ensures that the BNN produces accurate predictions, which is a fundamental aspect of any neural network. It is analogous to the forward loss function used in the UNN and SUNN models. On the other hand, the KL divergence term serves as a regularization term that ensures the approximating distribution remains close to the specified prior distribution. This term controls the complexity of the model and helps prevent overfitting. It encourages the BNN to respect the prior beliefs about the parameters. A smaller

KL divergence indicates that the model is not deviating significantly from the prior. Globally speaking, this ensures a balance between adhering to the prior and fitting the data to produce meaningful uncertainty estimates and avoiding extreme parameter values. Our proposed model is similar in the first term but different in the second one, where instead of keeping the weights' distribution close to one of reference, we optimize the parameters for a generative purpose. The PM is similar to the SUNN in terms of procedures to follow. In both methods, we perturb the models' parameters (in the case of the SUNN, only the bias terms) and then we minimize the energy function. In the SUNN we update the energy terms using a specific energy minimization equation with the general expression given in Equation (3.6), while in the PM different procedures may be used like graph cuts, linear programming relaxations, or loopy belief propagation, according to the authors. Furthermore, the objective is a bit different, because in the SUNN model, the energy minimization objective can be two-fold, being used to generate representations in the input layer and output layer at the same time. The PM on the other hand is used for sampling purposes, for example in the context of image segmentation where the problem can be modelled by a Markov random field, which can capture local dependencies between pixels and the goal is to assign a label to each pixel.

3.3 The Learning Algorithm

Every ML model necessitates a learning rule to effectively adjust its parameters for a given objective. Given their non-directed attribute, SUNNs exhibit the capacity to be concurrently trained for two tasks. For example, a SUNN model functioning as an object classifier can also serve as a prototype generator, producing new objects akin to those in the training set. Consequently, two suitable loss functions are required to quantify the disparity between a model's predicted outputs and the actual ground truth labels. Additionally, it becomes imperative to designate an optimizer that guides parameter updates by considering the loss function gradients.

In this context, the loss function $L(\mathbf{x}, \mathbf{y}, \hat{\mathbf{x}}, \hat{\mathbf{y}})$ defined in [1] was chosen. This loss function combines a *Multi-Class Cross Entropy* loss denoted as L_{MCCE} for the classification task and a *Binary Cross Entropy* loss, referred to as L_{BCE}, for the prototype generation task. The weighting of the second loss function is determined by the factor α, thereby positioning the second task as a means of regularization or penalty on the weights associated with the classification task. It is important to note that the precise value of the penalty term remains subjective, and alternative terms can be employed. Within the results Chapter 4, we provide outcomes for varying values of this term alongside other hyperparameters. For a set of N inputs in a dataset with C classes, the term x_i refers to an input sample of index i, the term $y_{i,c}$ is a binary indicator (0 or 1) if class label c is the correct classification for sample i, the term $\hat{x}_i$ refers to the reconstructed input samples i and $p_{i,c}$ refers to the predicted probability for class C, for the sample

i. The term α is the weight given to the backward loss function. Then, we have:

$$\begin{aligned} L(\mathbf{x}, \mathbf{y}, \hat{\mathbf{x}}, \hat{\mathbf{y}}) &= L_{MCCE} + \alpha L_{BCE} \\ &= -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c}) - \alpha\sum_{i=1}^{N}[x_i\log(\hat{x}_i) + (1-x_i)\log(1-\hat{x}_i)] \end{aligned} \tag{3.7}$$

The *Multi-Class Cross Entropy* loss function measures the dissimilarity between the predicted probabilities and the true labels for each class of each sample. It strongly penalizes incorrect predictions and encourages the model to predict high probabilities for the true class. The entire loss function is a combination of these penalties across all classes and samples, scaled by the number of samples. The *Binary Cross Entropy* loss function achieves a similar goal, but it is designed for scenarios with two classes only. This loss is calculated in the following way: the input images $\{x_1, ..., x_n\}$, which were pre-processed to have pixel values between -1 and 1, are changed to 0 or 1 values depending on the pixel value being less or greater than 0, respectively. The reconstructed inputs $\{\hat{x}_1, ..., \hat{x}_N\}$ are obtained by passing through a *sigmoid* function every pixel of the image, which is found in the input layer of node X. Therefore, all the pixel values are squashed into probability values. Globally, this dual loss function aims to compel the model to make predictions as accurate as possible by measuring the difference between predicted probabilities and true labels and, at the same time, to be able to recreate the input images provided during training.

When considering the choice of optimizer, a variety of options are available within the *PyTorch* framework, such as SGD, *Adam*, *RMSprop*, *Adamax*, and more (see [36]). In our approach, we initially opted to employ SGD for image classification tasks, as prior research has demonstrated its potential to outperform *Adam* in these specific contexts [37]. However, initial experiments with SGD revealed suboptimal performance compared to *Adam*. While convergence was observed with SGD, it occurred at a significantly slower rate, likely attributed to the fixed learning rate it employs. In consequence, we tested the use of *Adam* as an alternative optimizer, influenced by its application in the original work of the UNN model [1]. This decision was further reinforced by the absence of significant exploration of other optimizers. The latter one proved to be much more effective in the learning process of the SUNN model, and therefore it was used throughout the experiments of Chapter 4.

4

Experiments and Results

Contents

Chapter 4 presents all the experiments done in this work. First, in Section 4.1. it introduces the dataset used in this work. Next, in Section 4.2, an important tool used for model training is introduced, and in Section 4.3 the evaluation criteria and the benchmarking models are presented. Sections 4.4 and 4.5 contain the results and respective analyses. These two latter ones are divided into further sub-sections, focusing on different aspects of the experiments.

4.1 Dataset Description

The undirected and generative properties of the proposed model allow it to be applied to many tasks. We decided to experiment with our model in the *MNIST* dataset [9]. This dataset, widely used in ML as a benchmark, consists of a collection of handwritten digits commonly used for training and evaluating image classification algorithms. The *MNIST* dataset contains 70000 images, each of which is a gray-scale image with a resolution of 28×28 pixels. The images are normalized and centred, making them consistent in terms of size and position. Each image is associated with a corresponding label, which represents the digit depicted in the image ranging from 0 to 9. We decided to use 50000 images for training, 10000 for validation, and the remaining 10000 for testing. This partition of the dataset and the images each set contains are consistent throughout all the experiments done in this work. We also used a noisy version of the *MNIST* test images, where we added Gaussian Noise to each image of this set. In Chapter 4 we specify with better detail the amount of noise injected into the images. All the experiments were done using Google Colab, which provides a free GPU in a cloud environment, an important feature for training ML models faster. Many results were obtained in both premium and non-premium GPUs that Google Colab provides (T4, V100 and A100). Due to the limitations of this cloud environment, some experiments had to be run locally, on an 11th Gen Intel(R) Core(TM) i5-1135G7 processor. In terms of pre-processing of the images, we did a conversion of the images from the *PIL* format to *Tensors* and a scaling to the $[-1, 1]$ interval. This scaling process helps stabilise the gradients used to update the model parameters. When the input data has a large range or varying scales, the gradients may become unstable, leading to slow convergence or divergence. Normalizing the data mitigates this issue and helps to maintain stable and consistent gradients throughout the network.

4.2 Using Weights and Biases Platform

The process of tuning a ML model can be time-consuming and computationally expensive. These models generally have many hyperparameters that need to be tuned for a particular application, which generally involves running many experiments with different parameter configurations and keeping track of the experiment results. It is important to tune some of the hyperparameters of the model in order to

obtain a model that yields better results. This process may become very complex, especially if we have many hyperparameters to tune, which can lead to a combinatorial explosion: as the number of hyperparameters increases, the possible combinations of parameter values grow exponentially. For example, if we have three hyperparameters, each with four possible values, we already have $3^4 = 81$ combinations. In order to store the results and compare different configurations, we decided to use *Weights and Biases* [38], which is a platform suitable for this purpose. For the experiments done in Sections 4.4 and 4.5, we decided to tune the values of the hyperparameters seen in Table 4.1.

Table 4.1: List of hyperparameters for the optimization of UNN and SUNN models.

Hyperparameter	Values/Range
dropout	0.1 – 0.5
learning_rate	0-0005 – 0.1
batch_size	128, 256, 512
unn_iter	1, 3, 5, 10
unn_y_init	uniform, rand, zero
backward_loss_coef	0.01 – 1
activation	relu, tanh
noise_sd	0 – 1

For each experiment, we first ran 50 different combinations of parameters using the ranges of values provided in the table. Additional runs could not be made due to computational limitations. The platform names every run as a ”sweep”, as it will be presented in sections related to the results. Regarding the dropout parameter (in Table 4.1: *dropout*), it is one of the most common methods of regularization of ML models [39]. As the name suggests it drops out, randomly, some of the network neurons during training. The main idea is that by dropping some of the neurons, the remaining ones are forced to learn more generalized features. Without *dropout* there is a risk of having complex co-adaptations between the neurons *i.e.*, ”lazy” nodes that rely on each other to model a specific problem. During test time, *dropout* is turned off. A usual procedure is to start with a small range of moderated values and then adjust that range based on performance; therefore, we started with *dropout* probability values ranging from 0.1 to 0.5. The rationale was the same for the learning rate (in the Table 4.1: *learning_rate*). This parameter is related to the magnitude of the update done to the parameters. A high learning rate promotes larger updates, while a low learning rate means smaller updates. It is important to tune this hyperparameter in order to balance convergence and speed of training. High learning rates may promote divergence if the parameter increases too much. On the other hand, if the learning rate gets too low, the model may take a long time to converge or get stuck in a sub-optimal solution. Usually, high learning rate values range from 1 to 10 and low learning rate values range from 1×10^{-6} to 1×10^{-5}. We decided to start with a smaller range with values varying from 5×10^{-4} to 0.1. As for the batch size (in Table 4.1: *batch_size*), it is usually recommended to use power of 2 values (especially when using GPUs). Therefore, we decided to consider the values 128, 256, and 512, which are not too extreme (see Chapter 8, Section 3.1 in [4]).

This parameter determines the number of samples that are processed before updating the weights of the model. The parameter *unn_iter* refers to how many times we repeat the energy minimization procedure in the forward or backward pass, and here we tested the values 1,3,5 and 10 for this hyperparameter. Exemplifying, for the forward task, the sets are defined as $x_\pi = \{H, Y\}$ for the FFNN scenario and $x_\pi = \{H1, H2, Y, H2\}$ for the CNN scenario. When this parameter is set to 3, each element within the x_π set undergoes two additional update iterations, in the same order they appear in the set. The last configuration of Y is used as the final prediction for the given input. As was already mentioned, at each update the energy is minimized. Regarding the parameter *unn_y_init*, it is related to the initialization of Y. We experimented with the three configurations already tested in [1]: uniform, random, and zero initializations. We kept in mind that these could influence the model's performance as presented in the cited work, where the random initialization showed promising results. The parameter *backward_loss_coef* is related to a penalization, due to the consideration of the prototype generation task, given a label. We considered the range of values from 0.01 to 1. As for the activation function of all layers except the output one (referenced in Table 4.1 as *activation_function*), we decided to test both $relu$ and $tanh$ since they are commonly used in practice. The value of the standard deviation of the noise, *noise_sd*, ranges from 0 to 1, where the minimum value (0) allows us to recover the UNN model. Due to the high number of hyperparameters, and the many values that all of them can assume, it can be complex to intuitively say which ranges and combinations are better picks. The 50 combinations of hyperparameters were randomly computed by *Weights and Biases*, where the values are uniformly sampled from the ranges we defined above. All these 50 runs were done with the objective of finding the best configurations for the problem we were addressing. Since SUNNs have a bi-directional application we chose the best configurations taking into account the values of the loss function in the validation set for both the forward and backward tasks. For the forward task, we also considered the accuracy measured on the validation set. Finally, it is worth mentioning that we implemented an early stopping mechanism; if during training, for both loss functions, a minimal loss value was registered at least 10 epochs away from the current epoch training stops. This helps to avoid spending computational resources on runs that are not optimal.

4.3 Model Selection and Evaluation

In order to assess the viability of the proposed model, we need to select relevant evaluation metrics for that purpose. Regarding the forward task, we can evaluate it in different aspects, such as accuracy and uncertainty metrics. Since our model works as a classifier, the accuracy tells us if the model is a good digit predictor by dividing the correct predictions by the total number of predictions. Supposing we have a classification problem with C classes, we can train SUNNs in a dataset D to be able to classify new data samples. Due to their stochasticity, if we execute many times, say M, for a new data sample x, this

will induce a probability distribution estimated by the M samples $\{P_i(y|x,D)\}_{i=1}^{M}$ (can be abbreviated to $\{P_i\}_{i=1}^{M}$). After that, we can take some measures regarding that estimated probability distribution. We focus on both the entropy, Jensen–Shannon divergence (JSD) and mutual information (MI). The expression for the estimated probability distribution is

$$P(y|x,D) \approx \frac{1}{M}\sum_{i=1}^{M} P_i(y|x,D). \tag{4.1}$$

The first considered metric is the entropy of the estimated probability distribution, $\mathcal{H}[P(y|x,\theta)]$, given by:

$$\textit{entropy} = \mathcal{H}[P(y|x,\theta)] = -\sum_{c=1}^{C} P(y=y_c|x,D)\ \log(P(y=y_c|x,D)), \tag{4.2}$$

which can be considered an average of surprise, being surprise a measure of how unexpected or surprising an individual outcome of a random variable is. Here, higher entropy values indicate greater uncertainty or diversity in the distribution, while lower entropy values suggest more concentrated or certain probabilities.

The JSD measure (also known as *Information Radius (IRAD)* or *Total Divergence to the Average*) was also considered here. The JSD quantifies the variability or uncertainty in the model's predictions across different runs. It provides a measure of how much the predicted probability distributions diverge from the mean using the KL divergence. It is defined as:

$$\begin{aligned} \textit{JSD} = \frac{1}{M}\sum_{i=1}^{M} KL(P_i \mid\mid Q) \quad &= \frac{1}{M}\sum_{i=1}^{M}\sum_{\mathbf{x}\in\mathbf{X}} P_i(x)\log(\frac{P_i(x)}{Q(x)}), \\ \text{where } Q = \frac{1}{M}\sum_{i=1}^{M} P_i. & \end{aligned} \tag{4.3}$$

The MI between the categorical label y and the parameters of the model θ, $\mathcal{I}[y,\theta|x,D]$, is a measure of the spread of an ensemble of predicted probabilities, which assess uncertainty in predictions due to model uncertainty. Thus, MI implicitly captures elements of distributional uncertainty. The MI can be expressed as the difference between the total uncertainty, captured by the entropy of expected distribution, and the expected data uncertainty, captured by the expected entropy of each member, that it

$$\mathcal{I}[y,\theta|x,D] = \mathcal{H}[\mathbb{E}_{p(\theta|D))}[P(y|x,\theta)]] - \mathbb{E}_{p(\theta|D))}[\mathcal{H}[P(y|x,\theta)]]. \tag{4.4}$$

The quantification of such metrics can be somewhat hard to interpret, so simply reporting their values might not be intuitive. Therefore, an approach outlined in [40] is used. Instead of reporting the absolute values of these metrics, we will do two experiments. The first experiment, named **misclassification experiment**, involves detecting whether a given prediction is incorrect (the positive class) given an un-

certainty measure, while the second experiment, named **out of domain (OOD) detection experiment**, involves detecting if a sample is OOD (the positive class) given an uncertainty measure. Therefore, in the first experiment, we study how uncertainty measures are associated with the model's capacity to commit errors, while in the second experiment, we study how uncertainty measures are associated with the model's capacity to identify OOD samples. A similar experiment can be seen in [41]. Such performance is assessed by calculating the area under (AU) two specific curves: the area under the receiver operating characteristic (AUROC) curve and the area under the precision-recall curve (AUPR) (see sub-Section 5.7.2 in [18]). In a binary classification problem, the receiver operating characteristic (ROC) curve is obtained by plotting the estimated true positive rate (TPR) against the estimated false positive rate (FPR), using different threshold values. It allows us to identify a classifier that maximizes the TPR while minimizing the FPR. The precision-recall (PR) curve works similarly, but we consider the estimated precision and the estimated recall (or TPR), and we want to maximize both. In the case of a highly imbalanced dataset, where the positive class is in minority, only considering the ROC curve for analysis can be misleading since the ROC curve will show a low FPR. However, the PR curve will show a low precision, yielding the model's incapacity to detect samples from the minority class. This way of assessing performance has the advantage of working like an additional safety mechanism in test time, allowing the model to recognize and flag instances where it might be making incorrect predictions or encountering unfamiliar data. By leveraging the AUROC and AUPR metrics, one can set specific thresholds that balance the trade-off between true positives and false positives, ensuring that the model's predictions align with its expressed confidence. This not only enhances the reliability of the model but also provides actionable insights for real-world deployments. For instance, in critical applications such as medical diagnoses or autonomous driving, predictions flagged as potentially incorrect or OOD can be deferred for human review or further investigation. This ensures that decisions made based on the model's predictions are both informed and cautious. Furthermore, the ability to quantitatively assess the model's self-awareness regarding its predictions can lead to more robust and trustworthy machine learning systems. In essence, these metrics provide a comprehensive framework for evaluating, understanding, and leveraging a model's uncertainty estimates, making them indispensable tools for deploying automated systems in sensitive and dynamic environments. Figure 4.1, seen below, presents a confusion matrix which allows us to visualize how efficient a threshold is in distinguishing positive from negative samples. To the right of the confusion matrix, there are some ratio metrics (which correspond to the axes of the two aforementioned curves) that can be estimated using the values from the matrix. These ratios provide a more detailed understanding of the model's performance. To understand how to read a confusion matrix and the importance of each ratio see sub-Section 5.7.2 in [18]. It is imperative to note that these ratios are estimations derived from the performance of the model in a sample of data, not all the population study. They serve as approximations of the true metrics that would be obtained if we had access to the

entire population. Although we omit the term ”estimated” for brevity and in line with common practice, the reader should bear in mind the inherent uncertainty associated with these estimations when interpreting the results.

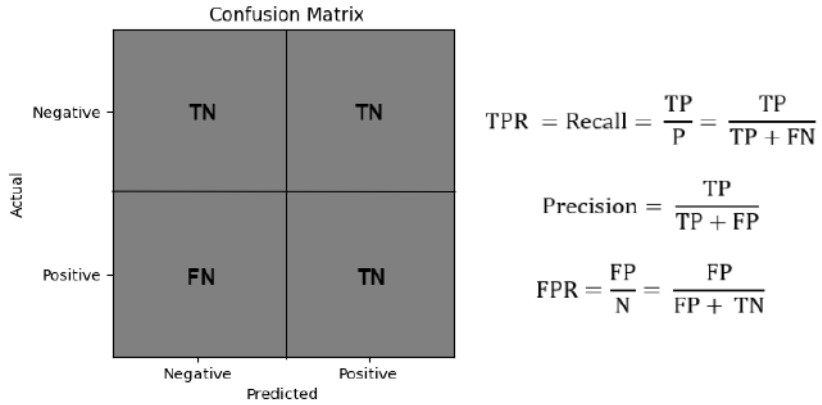


Figure 4.1: Example of a confusion matrix and formulas to estimate important performance ratios.

The metrics JSD and MI are meaningful only when computed for stochastic models, as their values are 0 for models that do not produce a probability distribution over a sample upon repeated runs on the same image (since the output is constantly the same). Consequently, we will exclude their computation for deterministic models such as the UNN. In fact, we will compare the proposed model, the SUNN, with other relevant models. We selected one UNN model, and two BNN models: one trained using MC dropout and the other trained under VI. The BNN using VI framework was created using the *Pytorch* implementation available in [42]. Some notes on the selected hyperparameters for these models as well as training metrics can be seen in the Figures included in Appendix B. Regarding image generation, we decided to compare the SUNN model with the digits generated by the UNN and the restricted BM. Although quantitative metrics do exist, the popular ones have their own limitations like being time-consuming and subjective (see [43]). In our experiments, we decided to rely solely on visual inspection. At the end of the day, the generated images will be viewed by humans, on deployed in real-world applications, whether they are used in art, media, research, or other applications. Ensuring that they pass the ”human eye test” is of paramount importance. We decided to pay close attention to the quality of the digit (*i.e.*, if it actually is a digit of the given class) and the diversity (*i.e.* if the model can produce different digits for the same class).

4.4 Models with FFNN-type Operations

Since UNNs subsume many well-known architectures, we can first build a stochastic (and undirected) version of a FFNN. They have been used for many years for classification tasks of different domains of

engineering and science [4]. In general, FFNNs are generally not used for the backward task of image generation. The operations involved in FFNNs are essentially weighted sums followed by activation functions; this includes the input image, which is treated as a flattened array. It can be challenging to decide which architecture to pick for the FFNNs, *i.e.*, the number of hidden layers, the number of nodes per layer, activation functions, and many other parameters. There has been research towards this subject of finding the best configuration (see, for example, [44]). However, we decided here to use a very simple FFNN as a benchmark - both for the classification and generation tasks. The architecture we picked here consists of an input layer where we have the flattened images - which makes it have 784 nodes, a hidden layer with 128 nodes, and the output layer with, of course, 10 nodes. Therefore, the total number of parameters in this FFNN is 101,770. The FG that corresponds to this setup is the one seen in the second image of Figure 2.5. The total energy function is, in this case:

$$\begin{aligned} E(x, y, h, n_H, n_Y) = E_X(x) + E_H(h) + E_Y(y) + E_{XH}(x, h) + E_{HY}(h, y) \\ + E_{HN_H}(h, n_H) + E_{YN_Y}(y, n_Y), \end{aligned} \tag{4.5}$$

where

$$\begin{aligned} &E_X(x) = -\langle b_X, x\rangle + \Psi_X(x) = \Psi_X(x), \quad b_X = 0, \\ &E_H(h) = -\langle b_H, h\rangle + \Psi_H(h), \\ &E_Y(y) = -\langle b_Y, y\rangle + \Psi_Y(y), \\ &E_{XH}(x, h) = -\langle h, Wx\rangle, \\ &E_{HY}(h, y) = -\langle y, Vh\rangle, \\ &E_{HN_H}(h, n_H) = -\langle n_H, h\rangle, \quad n_H \sim N(0, \sigma^2 I_{d_H}), \quad \sigma \text{ known}, \\ &E_{YN_Y}(y, n_Y) = -\langle n_Y, y\rangle, \quad n_Y \sim N(0, \sigma^2 I_{d_Y}). \end{aligned}$$

Using the update rule from Equation (3.6), the update terms, denoted by x_*, h_* and y_* are given by:

$$\begin{aligned} x_* &= (\nabla\Psi_X^*)\left(W^T h\right), \\ h_* &= (\nabla\Psi_H^*)\left(Wx + V^T y + b_H + n_H\right), \\ y_* &= (\nabla\Psi_Y^*)\left(Vh + b_Y + n_y\right). \end{aligned} \tag{4.6}$$

In the previous equation, $x \in \mathbb{R}^{784\times 1}$, $\{h, b_H, n_H\} \in \mathbb{R}^{128\times 1}$, $\{y, b_Y, n_Y\} \in \mathbb{R}^{10\times 1}$, $W \in \mathbb{R}^{128\times 784}$, $V \in \mathbb{R}^{10\times 128}$, $d_H = 128$ and $d_Y = 10$.

4.4.1 Undirected FFNN - Training Results

We opted to initiate an experiment where the inclusion of noise is omitted during the training process. This approach leads us to an undirected FFNN, as previously introduced in [1]. The primary objective of this experiment is to establish a baseline UNN model both in terms of uncertainty quantification and prototype generation. It is conceivable that when exposed exclusively to deterministic updates during training, the model could exhibit excessive confidence in its predictions, potentially overlooking the genuine uncertainty inherent in the data, similar to what happens to other similar ML models. Conversely, a model trained with the integration of randomness during the training phase might be more resilient to overfit. Consequently, it could display reduced confidence in its predictions, driven by a more cautious approach. By introducing noise throughout the training process, the model could cultivate enhanced generalization capabilities towards unseen data, consequently yielding predictions with reduced certainty in instances of uncertainty, similar to BNNs. In Figure 4.2, we present the results of the 50 runs in the validation set for the UNN model. This figure represents the accuracy, forward loss and backward loss plots generated during the training of the model.

Note that the accuracy plot does not start at the origin and some of the runs may not appear in the two loss plots because they are also truncated, and therefore only the runs with lower loss values are being shown. This was done for better visualization purposes. Our analysis centres on the validation set plots, which hold images outside the model's training scope. An intriguing result unfolds within the backward loss plot, related to the backward loss function. Here, a number of configurations become ensnared in a localized minimum, yielding approximately 19 as the loss value. However, a considerable subset of configurations manages to break free from this confinement, converging towards a more favourable minimum where the loss dips below 7. On the front of the forward pass, almost all iterations present accuracies that coalesce around diverse values surpassing the 90% mark, as showcased in the accuracy plot (on the top). Similarly, the plot illustrating the forward loss function (in the middle) demonstrates the convergence of different runs to values spanning from 0.31 to 0.10 (with only a few exceptions). It is of note that those runs exhibiting heightened accuracy correspond to smaller magnitudes of the forward loss function. Evidently, the forward loss function is filled with numerous local minima, each potentially serving as a convergence point for the model. In contrast, the backward loss function (on the bottom) appears to gravitate towards two predominant local minima. This strongly suggests that the interactions between hyperparameters can have a significant impact on the model performance. We then decided to filtrate the different results and only visualize runs with a final forward loss on the validation set smaller or equal to 0.3, accuracy on the validation set higher or equal to 97% and backward loss on the validation set smaller or equal to 7. Shedding light on this, the parallel coordinates plot in Figure 4.3 provides a lucid understanding of how hyperparameters exert influence over the backward loss function—an arena where configurations resolve into distinct values.

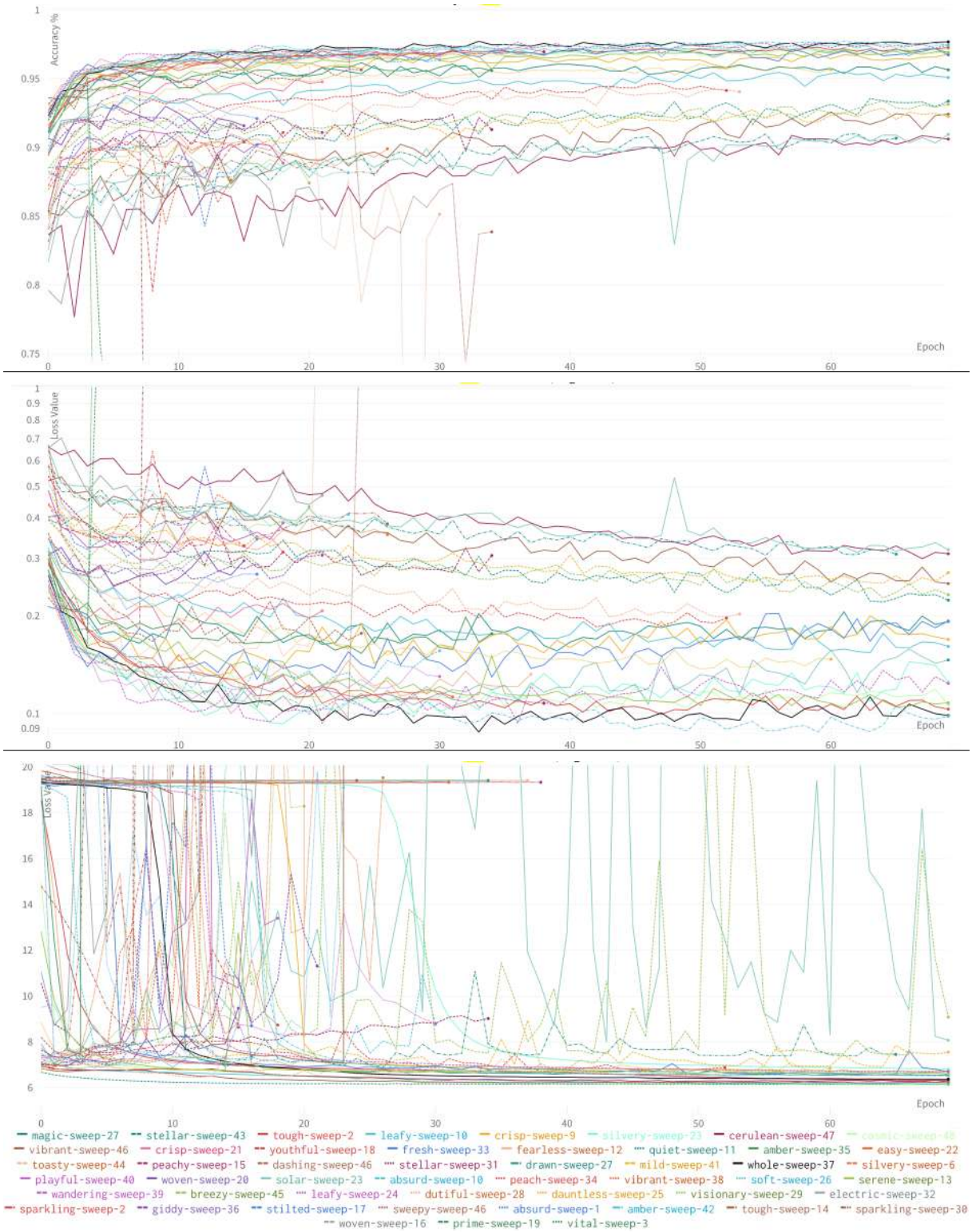


Figure 4.2: From top to bottom: Accuracy, forward loss, and backward loss plots of the undirected FFNN in the validation set. The name of each run is provided at the bottom.

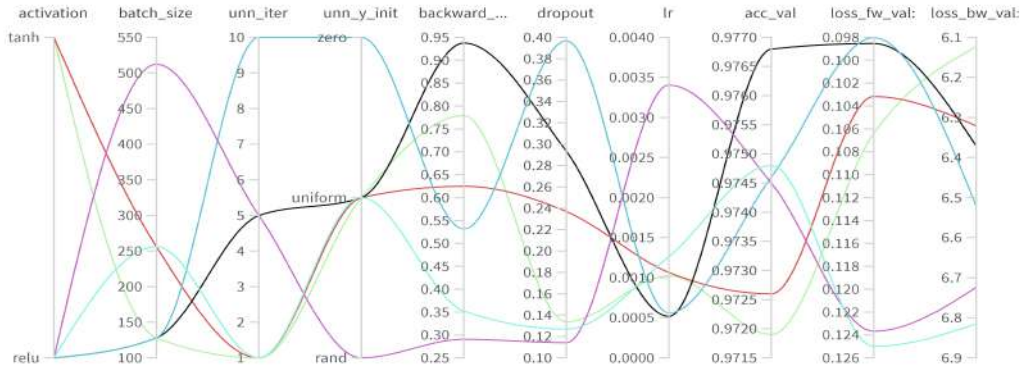


Figure 4.3: Parallel coordinates plot showing composition and final metrics of filtered undirected FFNN runs.

The previous Figure 4.3 unveils the presence of six runs that fulfil the established filtering criteria. Evidently, the model tends to exhibit enhanced performance with lower learning rate values. However, a definitive pattern delineating specific categories or optimal interval ranges for the remaining hyperparameters remains elusive. Among the array of plotted runs, the run denoted in black (labelled as *whole-sweep-37*) emerges as particularly noteworthy. This specific run not only attains the highest accuracy but also displays some of the lowest recorded forward and backward loss functions on the validation set. Given its exceptional performance, we have selected this specific run for testing purposes. The detailed configuration of this run can be seen in Table 4.2.

Table 4.2: Final configurations of the undirected FFNN: *whole-sweep-37*.

Hyperparameter	Values/Range
dropout	0.2933
learning_rate	0.0005
batch_size	128
unn_iter	5
unn_y_init	uniform
backward_loss_coef	0.9377
activation	relu
noise_sd	0

4.4.2 Stochastic and Undirected FFNN - Training Results

We adopted an approach similar to the one employed in sub-Section 4.4.1, entailing the execution of 50 randomized runs while utilizing the hyperparameters enumerated in Table 4.1. The initial observation drawn from these outcomes was the occurrence of more divergence behaviours in the runs obtained, upon the introduction of stochasticity. This divergence led to the activation of the early stopping mechanism, a concept we introduced above. This divergence was manifested in the accuracy plot, and in both loss functions plots seen in Figure 4.4.

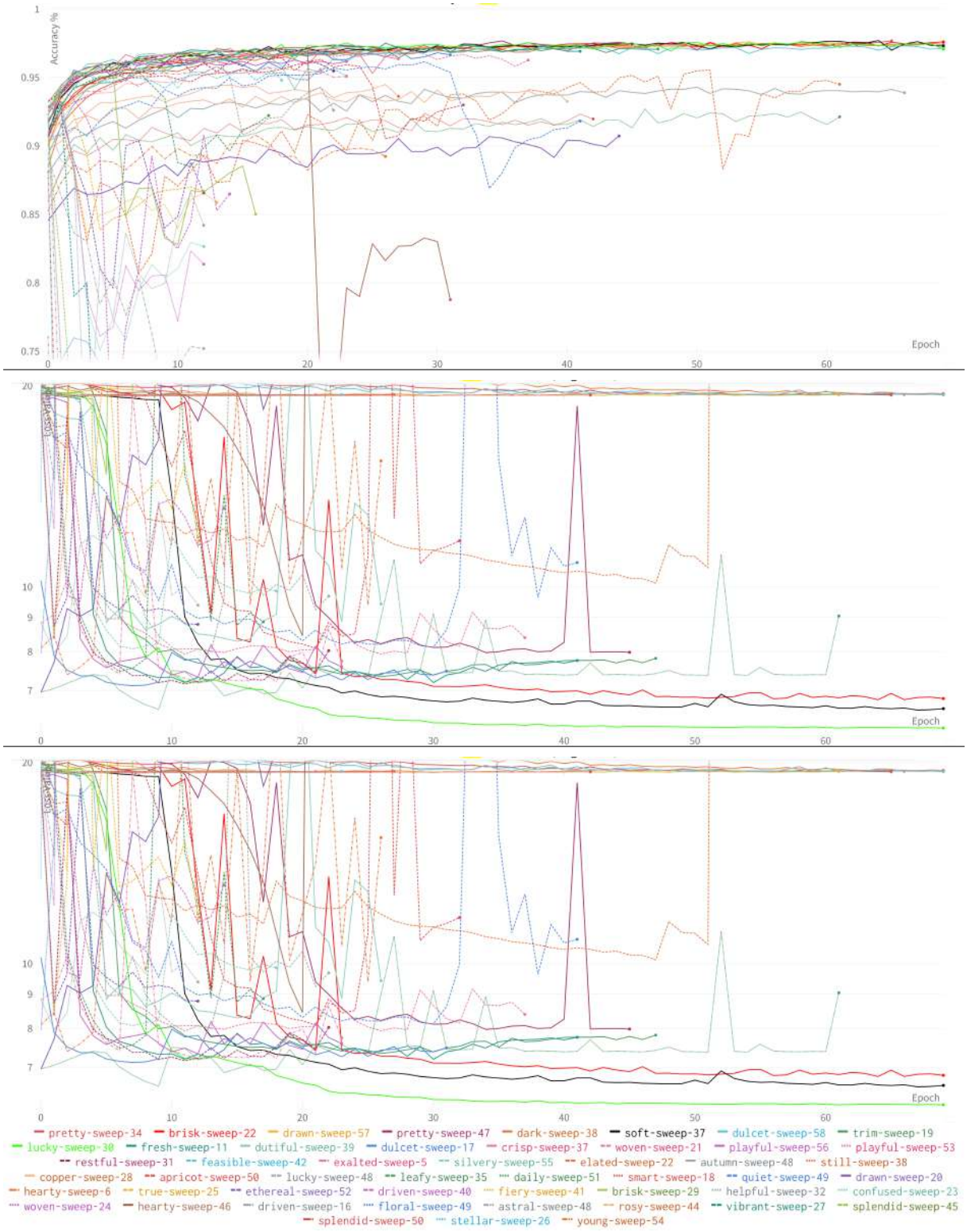


Figure 4.4: From top to bottom: Accuracy, forward loss, and backward loss plots of the stochastic and undirected FFNN in the validation set. The name of each run is provided at the bottom.

We noted that, for the backward loss, some runs converged to the same two different loss values already obtained in sub-Section 4.4.1; no new converged values were obtained. We also noted that only three runs converged into optimal loss and accuracy values that also satisfy the same filtration criteria applied in sub-Section 4.4.1. Figure4.5 shows these three configurations with their metrics.

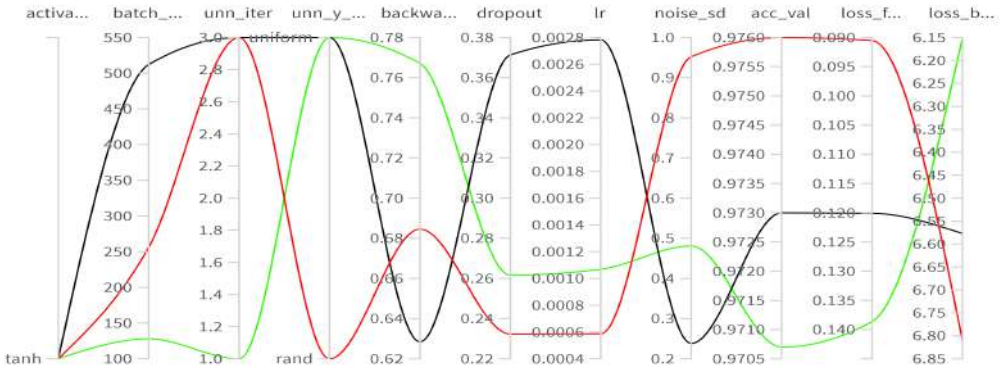


Figure 4.5: Parallel coordinates plot showing composition and final metrics of filtered stochastic and undirected FFNN runs.

The preceding plot illustrates that these three runs display some of the lowest forward values alongside high accuracies. Notably, in the presence of training stochasticity, these models tend to benefit from using the *tanh* activation function, lower *dropout*, *unn_init*, and *lr* values, as well as higher *backward_loss_coef* values. Additionally, selecting *tanh* as the activation function appears more favourable than using *relu*, and initialization different from zero for the *unn_y_init* parameter also seems advantageous. As for the *batch_size* and *noise_sd*, there appears to be no clear association. These conclusions are similar to the ones gathered in Sub-Section 4.4.1. We decided to use for testing *soft–sweep–37*, marked in black in Figure 4.5. This run is more balanced for both tasks, as it performs well in both, as opposed to the other two where one of the tasks performs better than the others. The final configuration can be seen below in Table 4.3.

Table 4.3: Final configuration of the stochastic and undirected FFNN: *soft–sweep–37*.

Hyperparameter	Values/Range
dropout	0.3714
learning_rate	0.0028
batch_size	512
unn_iter	3
unn_y_init	uniform
backward_loss_coef	0.7148
activation	tanh
noise_sd	0.2382

4.4.3 Assessment of Models with FFNN-type Operations on Forward Task

Upon successful training of the UNN and SUNN models, the subsequent phase involves assessing their performance on a designated test dataset. In addition, we introduce two BNNs models for the purpose of comparative analysis, particularly with respect to uncertainty quantification. These BNN models are trained using two distinct techniques: MC dropout and VI. Both BNN models share the same architectural configuration and hyperparameters, optimized specifically for classification tasks. In Table 4.4, we present the results of the misclassification experiment, detailed previously in Section 4.3.

Table 4.4: Results of FFNN-type models in misclassification experiment.

	AUROC (%)			**AUPR** (%)			**Accuracy %**
	Entropy	**MI**	**JSD**	**Entropy**	**MI**	**JSD**	
UNN	**96.15**	-	-	38.14	-	-	**97.92**
MC Droput	94.71	91.69	89.51	**43.25**	28.92	21.50	96.29
VI	94.66	94.52	90.65	40.44	37.13	17.04	96.42
SUNN	95.84	95.99	95.82	38.62	36.84	31.51	97.69

In the previous table, the rows represent different models: the first for the UNN, the second for the BNN trained with MC dropout, the third for the BNN trained with VI, and the last for the SUNN. The columns present scores for specific metrics in percentages. From left to right, they represent the AU of the ROC and PR curves, using metrics like entropy, MI, and JSD. The final column provides each model's accuracy. The rationale is the same for Table 4.8 and also for Tables 4.5 and 4.9, although the latter two do not have the accuracy column, as seen ahead. The results in the previous table reveal that the SUNN model demonstrates competitive performance when compared to the UNN model, particularly in terms of accuracy, with both outperforming the two BNN benchmark models. Notably, there exists a significant divergence between the AUROC and AUPR values for different uncertainty metrics. This discrepancy can be attributed to the pronounced class imbalance resulting from a low number of misclassifications for all models. Focusing first on the AUROC metric, it becomes evident that entropy plays a pivotal role in identifying positive class instances in this context, emerging as the more informative factor overall for all models, achieving AUROC values around 95%. It is noteworthy that for the SUNN model, the remaining metrics - MI and JSD, seem to be as informative as entropy. In sum, both the UNN and SUNN models exhibit superior performance in terms of the entropy metric compared to the two benchmark BNN models. Furthermore, the SUNN model also demonstrates higher values for the MI and JSD in comparison to the benchmark models. The AUPR results show, however, that the BNN trained using MC dropout exhibits a slight performance advantage over the other models, particularly when evaluated based on the entropy metric. It achieves an AUPR value of 43.25%, surpassing the SUNN model, which achieved a slightly lower value of 38.62%. To gain a more profound insight into the comprehension of AUROC and AUPR, we generate plots for both curves specifically for the SUNN

model, leveraging the MI metric as it exhibited the best performance among the metrics for this model. The plots can be seen below in Figure 4.6.

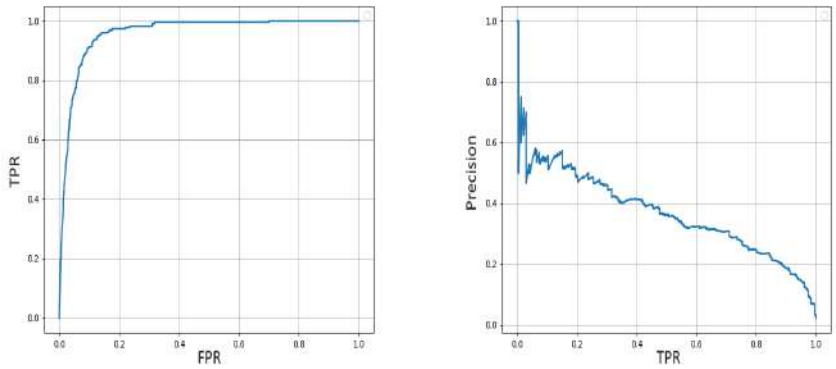


Figure 4.6: ROC (left) and PR (right) curves of the SUNN model with FFNN-type operations, in misclassification experiment.

Indeed, we can confirm the high AUROC of MI from the left plot. This is evident from the curve's proximity to the upper-left part of the graph, indicating a high TPR while maintaining a low FPR. Subsequently, as seen in the right plot, we observe that achieving both high precision and high recall is challenging. By considering a threshold of 5×10^{-4}, it becomes apparent that a significant number of positive samples can be detected for MI values exceeding this threshold, as seen below in Figure 4.7.

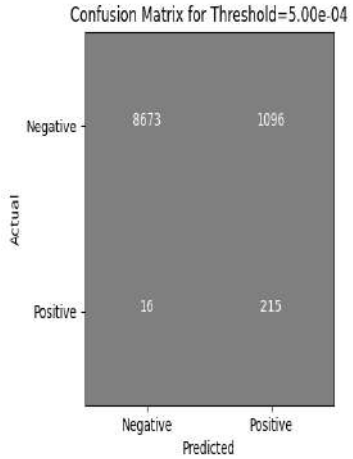


Figure 4.7: Confusion matrix of the SUNN model with FFNN-type operations, using mutual information with a 5×10^{-4} threshold, in misclassification experiment.

Based on the information provided in the previous confusion matrix, we can compute the following approximate metrics: a TPR or Recall of 91%, a FPR of 11%, and a precision of 19%, all under the influence of the aforementioned threshold. With this particular threshold, only 16 positive cases were

not detected. This extra protection system can be particularly advantageous in situations where the cost associated with false positives is not prohibitive, since there are some negative samples being predicted as positive (around 11%). Any new sample with a predicted MI higher than this threshold can trigger an automatic classification system, which can then undergo further evaluation by human intervention. Transitioning to the OOD detection experiment, also detailed previously in Section 4.3, the results are summarized in Table 4.5.

Table 4.5: Results of FFNN-type models in OOD detection experiment.

	AUROC (%)			**AUPR** (%)		
	Entropy	**MI**	**JSD**	**Entropy**	**MI**	**JSD**
UNN	**78.23**	-	-	**77.23**	-	-
MC Droput	77.99	76.90	74.32	76.59	75.34	71.96
VI	74.63	75.60	72.07	73.87	75.84	67.57
SUNN	76.27	76.53	76.37	74.99	75.61	74.85

It is noteworthy that there is a minimal discrepancy in the values recorded for AUROC and AUPR. This observation is attributed to the experiment's perfectly balanced scenario, featuring an equal number of in domain (ID) and OOD samples. Here we note that the UNN slightly outperforms the SUNN in maximizing the AUROC and the AUPR, especially when considering the entropy value, presenting with this metric AUROC and AUPR values around 78% and 77%, respectively. The SUNN exhibits, respectively, 76% and 75%. The BNN trained with MC dropout also slightly outperforms the SUNN, but again the difference is minimal. The worst performer is the BNN trained with VI. Overall, we see that the differences are not too significant among all models and uncertainty measures. Both for the AUROC and the AUPR, we see that entropy and MI are overall the best metrics for all models when it comes to detecting outliers. For the SUNN, the MI is slightly the better metric. We present both curves in Figure 4.8.

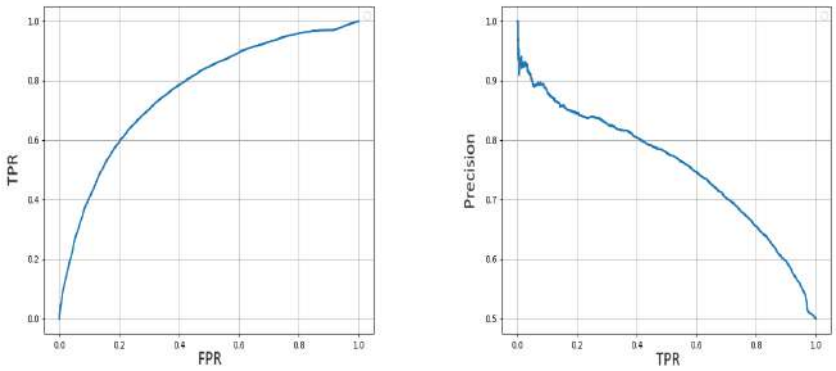


Figure 4.8: ROC (left) and PR (right) curves of the SUNN model with FFNN-type operations, in OOD detection experiment.

We note that the SUNN model is somewhat relatively balanced for detecting both ID and OOD samples. We see in both plots that for a fixed TPR or (recall), we can have a low FPR and high precision, at the same time. See below in Figure 4.9 a confusion matrix for a pertinent MI thresholds of 5×10^{-6}.

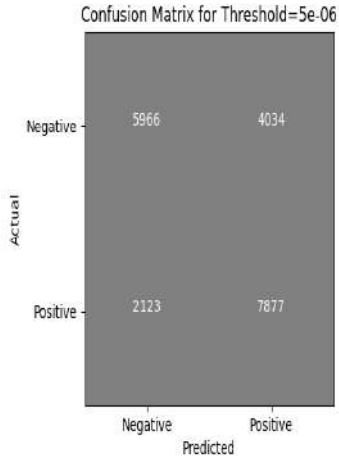


Figure 4.9: Confusion Matrix of the SUNN model with FFNN-type operations, using MI with a 5×10^{-6} threshold, in OOD detection experiment.

Using this particular threshold we are able to better separate OOD from ID samples, although not perfectly and achieving, approximately, a TPR of 79%, a FPR of 40% and a precision of 66%. Although having a bit more than one-fifth of the OOD samples identified as ID, it is essential to note that distinguishing these specific OOD samples from ID samples can pose a challenge for the models in question. This challenge may arise because these OOD samples retain digits in the background, potentially causing confusion to the models. At the same time, this problem may arise because models with FFNN-type operations are not very suitable for handling image data.

Overall, this section shows the potential and competitiveness of the SUNN model against other benchmarks. In terms of accuracy, it is superior to the classical methods, although slightly outperformed by the UNN. It exhibited good performance both in the misclassification experiment and OOD detection experiment, although slightly outperformed by the classical approaches. The choice of the uncertainty metric and threshold plays a crucial role in achieving optimal performance, and all the other model exhibits versatility in this regard, as well as the other models.

4.4.4 Examination of Backward Predictions of Models with FFNN-type Operations

In the upcoming subsection, we delve into the generative capabilities of the UNN and SUNN models with FFNN-type operations, discussed in the preceding sub-sections, with a specific focus on the stochastic

model. Our objective is to evaluate the numerical characters generated by these models. As previously discussed in sub-Section 2.2.2, the Bernoulli restricted BM can undergo training to produce data, such as digit patterns, even when initiated from nonsensical representations like pixelated images. In this context, our starting point involves an output label—an integer ranging from 0 to 9. We then proceed to propagate computations from this particular layer to the initial layer, which mirrors the dimensions of a typical *MNIST* image. Starting by the UNN with FFNN-type operations, its generated prototypes can be seen in Figure 4.10. Note that all images were created by first applying the *softmax* function over the *logit* values of the input layer.

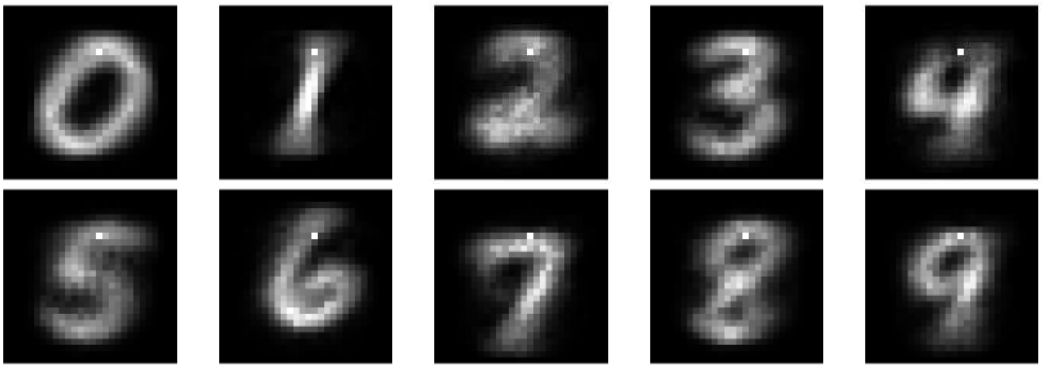


Figure 4.10: Digits generated by the UNN model with FFNN-type operations. From left to right, top to bottom, all the digits from 0 to 9.

The digits generated by this UNN are, in fact, representative of the class from where they are coming from, since they are realistic and easy to identify by the human eye. Running the network again would result in the same generations since it is a deterministic model. Moving on to the SUNN model, we decided to generate four prototypes of the same class. However, each prototype has an average of three backward passes in the network. This process was done because the network can output *logit* values with different ranges. The saturation of the *sigmoid* function at extreme input values, causes the outputs to have many pixels with values very close either to 0 or 1, leading to poor image quality. By averaging over three images, this effect is attenuated. In Figure 4.11 we present some generations of the digits of the first three classes. The generated digits of the remaining classes can be seen in Figure A.1, Appendix A. This average process was also done for the CNN version, which will be presented further ahead.

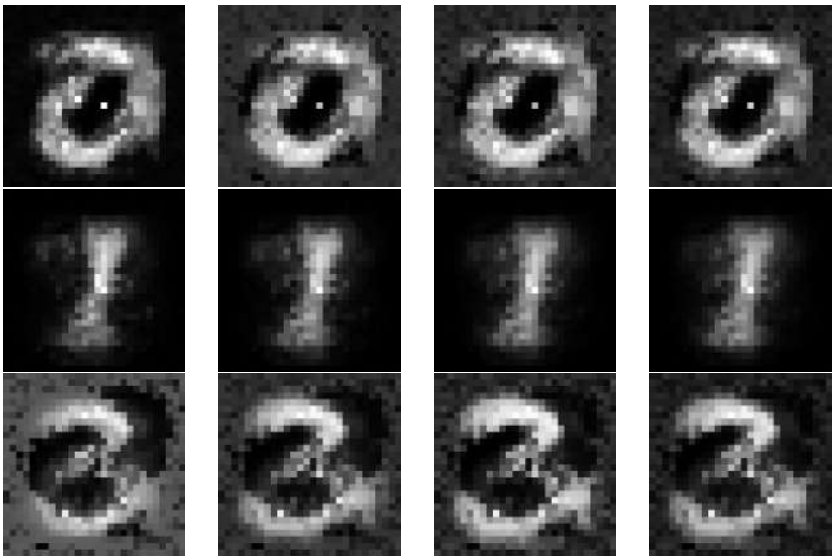


Figure 4.11: Digits generated by the SUNN model with FFNN-type operations. From top row to bottom row: classes 0 to 2.

In the examination of the prototypes generated by this model, we employed the same noise and iteration count as those used during training, as this presented itself as the most logical choice. An observable characteristic of these digits is their grainy texture, and some even exhibit blurred backgrounds. The diversity of the prototypes is somewhat limited; a discernible similarity is evident across different classes. Intriguingly, the prototype of class 2 bears a strong resemblance to that of class 3. This anomaly can likely be attributed to the inherent limitations of FFNN. Specifically, FFNN are generally less equipped to process spatial hierarchies and intricate patterns in images, especially in comparison to their CNN counterparts. The convolutional operations intrinsic to CNNs make them inherently more apt for such tasks. As we delve into subsequent sections focusing on the CNN rendition of the SUNN, this observation will serve as a reference point, hinting at the potential superior efficacy of CNNs in addressing these challenges. Increasing the standard deviation of the noise term did not lead to improvements in diversity across all classes, but only to more blurred digits.

4.5 Models with CNN-type Operations

When dealing with images, models of the CNN type prove to be more suitable. These specific models surpass FFNNs in image-related tasks, a well-established fact in the field [45]. This superiority is attributed to distinct characteristics inherent to CNNs, such as local connectivity, parameter sharing, hierarchical feature extraction, pooling and convolutional layers, among others (see Chapter 9 in [4]). The architecture of CNNs had already been implemented and made open-source in the UNN paper [1]. Therefore, we adapted this architecture to introduce stochasticity. The corresponding FG for this config-

uration is depicted in Figure 3.1 (c). It is common not to consider a bias term b_X in the input layer, and therefore we defined $b_X = 0$. As for the activation functions, it is common to use some of the well-known ones found in Table 2.1. The total energy function is, in this case:

$$\begin{aligned} E(x, y, h_1, h_2, n_{H_1}, n_{H_2}, n_Y) = E_X(x) + E_{H_1}(h_1) + E_{H_2}(h_2) + E_Y(y) + E_{XH_1}(x, h_1) + E_{H_1H_2}(h_1, h_2) \\ + E_{H_2Y}(h_2, y) + E_{H_1N_{H_1}}(h_1, n_{H_1}) + E_{H_2N_{H_2}}(h_2, n_{H_2}) + E_{YN_Y}(y, n_Y), \end{aligned} \tag{4.7}$$

where

$$\begin{aligned} E_X(x) &= -\langle x, b_X \otimes 1_{d_X} \rangle + \Psi_X(x) = \Psi_X(x), \quad b_X = 0, \\ E_{H_1}(h_1) &= -\langle h_1, b_{H_1} \otimes 1_{d_{H_1}} \rangle + \Psi_{H_1}(h_1), \\ E_{H_2}(h_2) &= -\langle h_2, b_{H_2} \otimes 1_{d_{H_2}} \rangle + \Psi_{H_2}(h_2), \\ E_Y(y) &= -\langle b_Y, y \rangle + \Psi_Y(y) \\ E_{XH_1}(x, h_1) &= -\langle h_1, C_1(x, W_1) \rangle, \\ E_{H_1H_2}(h_1, h_2) &= -\langle h_2, C_1(h_1, W_2) \rangle, \\ E_{H_2Y}(h_2, y) &= -\langle y, Vh_2 \rangle, \\ E_{H_1N_{H_1}}(h_1, n_{H_1}) &= -\langle h_1, n_{H_1} \otimes 1_{d_{H_1}} \rangle, \quad n_{H_1} \sim N(0, \sigma^2 I_{d_{H_1}}),\ \sigma \text{ known}, \\ E_{H_2N_{H_2}}(h_2, n_{H_2}) &= -\langle h_2, n_{H_2} \otimes 1_{d_{H_2}} \rangle, \quad n_{H_2} \sim N(0, \sigma^2 I_{d_{H_2}}), \\ E_{YN_Y}(y, n_Y) &= -\langle n_Y, y \rangle, \quad n_Y \sim N(0, \sigma^2 I_{d_Y}). \end{aligned} \tag{4.8}$$

Using the update rule from Equation (3.6), the update terms, denoted by x_*, h_{1*}, h_{2*} and y_* are given by:

$$\begin{aligned} x_* &= (\nabla \Psi_X^*) \left(C_1^T(x, W_1) \right) \\ h_{1*} &= \left(\nabla \Psi_{H_1}^* \right) \left(C_1(x, W_1) + C_2^T(h_2, W_2) + (b_{H_1} + n_{H_1}) \otimes 1_{d_{H_1}} \right) \\ h_{2*} &= \left(\nabla \Psi_{H_2}^* \right) \left(C_2(h_1, W_2) + \sigma_y(V) y + (b_{H_2} + n_{H_2}) \otimes 1_{d_{H_2}} \right) \\ y_* &= (\nabla \Psi_Y^*) \left(Vh_2 + (b_Y + n_Y) \right) \end{aligned} \tag{4.9}$$

In this experience, each input image x is treated as a grey-scale image, and therefore $x \in \mathbb{R}^{1 \times 28 \times 28}$. We decided to keep the same shapes of the weights and biases terms as in the convolutional UNN experiment done in [1], therefore we have: $W_1 \in \mathbb{R}^{32 \times 1 \times 6 \times 6}$, $W_2 \in \mathbb{R}^{64 \times 32 \times 4 \times 4}$, $V \in \mathbb{R}^{10 \times 64 \times 5 \times 5}$,

$b_{H_1} \in \mathbb{R}^{32}$, $b_{H_2} \in \mathbb{R}^{64}$, $b_Y \in \mathbb{R}^{10}$. In consequence, $h_1 \in \mathbb{R}^{32\times12\times12}$ and $h_2 \in \mathbb{R}^{64\times5\times5}$. The σ_y operator rolls the axis of V corresponding to y to the last position, such that $\sigma_y(V) \in \mathbb{R}^{64\times5\times5\times10}$.

4.5.1 Undirected CNN - Training Results

Similarly to what we did in sub-Section 4.4.1, we ran 50 different runs with hyperparameters being uniformly sampled from the intervals or categories seen in Table 4.1. The evolution of the accuracy, forward and backward losses in the validation set can be seen in Figure 4.12.

As expected, we observe higher accuracy values and lower values of the loss functions, as a consequence of using CNN-type operations instead of the ones used by FFNNs. We applied stricter filtering to the results due to the superior performance of the CNNs in these particular tasks. Subsequently, we chose to visualize only the runs meeting the following criteria: final forward loss on the validation set less than or equal to 0.2, validation set accuracy greater than or equal to 98%, and validation set backward loss less than or equal to 7. There were in fact 13 runs meeting these restrictions. Figure 4.13 presents these qualifying runs in a Parallel Coordinates plot.

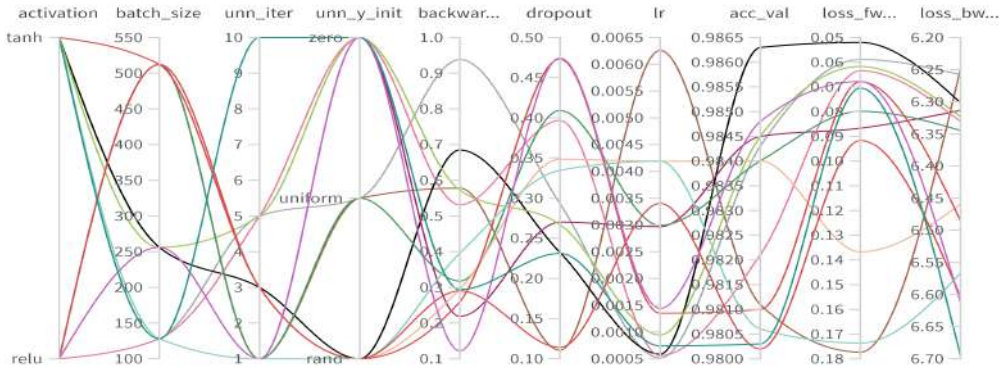


Figure 4.13: Parallel coordinates plot showing composition and final metrics of filtered undirected CNN runs.

Among the array of plotted runs, we selected for testing purposes the one represented in black (labelled as *sweet-sweep-53*) as it shows the highest accuracy and lowest forward loss in the validation set, as well as one of the lower backward loss function values in the same set. The detailed configuration of this run can be seen in Table 4.6.

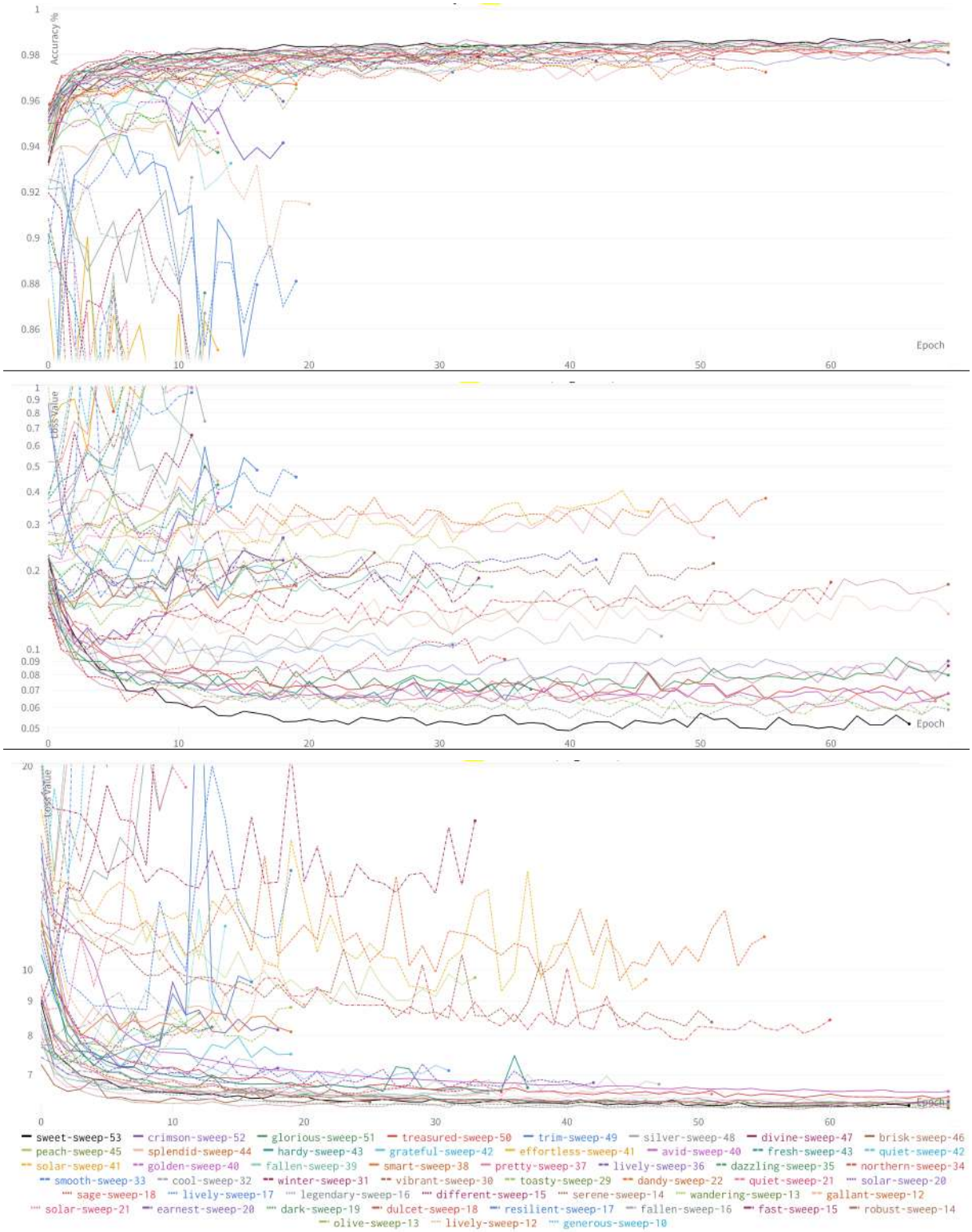


Figure 4.12: From top to bottom: Accuracy, forward loss, and backward loss plots of the undirected CNN in the validation set. The name of each run is provided at the bottom.

Table 4.6: Final configuration of the undirected CNN: *sweet-sweep-53*.

Hyperparameter	Values/Range
dropout	0.2323
learning_rate	0.0006
batch_size	256
unn_iter	3
unn_y_init	rand
backward_loss_coef	0.6845
activation	tanh
noise_sd	0

It is important to highlight that this UNN incorporates an element of randomness: the random initialization of its final layer. Based on our experimentation, we observed minor distinctions when employing various initializations. To eliminate this stochastic effect, we adopted a uniform initialization during testing, ensuring deterministic predictions. Alternatively, for UNN models employing random y-initialization, an effective strategy could involve retaining the randomness while averaging the *logits* across a substantial number of predictions. We determined that utilizing 50 samples maintained relatively consistent outcomes.

4.5.2 Stochastic and Undirected CNN - Training Results

Using stochasticity in training promoted more runs to diverge completely, which stopped their training process consequentially, as seen in the accuracy and loss plots shown in Figure 4.14.

This was something that we already saw happen with the FFNN-models in Chapter 4.4.2. Only five runs met the previous criteria we applied, with their configurations and metrics seen below in Figure 4.15.

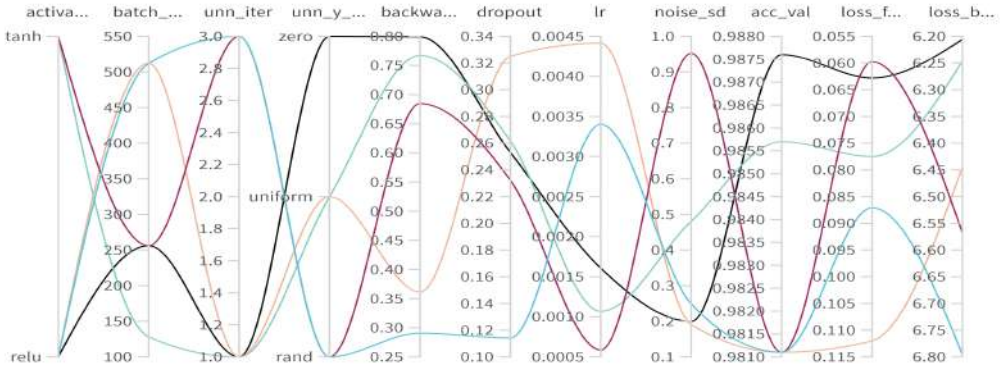


Figure 4.15: Parallel coordinates plot showing composition and final metrics of filtered stochastic and undirected CNN runs.

We selected for testing the one marked in black, named *fersh-sweep-53* since it was overall better

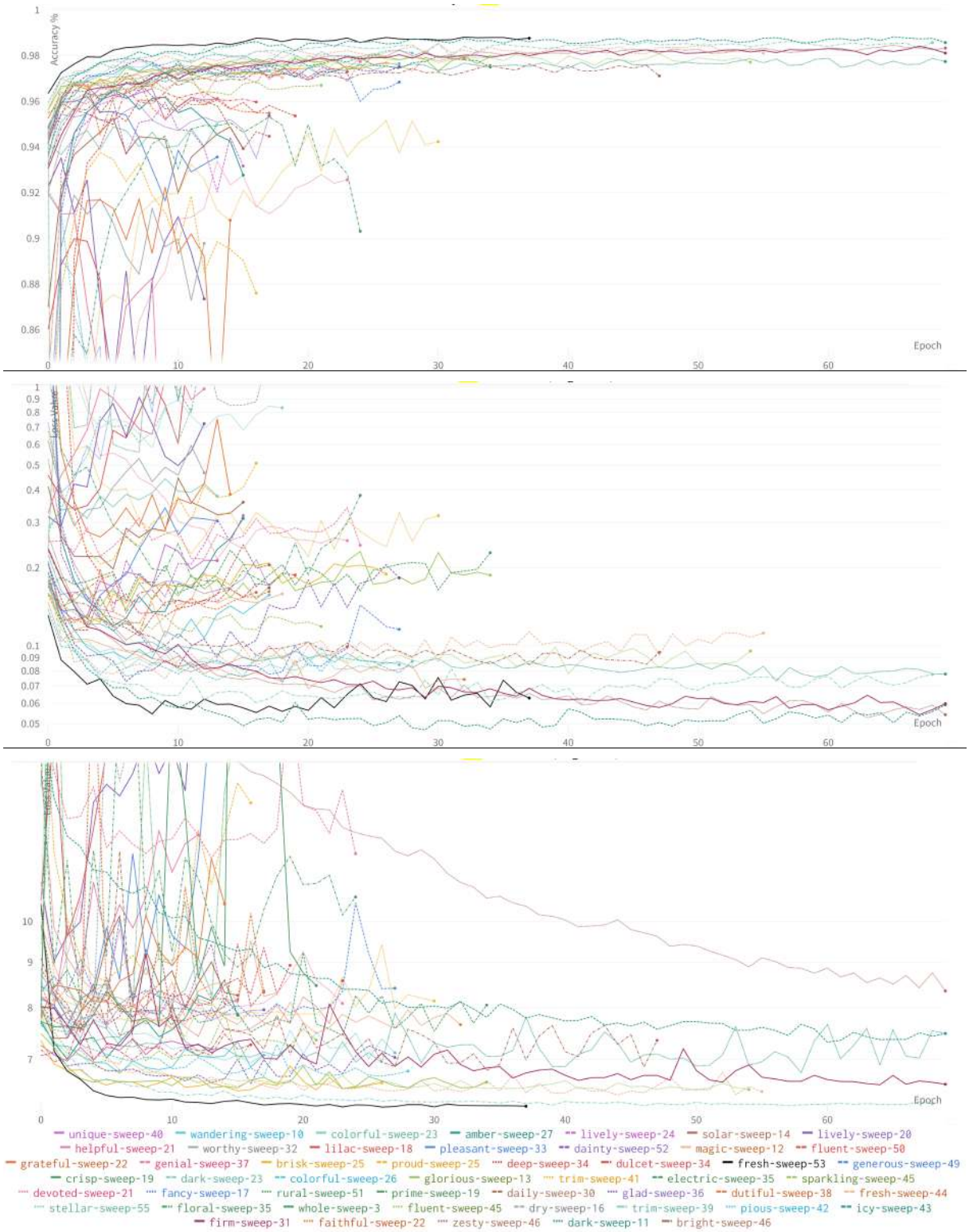


Figure 4.14: From top to bottom: Accuracy, forward loss, and backward loss plots of the stochastic and undirected CNN in the validation set. The name of each run is provided at the bottom.

than the other runs because it had the lowest backward loss values, second lowest forward loss value and highest accuracy in the validation set. Its configuration can be seen below in Table 4.7. In [1], it was shown that the random initialization of the output layer had promising results, particularly in terms of accuracy. However, here we can see that there is an interaction effect between hyperparameters.

Table 4.7: Final configurations of the stochastic and undirected CNN: *fersh-sweep-53*.

Hyperparameter	Values/Range
dropout	0.2541
learning_rate	0.0016
batch_size	256
unn_iter	1
unn_y_init	zero
backward_loss_coef	0.7992
activation	relu
noise_sd	0.1994

4.5.3 Assessment of Models with CNN-type Operations on Forward Task

In this section, we delve into the evaluation of the CNN-type UNN and SUNN on the test set, alongside with the BNN benchmark models, featuring a comparable architecture in both experiments. The respective outcomes of the first experiment are displayed below in Table 4.8.

Table 4.8: Results of CNN-type models in misclassification experiment.

	AUROC (%)			**AUPR** (%)			**Accuracy %**
	Entropy	**MI**	**JSD**	**Entropy**	**MI**	**JSD**	
UNN	**98.14**	-	-	36.79	-	-	98.87
MC Droput	97.89	97.69	97.21	**38.36**	33.72	31.28	99.01
VI	97.76	97.74	97.69	35.05	29.00	26.62	**99.18**
SUNN	97.82	97.84	96.89	34.08	35.27	16.27	98.95

The first thing to notice is a substantial increase in the AUROC values, which consistently approach nearly 100%. This stands in stark contrast to the AUROC values recorded for FFNN-type models, as presented in Table 4.4. Across all models under consideration, it becomes evident that both entropy and MI exhibit a pronounced association with misclassification, consistently yielding values approaching approximately 98%. It is worth noting that the application of hyperparameter optimization had a more substantial impact on BNNs, resulting in test set accuracies surpassing the 99% threshold. In comparison, the UNN and SUNN achieved test accuracies of nearly 99%, with the difference between them being relatively minor. Shifting our focus to the area under the AUPR values, one particular model stands out slightly in comparison to our SUNN. Specifically, the BNN trained using MC dropout attains an AUPR of approximately 38% when employing entropy as the criterion. In contrast, our SUNN achieves an AUPR value of approximately 35% when considering MI. Below, we present in Figure 4.16 both curves for the

SUNN model.

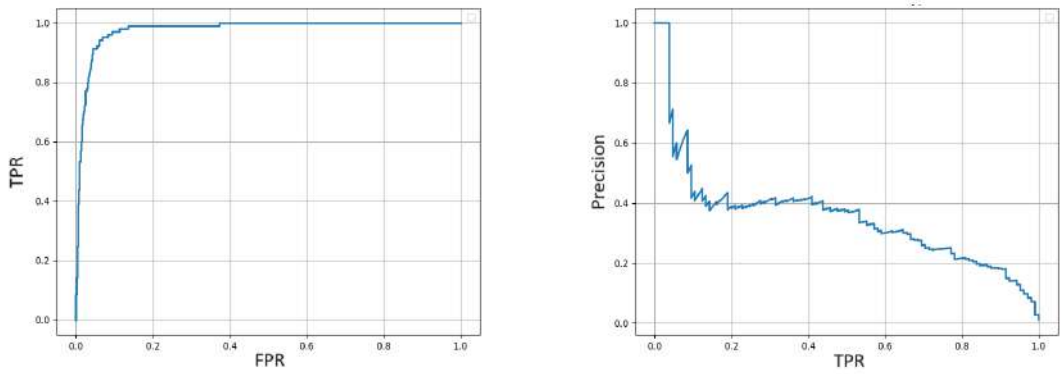


Figure 4.16: ROC (left) and PR (right) curves of the SUNN model with CNN-type operations, in misclassification experiment.

We can confirm that, in fact, we have a better ROC curve, but a slightly worse PR recall curve. Most likely, it will be hard to find a MI threshold to adequately separate positive from negative predictions. If we select 5×10^{-5}, we obtain the confusion matrix seen below in Figure 4.17.

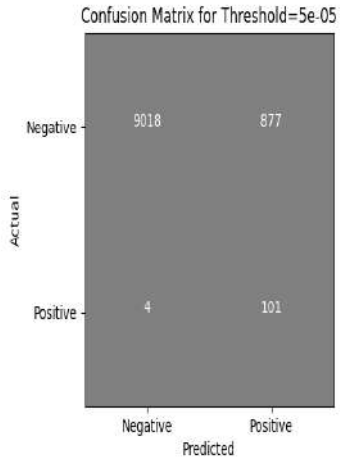


Figure 4.17: Confusion Matrix of the SUNN model with CNN-type operations, using MI with a 5×10^{-5} threshold in misclassification experiment.

We note that with this particular threshold, only 4 positive samples are classified as negative. As in the misclassification experiment of FFNN-type models, whose results were presented in Figure 4.7, there is still a large number of negative samples being classified as positive. Once again, this will not be problematic in real-life applications if the cost of having false alarms is low. In sum, this model achieves, approximately, a TPR of 96%, a FPR of 9% and a precision of 10%. Compared with the FFNN version, there is a decrease in precision, however, it is compensated by an increase in TPR and a decrease in

FPR.

As for the OOD detection experiment using CNN-type models, the results are presented below in Table 4.9.

Table 4.9: Results of CNN-type models in OOD detection experiment.

	AUROC (%)			**AUPR** (%)		
	Entropy	**MI**	**JSD**	**Entropy**	**MI**	**JSD**
UNN	86.13	-	-	85.24	-	-
MC Droput	97.03	96.11	95.36	96.65	94.24	93.43
VI	**98.43**	97.15	97.10	**98.21**	93.80	93.62
SUNN	92.26	92.09	91.15	91.00	90.86	86.93

Based on the previous case results, the BNN models notably outperformed both the UNN and the SUNN, indicating a significant performance difference. While the SUNN model proved better at uncertainty quantification than the UNN, it still did not match the performance of the other models. Regarding the AUROC, the BNN trained under VI obtained a value around 98%, while the SUNN only obtained a value around 92%, when considering the entropy metric. As for the AUPR, the BNN trained underVI obtained a similar value of about 98%, while the SUNN only obtained a value of 91%, again considering the entropy metric. In Figure 4.18, we present both curves for the SUNN model.

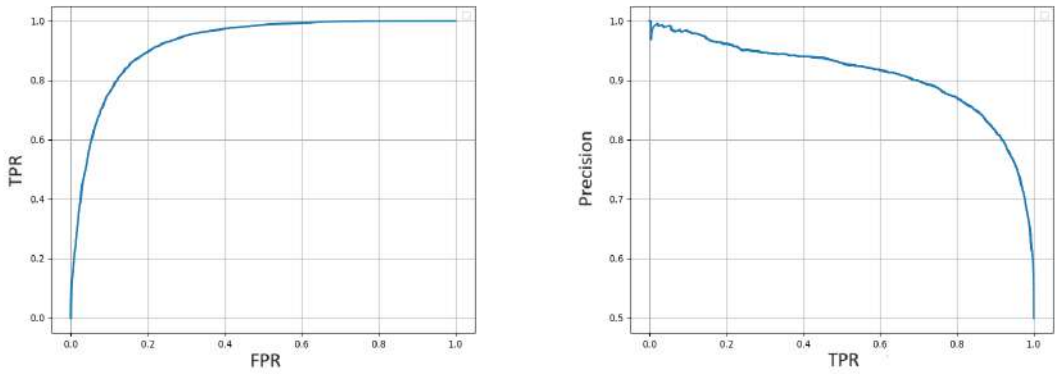


Figure 4.18: ROC (left) and PR (right) curves of the SUNN model with CNN-type operations, in OOD detection experiment.

We note that the ROC and PR curves are indeed better in the SUNN model with CNN-type operations than in the SUNN model with FFNN-type operations, whose results are shown in Figure 4.9. If we build a confusion matrix with an entropy threshold of 5×10^{-4}, we obtain the one in Figure 4.19.

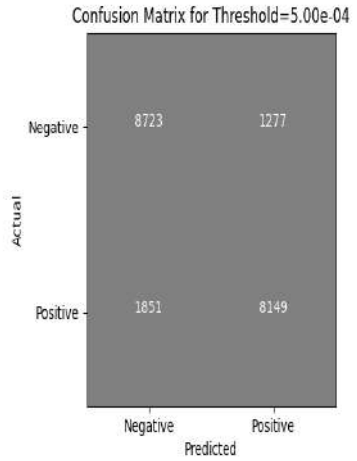


Figure 4.19: Confusion Matrix of the SUNN model with CNN-type operations, using entropy with a 5×10^{-4} threshold in OOD detection experiment.

Again, we note an improvement with respect to the SUNN of the aforementioned scenario of the FFNN-type models. There are many less misclassification, which translates to the improved ratio metrics. This model achieves, approximately, a TPR of 81%, a FPR of 13% and a precision of 86%. It is worth noticing that for the BNN trained under VI and with the same entropy threshold, we obtain the confusion matrix seen below in Figure 4.20.

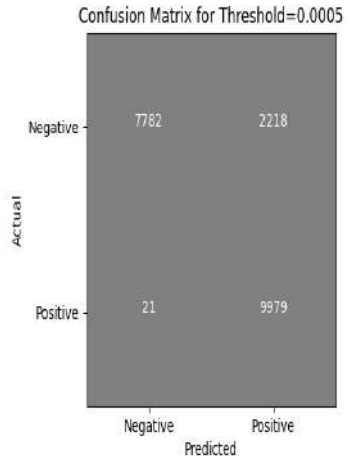


Figure 4.20: Confusion Matrix of the BNN-VI model using entropy with a 5×10^{-4} threshold in OOD detection experiment.

There is notably a more pronounced association of entropy with the identification of OOD samples, which translate, approximately, in a TPR of 100%, a FPR of 22% and a precision of 82%. We must not forget that although there is a more pronounced performance gap between the SUNN and this particular benchmark model, the SUNN has the capacity to generate samples, something that the benchmarks

can not do.

We conclude this section by highlighting that the utilization of CNN-type operations enhances the performance of the SUNN model in both experiments. It achieves higher AUROC and AUPR values compared to those discussed in sub-Section 4.4.3. However, it's worth noting that the performance gap between the SUNN model and the benchmark models has increased slightly, with the benchmarks outperforming the proposed model by a few percentage points. Nevertheless, it's important to emphasize that the proposed model possesses generative capabilities, a unique feature not present in the benchmark models, which will be the focal point of discussion in the upcoming sub-section.

4.5.4 Examination of Backward Predictions of Models with CNN-type Operations

The generated prototypes obtained with the UNN with CNN-type operations can be seen below in Figure 4.21.

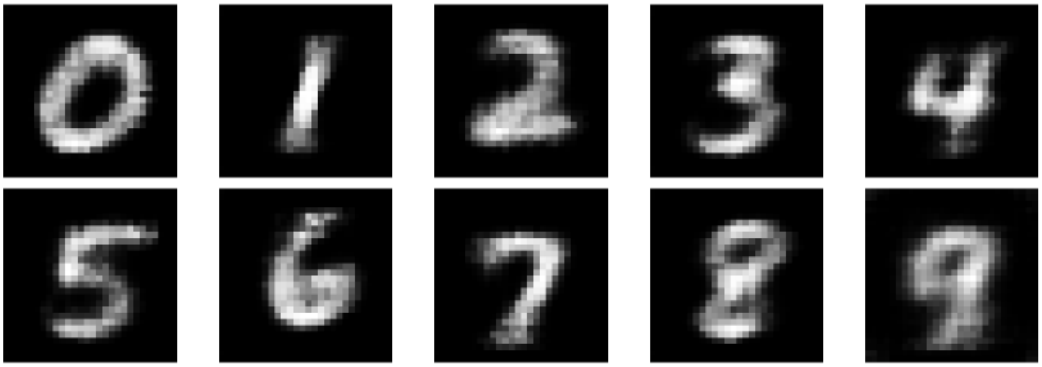


Figure 4.21: Digits generated by the UNN model with CNN-type operations. From left to right, top to bottom, all the digits from 0 to 9.

It is observed that the prototypes produced by the UNN with CNN-type operations are not markedly different from those generated using FFNN-type operations. Each generated digit accurately reflects its respective class. The prototypes from the SUNN variant are presented in Figure 4.22. For this specific SUNN, noise set at 1.5 times the standard deviation was employed, since utilizing noise equivalent to the training conditions would result in diminished diversity in the produced images. The generated digits for the first three classes can be seen below, and the remaining ones can be found in Table A.2, Appendix A.

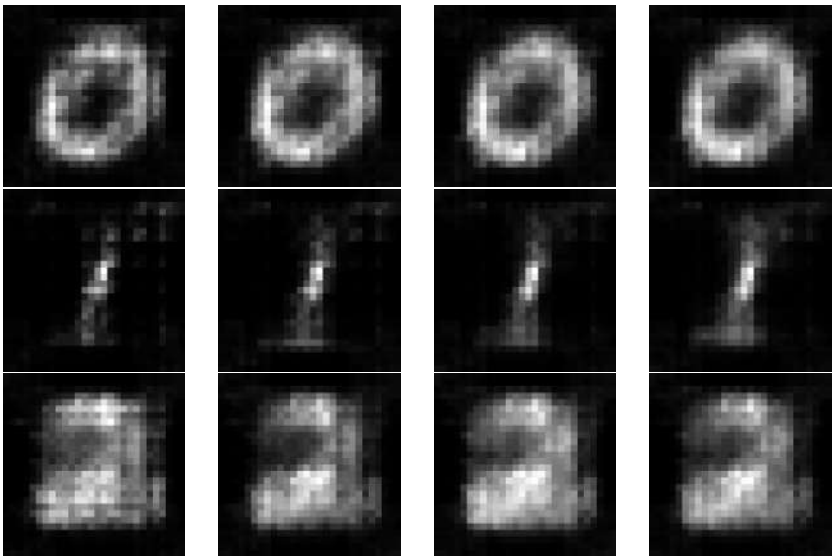


Figure 4.22: Digits generated by the SUNN model with CNN-type operations. From top row to bottom row: classes 0 to 2.

It is immediately evident that the images are considerably crisper than those generated by the FFNN versions. There is a noticeable uptick in diversity, especially in class 1, when we look at the digits from the first and last column. While the first digit has a standard or regular thickness throughout, the last one is characterized by pronounced or increased thickness at both the top and bottom parts, making it appear bolder or more robust in these regions compared to the first digit. While the other two classes also display some diversity, it predominantly manifests in pixel intensity rather than digit shape. A more detailed analysis and comparison of the two SUNN models can be found in Appendix A, highlighting a marginal advantage in utilizing SUNN. Upon visual assessment, the quality of these digits is on par with the prototypes produced by the preceding restricted BM. Even though the latter is a benchmark model previously refined through research, the SUNN model appears promising for this application. It is pivotal to remember that this model not only functions as a classifier but also gauges the uncertainty in its predictions, aligning well with established benchmark models for this specific task.

5

Conclusion

Contents

5.1 Conclusions

We introduced a model, termed the Stochastic Undirected Neural Network (SUNN), which possesses intrinsic undirected and stochastic characteristics. This structure allows the SUNN to perform various tasks, including uncertainty estimation and prototype generation. When incorporating FFNN-type operations, the SUNN showcased competitive uncertainty estimation against other benchmarks, proving particularly effective in identifying both misclassification and OOD samples. However, when integrated with CNN operations, its performance lagged behind the benchmarks. On the prototype generation front, the SUNN was adept at creating digits that accurately mirrored their classes. This capability was especially pronounced with CNN-type operations, resulting in a diverse set of outputs. Conversely, the quality and diversity of outputs diminished when the SUNN utilized FFNN-type operations.

5.2 Future Work

This study has established both a theoretical and practical groundwork for the SUNN model, thereby paving the way for further research endeavours aimed at gaining a deeper comprehension of the principles and applications raised by this innovative framework. Throughout this research work, various potential research directions have emerged, which will be mentioned below.

An intriguing avenue for exploration involves extending this framework to architectures utilized in diverse domains. For instance, in [1], the UNN model was effectively applied to the natural language processing (NLP) field, in particular to language datasets for **dependency parsing tasks**. Consequently, the prospect of designing a SUNN framework tailored to a task like this one holds considerable promise. In fact, the integration of confidence metrics within parse trees becomes pivotal, as these metrics serve as a means to gauge whether the model has effectively captured the nuances of different linguistic characteristics and syntactic structures within a given language. The same holds for other NLP tasks like **language translation**, where a SUNN model has the ability to translate between two languages and also to output some uncertainty measures regarding its predictions. In translation tasks, transformer-based models are a popular model to use, and recently, **energy-based transformers** [46] have garnered interest, proving to be competitive with other state-of-art approaches in graph anomaly detection tasks. The incorporation of stochasticity through an approach reminiscent of the reparametrization trick, may be worth investigating. Exploring this topic further by establishing a solid theoretical foundation for this model and benchmarking against other models could be valuable.

On a different note, future research work could be done with respect to the **architecture design** of the SUNN. In this work, we employed quite a few restrictions over the factor graph, such as having a number of stochastic nodes that is dependent on the architecture defined, as well as having normal Gaussian noise. Relaxing some of these assumptions and laying a theoretical understanding of different

setups, as well as their implications could also be an interesting avenue to pursue.

In this work, and as well as in [1], both the UNN and the SUNN models have an underlying energy function which, when its terms are updated, allows the information to flow in any directions and with many fewer restrictions as in usual NN models. In fact, the **energy function** is used as a proxy to do such undirected computations that introduce randomness in the network. In this work, we did not explore what were the consequences and performance impacts of performing more or less energy minimization steps than the ones employed during training. Intuitively, the energy minimization steps should allow for obtaining better representations, but how many should be made? And regarding image generation, are more energy updates beneficial?

In the context of this research and as demonstrated in [1], both the UNN and the SUNN models incorporate an intrinsic energy function. This energy function, when its constituent terms are updated, facilitates the unconstrained flow of information in multiple directions, in stark contrast to conventional NN models. Notably, the energy function serves as a surrogate for carrying out undirected computations, thereby introducing an element of randomness into the network's operations. However, it is worth noting that in our work, we did not delve into the implications and changes in performance of varying the number of energy minimization steps beyond those employed during the training process. While it is intuitive to assume that a greater number of energy minimization steps could lead to the acquisition of more refined representations, the question arises: How many such steps should ideally be undertaken? Furthermore, when it comes to image generation tasks, can an increased number of energy updates prove to be advantageous to obtain sharper and more diverse images?

In closing, this research has set the stage for a new chapter in the exploration of the SUNN model and its implications. We hope these questions, and many others that can arise, inspire further investigations, propelling the field of machine learning into exciting and uncharted territories.

Bibliography

[1] T. Mihaylova, V. Niculae, and A. F. T. Martins, "Modeling structure with undirected neural networks," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2022.

[2] D. P. Kingma, "Variational inference & deep learning: A new synthesis," PhD thesis, Faculty of Science (FNWI), Informatics Institute (IVI), University of Amsterdam, 2017.

[3] D. J. MacKay, *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[5] T. Dozat and C. D. Manning, "Deep biaffine attention for neural dependency parsing," in *International Conference on Learning Representations*, 2017.

[6] E. Kiperwasser and Y. Goldberg, "Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 313–327, 2016.

[7] O. Duerr, B. Sick, and E. Murina, *Probabilistic Deep Learning: With Python, Keras and Tensorflow Probability*. MANNING PUBN, 2020.

[8] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi, "Credit card fraud detection: A realistic modeling and a novel learning strategy," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 8, pp. 3784–3797, 2018.

[9] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, pp. 141–142, 2012.

[10] R. Simões, "Sunns - msc thesis," 2023, accessed: June 23, 2023. [Online]. Available: https://github.com/ricardosimoes00/Thesis

[11] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, pp. 498–519, 2001.

[12] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol. 27, pp. 533–547, 1981.

[13] R. E. Blahut, *Algebraic Codes for Data Transmission*. Cambridge University Press, 2003.

[14] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[15] J. M. Seddon and J. D. Gale, *Thermodynamics and Statistical Mechanics*. Royal Society of Chemistry, 2002.

[16] G. Montufar, “Restricted boltzmann machines: Introduction and review,” *arXiv preprint arXiv:1806.07066*, 2018.

[17] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities,” *Proceedings of the National Academy of Sciences*, vol. 79, pp. 2554–2558, 1982.

[18] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[19] W. Ping, Q. Liu, and A. T. Ihler, “Learning infinite rbms with frank-wolfe,” in *NIPS*, 2017.

[20] L. A. P. Júnior and J. P. Papa, “Fine-tuning infinity restricted boltzmann machines,” in *SIBGRAPI 2017 - Conference on Graphics, Patterns and Images*. University of Wolverhampton and São Paulo State University, 2017.

[21] G. E. Hinton, “Training products of experts by minimizing contrastive divergence,” *Neural Computation*, vol. 14, pp. 1771–1800, 2002.

[22] S. Geman and D. Geman, “Stochastic relaxation, gibbs distributions, and the bayesian restoration of images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, pp. 721–741, 1984.

[23] M.-A. Côté and H. Larochelle, “An Infinite Restricted Boltzmann Machine,” *Neural Computation*, vol. 28, pp. 1265–1288, 2016.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[26] Y. Xu and W. Yin, “A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion,” *SIAM Journal on Imaging Sciences*, vol. 6, pp. 1758–1789, 2013.

[27] H. H. Bauschke and Y. Lucet, “What is... a fenchel conjugate,” *Notices of the American Mathematical Society*, vol. 59, pp. 44–46, 2012.

[28] G. Papandreou and A. L. Yuille, “Perturb-and-map random fields: Using discrete optimization to learn and sample from energy models,” *2011 International Conference on Computer Vision*, pp. 2257–2264, 2011.

[29] Y. Kwon, J.-H. Won, B. J. Kim, and M. C. Paik, “Uncertainty quantification using bayesian neural networks in classification: Application to biomedical image segmentation,” *Computational Statistics & Data Analysis*, vol. 142, p. 106816, 2020.

[30] T. Auld, A. W. Moore, and S. F. Gull, “Bayesian neural networks for internet traffic classification,” *IEEE Transactions on Neural Networks*, vol. 18, pp. 223–239, 2007.

[31] Y. Gal and Z. Ghahramani, “Dropout as a bayesian approximation: Representing model uncertainty in deep learning,” in *Proceedings of The 33rd International Conference on Machine Learning*. New York, New York, USA: PMLR, 2016, pp. 1050–1059.

[32] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” *arXiv:1312.6114 [stat.ML]*, 2013. [Online]. Available: https://arxiv.org/abs/1312.6114v11

[33] B. Kosko, “Differential Hebbian learning,” in *Neural Networks for Computing*, ser. American Institute of Physics Conference Series, vol. 151, 1986, pp. 277–282.

[34] L. Personnaz, I. Guyon, and G. Dreyfus, “Collective computational properties of neural networks: New learning mechanisms,” *Physical Review A, 34(5):4217*, vol. 34, pp. 4217–4228, 1986.

[35] L. Almeida and J. Neto, “Recurrent backpropagation and hopfield networks,” in *Neurocomputing*, ser. NATO ASI Series, vol. 68. Berlin, Heidelberg: Springer, 1990.

[36] “PyTorch optimizers,” 2023, accessed: June 26, 2023. [Online]. Available: https://pytorch.org/docs/stable/optim.html

[37] A. Gupta, R. Ramanath, J. Shi, and S. Sathiya Keerthi, “Adam vs. sgd: Closing the generalization gap on image classification,” in *OPT2021: 13th Annual Workshop on Optimization for Machine Learning*, 2021.

[38] L. Biewald, “Experiment tracking with weights and biases,” 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, p. 1929–1958, 2014.

[40] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," in *International Conference on Learning Representations*. Toulon, France: International Conference on Learning Representations (ICLR), 2017.

[41] A. Malinin and M. Gales, "Predictive uncertainty estimation via prior networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 7047–7058.

[42] "Bayesian neural network - pytorch," 2022, accessed: June 23, 2023. [Online]. Available: https://github.com/Harry24k/bayesian-neural-network-pytorch

[43] S. Bond-Taylor, A. Leach, Y. Long, and C. G. Willcocks, "Deep generative modelling: A comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 7327–7347, 2022.

[44] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, pp. 27–31, 1994.

[45] S. Ben Driss, M. Soua, R. Kachouri, and M. Akil, "A comparison study between MLP and convolutional neural network models for character recognition," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, 2017, p. 1022306.

[46] B. Hoover, Y. Liang, B. Pham, R. Panda, H. Strobelt, D. H. Chau, M. J. Zaki, and D. Krotov, "Energy transformer," *ArXiv*, vol. abs/2302.07253, 2023.

A

More images generated by the SUNN models

Digits generated by the SUNN model with FFNN-type operations, for the classes 3 to 9 (see Table A.1).

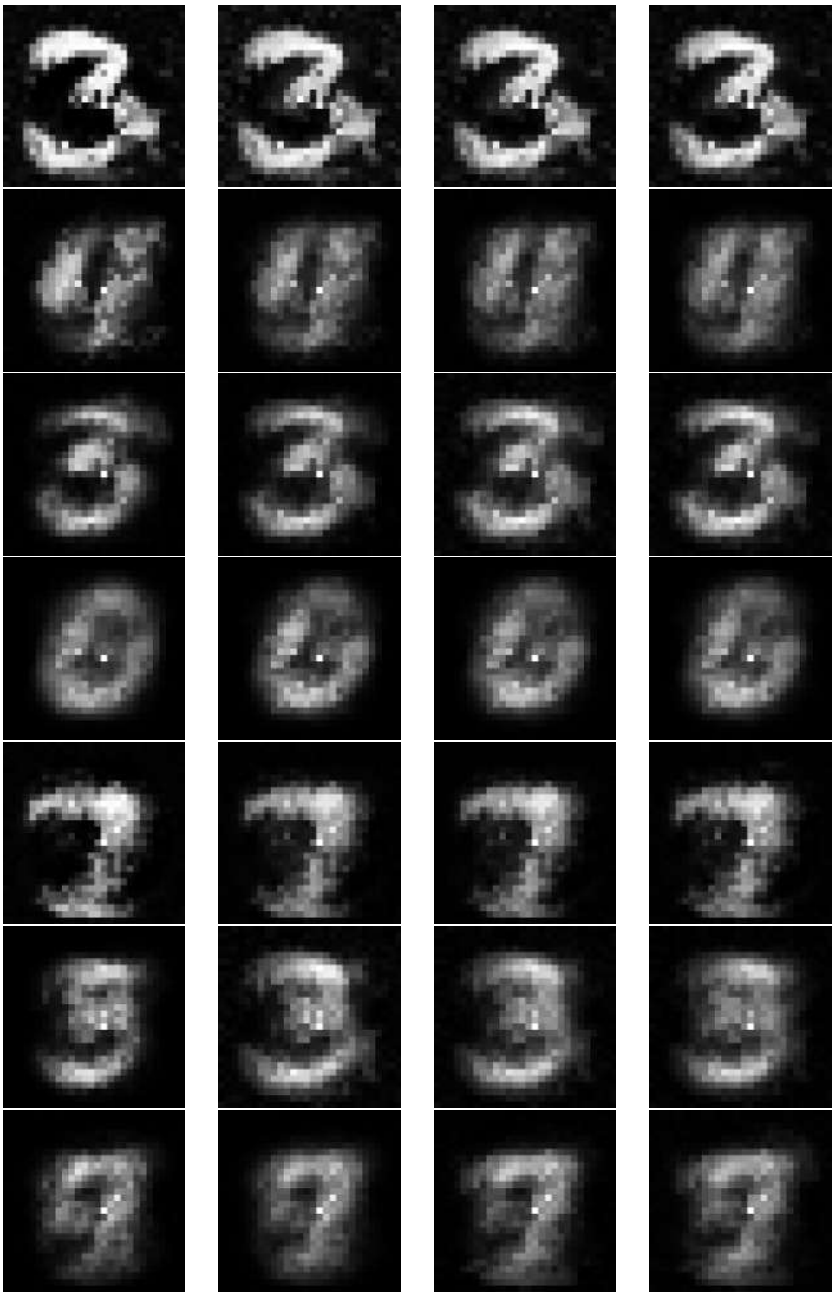


Figure A.1: Digits generated by the SUNN model with FFNN-type operations. From top row to bottom row: classes 3 to 9

Digits generated by the SUNN model with CNN-type operations, for the classes 3 to 9 (see Table A.2).

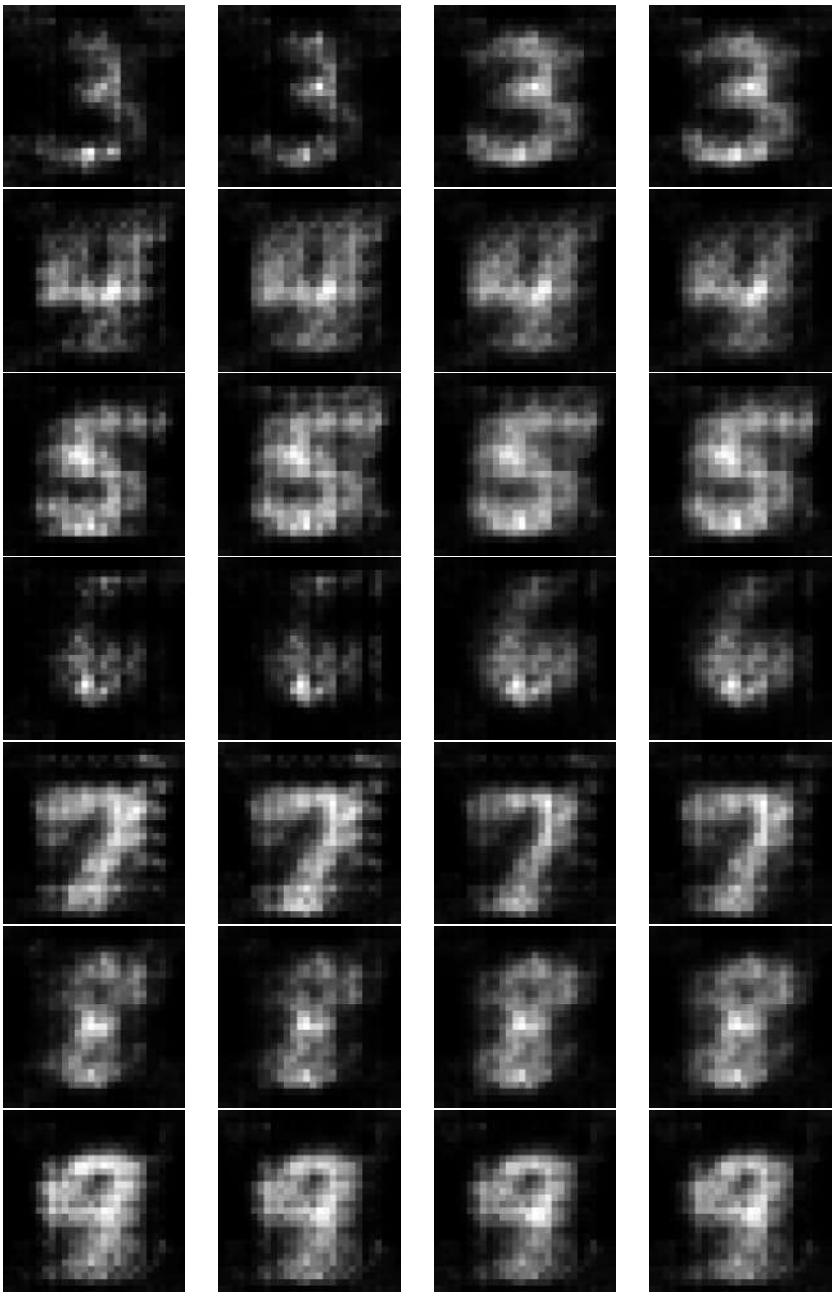


Figure A.2: Digits generated by the SUNN model with CNN-type operations. From top row to bottom row: classes 3 to 9

The conclusions for the remaining classes are similar to the ones we took in sub-Sections 4.4.4 and 4.5.4. We again note that the FFNN version confuses classes 5 and 8 with class 3, showing that this model is unsuitable for prototype generation. On the other hand, the CNN version does not confuse these classes and instead always produces digits the do in fact belong to the class they are coming from. Another difference resembles in the diversity, where the FFNN produces digits with little to no diversity. The diversity is more notorious in the CNN version. We had already seen in sub-Section 4.5.4

that the digits of class 1 had some diversity, and here we also note that, for example in class 3. The top part of first digit seems less elongated and the bottom part is more elongated when compared to the other digits of the same class.

Overall, it can be seen that the SUNN model with CNN operations produces diverse images more on individual pixel-level, since overall the shapes of the digits are quite proximate and they tend to change more on the intensity of the pixel. This is also the case in the SUNN model with FFNN operations, however the latter one tends to mix different classes as seen above.

B

Additional figures regarding BNN models training

The BNNs models used during our experiments had their hyperparameters optimzed, however not as extensively as in the UNN and SUNN models. For the two BNN with FFNN-type operations, the standard deviation the of prior normal distribution is 0.05 and For the two BNN with CNN-type operations, the standard deviation the of prior normal distribution is 0.1. In all models a batch size of 512, and a dropout of 0.3 in training (and also this value in test times, for the BNNs with MC dropout). We noted that the BNNs were very sensible to the standard deviation of the prior, and having larger values did not allow the network to learn.

Below, we present Figure B.1, with the accuracy and loss of the FFNN-type BNN, trained under VI.

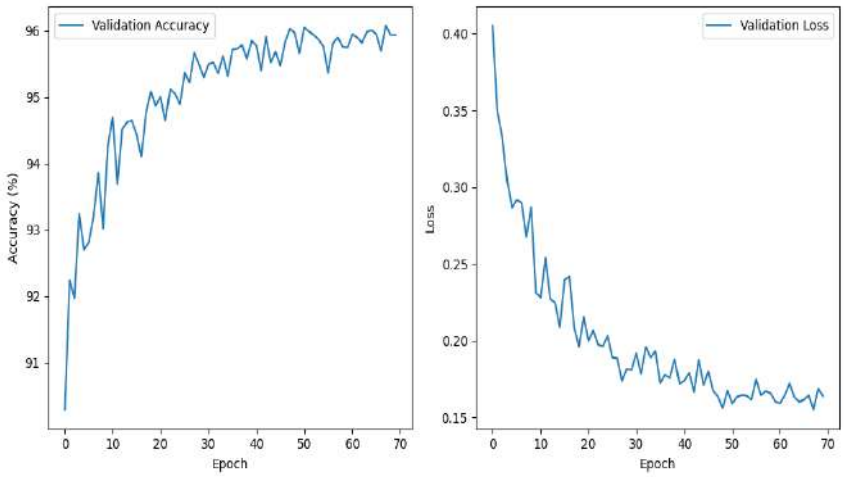


Figure B.1: Training results of the FFNN-type BNN, trained under VI, in the validation set: accuracy plot (left) and loss plot (right)

For our experiments we decided to we the 70^{th} epoch, since it maximized accuracy and minimized the loss. Below, we present Figure B.2, with the accuracy and loss of the FFNN-type BNN, trained under MC dropout.

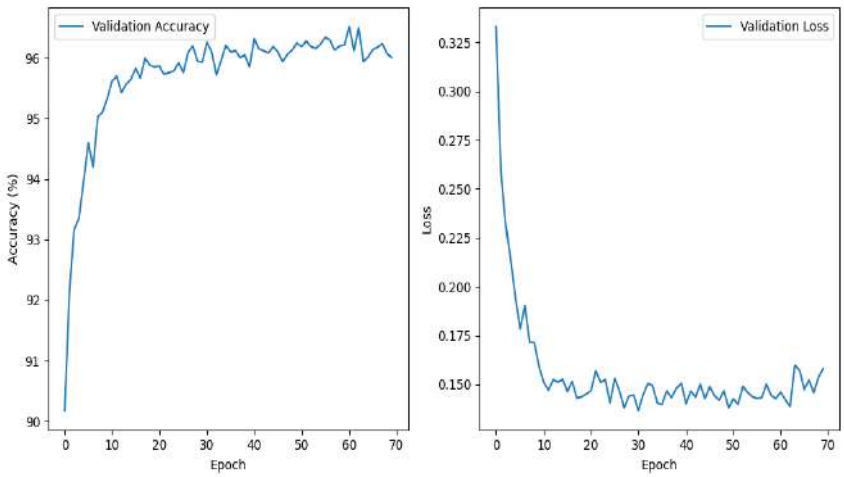


Figure B.2: Training results of the FFNN-type BNN, trained under MC dropout, in the validation set: accuracy plot (left) and loss plot (right)

For our experiments we decided to we the 60^{th} epoch, since the model converged to an optimal high accuracy a low loss value around that epoch. Below, we present Figure B.3, with the accuracy and loss of the CNN-type BNN, trained under VI.

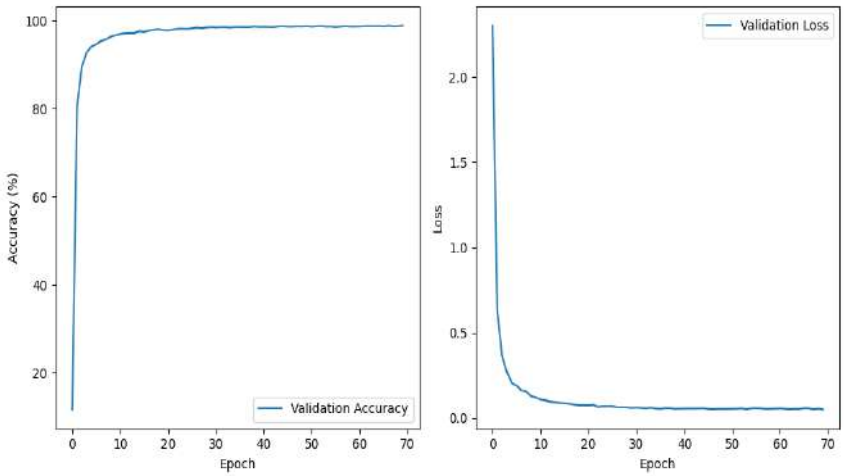


Figure B.3: Training results of the CNN-type BNN, trained under VI, in the validation set: accuracy plot (left) and loss plot (right)

For our experiments we decided to we the 70^{th} epoch, since it maximized accuracy and minimized the loss. Below, we present Figure B.4, with the accuracy and loss of the CNN-type BNN, trained under MC dropout.

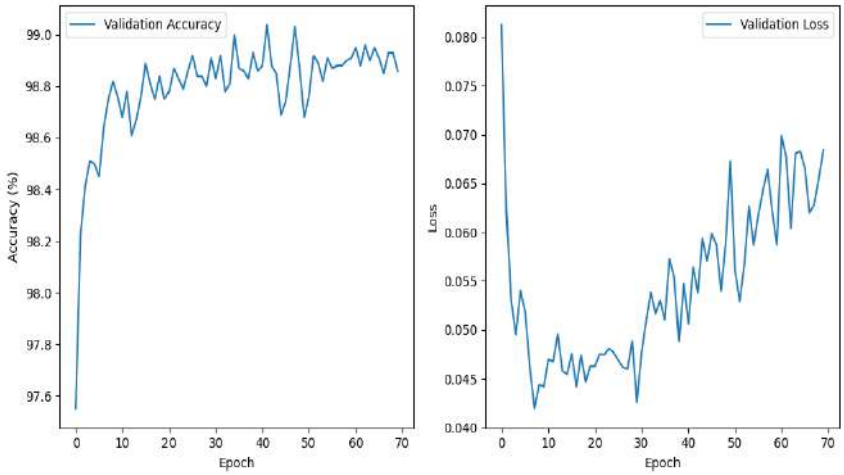


Figure B.4: Training results of the CNN-type BNN, trained under MC dropout, in the validation set: accuracy plot (left) and loss plot (right)

For our experiments we decided to we the 40^{th} epoch, since the model converged to an optimal high accuracy a low loss value around that epoch. However, other epochs could be valid choices like 30^{th} epoch, for example.