CHALMERS
UNIVERSITY OF TECHNOLOGY

Deep Learning with Ensembles of Neural Networks

Estimating the Predictive Uncertainty of Deep Neural Network Classifiers, an example using MNIST digits.

Master's thesis in Complex Adaptive Systems

Henry Yang

Department of Physcis
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

Deep Learning with Ensembles of Neural Networks

Estimating the Predictive Uncertainty of
Deep Neural Network Classifiers, an example using MNIST digits

Henry Yang

Department of Physics
Division of Technical Physics
Chalmers University of Technology
Gothenburg, Sweden 2020

Estimating the Predictive Uncertainty of
Deep Neural Network Classifiers, an example using MNIST digits
Henry Yang

© Henry Yang, 2020.

Supervisor: Bernhard Mehlig, Department of Physics
Examiner: Bernhard Mehlig, Department of Physics

Master's Thesis 2020:NN
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of the classic tale of blind men and an elephant. From Martha Adelaide Holton & Charles Madison Curry, Holton-Curry readers, Rand McNally Co. (Chicago), p. 108.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Henry Yang
Department of Physics
Chalmers University of Technology

Abstract

Deep neural networks are powerful machine-learning models that excel at a large array of machine-learning tasks. A major challenge in machine-learning is the problem of assessing the uncertainty of deep-learning predictors. Coupled with this problem is that deep neural networks tend to make overconfident predictions, which can lead to incorrect decision making where errors go unnoticed. A number of schemes for uncertainty detection have been proposed in recent years ranging from using Bayesian methodology and Monte Carlo simulations to reading neuron states in the upstream layers and letting the neural-networks learn to recognize its certainty. In this thesis, we analyze one of the proposed methods that estimates the predictive uncertainty of deep learning algorithms using an ensemble of deep neural networks. Using classification of handwritten digits as the reference problem, we demonstrate that this method is effective at assessing predictive uncertainty when faced with out-of-distribution inputs and inputs that are distorted by deformation and noise. Our results demonstrated that the distribution of the estimated predictive uncertainty differs substantially between correctly and incorrectly classified inputs, indicating that this method can be used to predict incorrect decision making.

Keywords: Deep Learning, Neural Networks, Ensemble Learning, Machine Learning, Entropy, Predictive Uncertainty, Classification, CNN, Perceptrons.

Acknowledgements

Mainly, I want to thank my supervisor, examiner, and teacher Bernhard Mehlig for all the guidance throughout this project and saw promise in it. I would also like to thank Oleksandr Balabanov and Hampus Linander of the department of Physics for all the help and discussions of the results from the simulations, Balaji Lakshminarayanan for clearing out some questions, and a fellow classmate Martin Selin whom I have worked closely and exchanged ideas with. Finally, I would also like to thank the professors, IT-Students, and MPCAS students who have contributed handwritten digits to the testing data set used throughout my thesis.

Henry Yang, Gothenburg, May 2019

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Problem specification 2
1.1.1 Scope 2
1.1.2 Contributions 2
1.2 Prior work 3

2 Deep-learning and uncertainty estimation **5**
2.1 Artificial neural-networks 5
2.1.1 Multi-layer perceptrons and fully connected layers 6
2.1.2 Convolutional neural-networks 8
2.2 Classification problems 9
2.3 Training the neural-network 11
2.3.1 Energy functions 11
2.3.2 Backpropagation 12
2.3.3 Stochastic gradient-descent and mini batches 13
2.4 Scheme for estimating predictive uncertainty 15
2.4.1 Proper scoring rule 15
2.4.2 Adversarial training 16
2.4.3 Ensembles: training and predictions 16
2.5 Scoring metrics 18
2.5.1 Information Entropy 18
2.5.2 Classification error 18

3 Description of simulations **21**
3.1 Datasets and experiments setup 21
3.2 Testing digits and distortions 22
3.2.1 Line thickness 23
3.2.2 Salt and Pepper Noise 23
3.3 Predictive uncertainty and incorrect decisions 23
3.4 Implementation details 24

4 Results and Discussions **25**
4.1 Comparison of MNIST and CTHMNIST 25

4.2 Distortions using line-thickness 26
4.3 Salt and pepper noise. 30
4.4 Predictive uncertainty and incorrect decisions 31

5 Summary and conclusions 35
5.1 Future work 36

Bibliography 37

A Appendix 1 I
A.1 Algorithm for modifying the line thickness I
A.2 Additional results and discussions II
A.3 Unexplainable anomaly with convolutional neural-networks III
A.4 Additional results with distributions III

List of Figures

2.1 Illustration of a activation process of McCulloch-Pitts neuron. Here, neuron i is connected to 5 other neurons with connection strengths (weights) $w_{i1}, w_{i2}, ..., w_{i5}$, respectively. The incoming signals $v_1, ..., v_5$ are multiplied with the connection strengths and summed together. The activation function $g(.)$ is applied to this sum to turn it into the output s_i. Based on figure 2.1 in [5]. 6
2.2 Illustration of a multi-layer-perceptron with 3 layers, along with the flow of a feed-forward operation. Based on figure 2.2 from [5]. All mathematical transformations applied to the input pattern $\boldsymbol{x}^{(\mu)}$ are documented below each layer. 7
2.3 Illustration of convolution in CNNs. Given a 3×3 input, using a Kernel of size 2×2, the feature maps are calculated according to the formulas in the figure. Based on figure 2.3 in [5] which is based of a figure used by [13]. 9
2.4 A satirical, but very informative description of machine-learning. Reproduced from xkcd.com/1838 under the creative commons attribution-noncommercial 2.5 license. 14

4.1 Samples of both the MNIST test set and CTHMNIST. The top consists of samples from the MNIST test set, and the bottom row consists of samples from CTHMNIST of the same numbers. 26
4.2 The classification error and entropy from an ensemble consisting of multi-layered perceptrons of size 5, 20, 50, and 100. The left panel plots the classification error as a function of line-thickness. The right panel plots the average entropy over the mutated CTHMNIST dataset as a function of line-thickness. These results are produced over 5 independent trials. Here the darker lines represent the average over the runs, while the lighter lines show the different fluctuations from each run. The horizontal dashed lines are the average classification error and average entropy respectively over the trials for the MNIST test set, and the vertical black dashed line is an approximated average line-thickness of the MNIST test set scaled up to that of 280-by-280. 27
4.3 The change of sample images of an 8,4 and a 9 over line-thicknesses. The columns are ordered by line-thickness. Starting from the left most columns, the figure illustrates these same images adjusted to the line-thicknesses in the order 2, 5, 10, 15, 20, 25, and 30. 27

4.4 Classification error and average entropy of ensembles consisting of convolutional nets. This figure illustrates the results from running the same experiment that produces 4.2 using convolutional nets instead of multi-layer perceptrons. The rest of the settings for this figure remains the same as 4.2. 28
4.5 Performance comparison between individual Multi-layer Perceptron classifiers and an ensemble of 100 MLPs. The left panel plots the classification error as a function of line-thickness, and the right panel plots the Average entropy over the CTHMNIST dataset. In both panels, light blue colored lines represent the results of individual MLP classifiers over 100 trials. This is compared to the darker blue line, which corresponds to the results produced by a size 100 ensemble during 1 single trial. The red lines represent the average performance of each individual ANN at each line-thickness. 28
4.6 Same experiments as figure 4.5, produced with CNNs and ensemble of CNNs instead MLPs. 29
4.7 Classification error and entropy produced by 2 ensembles over 5 trials each with the inputs distorted with salt and pepper noise. One ensemble consists of MLPs, and the second consists of CNNs. The noise is applied to the digits adjusted to that of line-thickness with the least classification error. Based previous observed results from figure 4.3, this line-thickness is 14. The noise is applied by first selecting a number of black pixels, turning them white, and then from the unmodified image, choosing a number of white pixels turning them black. Each step on the x-axis corresponds to a multiple of 50 pixels. 30
4.8 Images of an 8 Distorted with salt and pepper noise applied the same way it is described in figure 4.7. The first row is the 280-by-280 sized image of the digits just after the noise is applied. Bottom row images are the results of scaling down the images to 28-by-28, which is the images given to NN classifiers. The leftmost and rightmost columns correspond to removing resp. adding 2000 black pixels, and the middle-left and middle-right columns correspond to removing resp. adding 800 black pixels. 31
4.9 Histogram of Information entropy distribution of correctly and incorrectly classified data points. The blue histograms correspond to correctly classified input, and the red ones are incorrectly classified inputs. These values are produced by using a single multi-layer perceptron classifier. The left-most column is produced using the MNIST test-set, center column CTHMNIST without any adjustments to line-thickness, and the right one is produced using the CTHMNIST digits adjusted to the vicinity of optimal line-thickness, which ranges from 13-15. 32
4.10 Same histograms as in figure 4.9. This one is produced using a single convolutional neural-network trained using MNIST. 32
4.11 The experiments from figure 4.9 and 4.11, but performed using an ensemble of 20 multi-layer perceptrons. 33

4.12 The experiments from figure 4.9 and 4.11, but performed using an ensemble of 20 CNNs. 33

A.1 The results from using an ensemble that uses member voting instead of Averaging the member predictions. II
A.2 Comparisons of between two size 20 ensembles of multi-layered perceptrons where one is trained using adversarial training. These results are produced over 5 independent trials. Once again, the darker colored lines are the average over these independent runs, and the lighter once are the fluctuations. II
A.3 Classification error and average entropy produced by an ensemble of CNNs and an ensemble of MLPs with inputs distorted with Salt and pepper noise. The noise is applied by randomly selecting a number of pixels and flipping their values. The randomly selected white pixels turn black, and the black pixels turn white. In this plot, the noise is applied to CTHMNIST digits adjusted to line thickness with the least classification error. The darker markers represent the average over the trials, and the lighter ones represent the fluctuations over the trials III
A.4 Results from the same experiments as figure A.3. The difference here is that the noise is applied to the unmodified CHTMNIST digits rather then the once adjusted to optimal line thickness. IV
A.5 Images of an 8 Distorted with applied salt and pepper noise. The noise is applied the same way as those that gave the results for figures A.3 and A.4. Sorted according to the number of pixels that were selected to flip their values. From the left, the number of pixels in each column is 1000, 2000, 5000, 10000. IV
A.6 Histograms from the same experiments that produced the previous figures, this time using a 20 member MLP ensemble that uses voting rather than prediction averaging. V
A.7 Histograms of the distribution of entropy like in the previous figures. This figure is produced using ensembles of MLPs trained using adversarial training. V
A.8 Histogram of the distribution of Entropy of heavily distorted inputs. The left column represents the entropy distribution of the CTHMNIST dataset with high line thickness(between 26 and 30), the center column with low line thickness (between 2 and 6) and the third column is made with CTHMNIST with salt and pepper noise with 12000 pixels flipped in the dimensions of 280-by-280. These distributions were produced by a single MLP classifier over 20 trials. VI
A.9 Histogram using the same datasets as A.8, produced over 5 trials using an ensemble of 20 MLPs instead of a single neural network. The columns remain in the same order as they were in A.8. VI

List of Tables

4.1 Comparisons of basic attributes of CTHMNIST and the MNIST test set. The line-thicknesses (LT) are calculated using the methods described by [23] on 28 by 28 images. The CTHMNIST digits are preprocessed, only using the procedure described in [50]. The mean entropy is calculated by averaging the entropy of each digit that were fed to the network. Both the mean accuracy and mean entropy in this table are calculated by averaging the results over 20 independent trials. 25

1

Introduction

Recent research have shown that deep-learning based methods achieve state-of-the-art performance in numerous machine-learning tasks. This includes computer vision [24], image recognition [46], speech recognition [19], natural-language processing [36], and bioinformatics [2]. Well documented success of deep learning methods includes that of conquering the game of Go [45] and the digital strategy game StarCraft2 [49]. Recent research also explored the viability of neural nets and deep-learning based methods in a wider variety of more complex tasks such as sentiment analysis in short texts [11] and it's subproblems like sarcasm detection [40, 41], driving autonomous vehicles [6], and health care [37].

A major challenge in machine-learning is to measure the confidence or uncertainty of the predictions made by the algorithm [42]. While there exist measurement metrics in statistical mechanics and information theory that measure uncertainty, such as information entropy [44], negative-log probability measurements [42], and Bayesian methods [12], there exists no general and uniformly agreed on procedure for measuring uncertainty.

Another problem in deep-learning is that neural-network classifiers tend to be overconfident in their decision making [25], especially when trained on smaller data sets [4]. Studies have shown that this overconfidence can cause the algorithm to make incorrect predictions without noticing [39]. In some cases, this is attributed to overfitting [14], but it remains unclear whether this is the case, or whether the error can be attributed to other causes.

As mentioned above, research has explored the possibilities for using neural net based algorithms to aid decision making for autonomous vehicles [6, 47] and for application in advanced health-care [37]. The ability to measure the predictive uncertainty is crucial for decision making in these areas.

As suggested in ref. [33], a well-calibrated uncertainty (or confidence) measurement can indicate whether to trust the prediction or not.

1.1 Problem specification

It is necessary to know whether a machine-learning algorithm is uncertain about its decision or not. With proper uncertainty measurements, the user can better decide whether or not to trust the decision made by the algorithm.

In this thesis, I analyze the method proposed by Lakshminarayanan et al. [25] for uncertainty estimation using deep neural network ensembles. I evaluate this method using classification of handwritten digits as the reference problem with focus on out-of-distribution inputs and inputs affected by different types of distortion.

In short, the goal of this thesis is to answer the following questions:

- How does the method proposed by [25] perform with regards to out-of-distribution and distorted inputs?
- Can we use the estimated predictive uncertainty to forecast incorrect predictions?

1.1.1 Scope

This thesis focuses on analyzing the method proposed by Lakshminarayanan et al. [25] for estimating predictive uncertainty through an applied example. The method involves training an ensemble of deep neural networks, using a specific objective function for training. As thus, in my experiments, I have used the hyperparameters and network structures similar to those used in Ref. [25].

Unlike Lakshminarayanan et al. [25], who performed experiments on a wide array of problems and datasets for both regression and classification, for my numerical experiments, I consider the task of classifying images of handwritten digits [27]. My goal is to quantify the uncertainty in classifying out-of-distribution- and distorted inputs, using change of line thickness and salt and pepper noise as the two main distortion types considered in the experiments.

1.1.2 Contributions

The main contribution of this thesis if the more in-depth analyses of the uncertainty estimation scheme proposed in ref. [25]. These analyses were carried out using the classification problem of handwritten digits with the MNIST dataset as reference.

This thesis provides a more thorough analyses of the scheme by [25] with regards to out-of-distribution inputs and distorted inputs. Lakshminarayanan et al. also performed experiments involving out-of-distribution inputs. The contributions of

this thesis in this regard includes another applied example of the proposed method using inputs that are different but closer to the training distribution than example demonstrated in ref. [25]. I demonstrate how uncertainty changes as the inputs diverges from the training distribution with regards to one or more attributes of the inputs.

Aside from providing an additional example of the method proposed by [25], this thesis also provides a more in-depth understanding of the methods main components and their impact on the method itself by demonstrating the effect of each component separately and the impact and contribution of these components when used together.

The final contribution of this thesis is the insight in to relationship between predictive uncertainty and incorrect decision making in deep learning algorithm. And how it can be used to forecast an incorrect decision based on the input.

1.2 Prior work

The estimation of predictive uncertainty in machine-learning predictors was highlighted as a problem already in ref. [42]. The case of neural networks possibly failing at assessing their predictive uncertainty is highlighted by many studies such as [25, 31] and [39].

As the interest in adapting neural-networks to encompass uncertainty is growing, publications such as [38] and [15] have focused on utilising Bayesian formalism to train neural networks for this purpose. According to Lakshminarayanan et al. [25], Bayesian neural nets are neural networks where a prior distribution is specified upon the parameters and a posterior distribution of the parameters are found during training. However, Lakshminarayanan et al. [25] argues that Bayesian networks are harder to implement and computationally slower to train in practice because of the modified none-streamlined training process.

Lakshminarayanan et al. [25] proposed a method inspired by the work of Gal et al. [12], which was published earlier. Gal et al. proposed a method of using Monte Carlo dropout, or MC-dropout to estimate uncertainty. According to Lakshminarayanan et al. [25], MC-dropout behaves similarly to machine-learning ensembles, which is why their proposed method consists of training an ensemble of deep neural nets and then evaluate the combined average output of each member network to obtain better-calibrated uncertainty measurements. In their research, ref. [25] demonstrated that the ensemble method outperforms MC-dropout method proposed in [12].

Mandelbaum et al. [33] proposed an evaluation metric for certainty (confidence) that is based on the euclidean distance between the resulting vector of neuron states in the earlier upstream layers. However, their research question is different from that of Lakshminarayanan et al. [25] in that while [25] is interested in the neural network uncertainty and indecisiveness based on the distribution of the outputs, the

measurement metric proposed in ref. [33] is independent of the network outputs.

Like Mandelbaum et al. [33], DeVries et al. [10] proposed a method where the neural nets are trained to calibrate their own uncertainty. The difference between ref. [33] and [10] is that the latter trained the network to output the certainty as a numerical value in an additional output neuron separated from neurons used for the machine-learning task itself. A similarity between the research of Mandelbaum et al. [33] and DeVries et al. [10] is that their proposed methods are independent of the output of the neural networks. However, their method require modifications to the training process, which was already an argument against the Bayesian neural nets in ref. [25].

2

Deep-learning and uncertainty estimation

According to [43], machine-learning is an area in computer science where the algorithms learn and infer. Most problems in machine-learning can be boiled down to approximating the function that maps an input data point $\boldsymbol{x}^{(\mu)}$ to its correct target $\boldsymbol{t}^{(\mu)}$. Cases where the targets are known are called supervised learning, and cases where they are unknown are called unsupervised learning. In this thesis, the focus is on a classification task which falls under supervised learning [35].

Deep-learning is a field in machine-learning where the algorithms are learning by performing the task over and over. The learned concepts are stacked on top of each other to piece together information of higher level [13]. In classic machine-learning, the input needs to be processed and filtered for the necessary information. This process is called feature engineering [43]. According to [13], deep-learning mitigates the need of feature engineering.

The first part of this chapter deals with the basics of deep-learning, what are neural-networks, and how to train them. The second part of this chapter describes the scheme proposed by Lakshminarayanan et al. [25] using neural-network ensembles. In the last section, I present the metrics I use in this thesis. This includes classification error, accuracy, and information entropy.

2.1 Artificial neural-networks

Neural-networks are mathematical models inspired by the interaction between neurons in the brain [35, 13]. In the brain, brain cells, also known as neurons, are wired together to form a complex network. Information is transmitted from neuron to neuron using their intermediate connections as electrical signals [35]. A neuron processes the information it receives from its connected neurons. An output is produced and sent to other connected neurons.

In machine-learning, most neural-networks are based on the neuron model proposed by McCulloch and Pitts [34, 20]. Consider a neuron unit i which is connected to

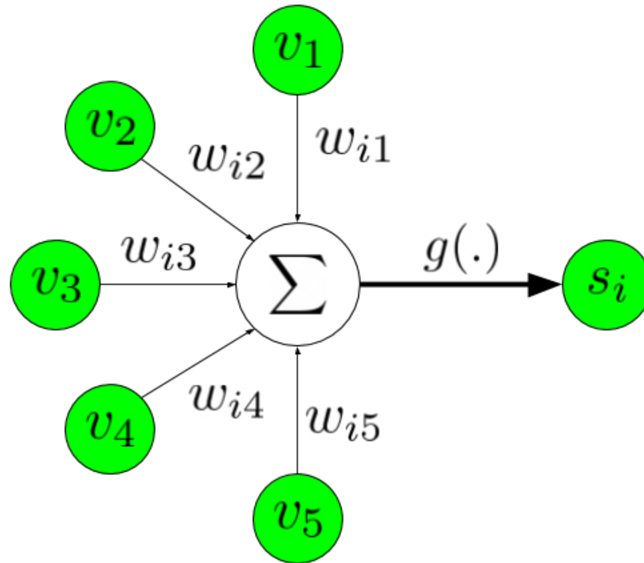

Figure 2.1: Illustration of a activation process of McCulloch-Pitts neuron. Here, neuron i is connected to 5 other neurons with connection strengths (weights) $w_{i1}, w_{i2}, ..., w_{i5}$, respectively. The incoming signals $v_1, ..., v_5$ are multiplied with the connection strengths and summed together. The activation function $g(.)$ is applied to this sum to turn it into the output s_i. Based on figure 2.1 in [5].

N neurons j in its vicinity. McCulloch and Pitts [34] modeled the activation s_i of neuron i using equation (2.1). Here, v_j represents the incoming signals from the j connected neurons, w_{ij} represents the connection strengths (weights) between neuron i and j. The function $g(.)$ is called an activation function, and θ_i represents the activation threshold (this is also known as bias [13])

$$s_i = g\left(\sum_{j}^{N} w_{ij} v_j - \theta_i\right). \tag{2.1}$$

A simple caricature of this neuron model is illustrated in figure 2.1.

2.1.1 Multi-layer perceptrons and fully connected layers

Suppose that we are modeling the output of several neurons, let $\boldsymbol{s}$ denote the n-dimensional vector that represents the outputs produced by n neurons. A consequence of (2.1) is that these outputs can be computed by computing each component s_i using eq (2.3). The vector $\boldsymbol{b}$ can be computed using linear algebra according to (2.2). Here, $\mathbb{W}$ is the $n \times m$ weights matrix that represents the connection weight between these n neurons and m other neurons. The vector $\boldsymbol{v}$ is the m-dimensional vector that represents the incoming signals from the m connected neurons [35]

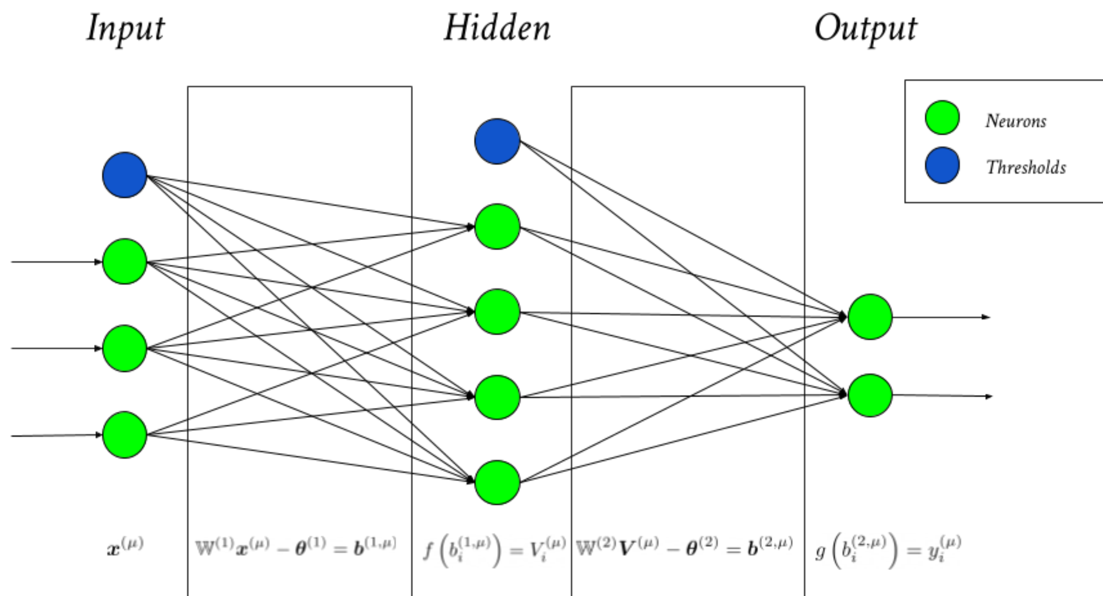

Figure 2.2: Illustration of a multi-layer-perceptron with 3 layers, along with the flow of a feed-forward operation. Based on figure 2.2 from [5]. All mathematical transformations applied to the input pattern $\boldsymbol{x}^{(\mu)}$ are documented below each layer.

$$\boldsymbol{b} = \mathbb{W}\boldsymbol{v} - \boldsymbol{\theta}, \tag{2.2}$$
$$s_i = g(b_i). \tag{2.3}$$

Note that eq. 2.3 is applied element-wise on the entire vector $\boldsymbol{b}$ In deep-learning, several McCulloch-Pitts neurons are wired together in a layer-by-layer-structure using equation (2.3) and (2.2). These networks consist of an input layer, an output layer, and a number of hidden layers. Such networks are known as multi-layer perceptrons (MLPs) [35]. An example of a MLP network with one hidden layer is illustrated in figure 2.2.

Suppose that we are to feed the n-dimensional input pattern $\boldsymbol{x}^{(\mu)}$ through an MLP network similar to the one illustrated in figure 2.2 with m neurons in the hidden layer. The activation $s_i^{(1,\mu)}$ for each neuron i in the hidden layer is calculated by

$$s_i^{(1,\mu)} = g\left(\sum_{j=1}^{n} w_{ij}^{(1)} x_j^{(\mu)} - \theta_i^{(1)}\right). \tag{2.4}$$

The final outputs $y_j^{(\mu)}$ from this network is then calculated as a function of the neuron synapses $s_k^{(1,\mu)}$ from the hidden layers by equation (2.5)

$$y_j^{(\mu)} = f\left(\sum_{k=1}^{m} w_{jk}^{(2)} s_k^{(1,\mu)} - \theta_j^{(2)}\right). \tag{2.5}$$

In the case were there are L hidden layers instead of just 1, then the outputs are still calculated in the same way as in equation (2.5), and each of neuron activation $s_j^{(\ell,\mu)}$ in any hidden layer ℓ are calculated in terms of the previous layer $\ell-1$ according to

$$s_i^{(\ell,\mu)} = g\left(\sum_j w_{ij}^{(\ell)} s_j^{(\ell-1,\mu)} - \theta_i^{(\ell)}\right). \tag{2.6}$$

A layer where all neurons are connected to every neuron in the previous layer is known as a *fully connected layer*[13], in some literature, these layers are known as *dense* layers [9].

2.1.2 Convolutional neural-networks

According to [13], convolutional neural-networks (CNNs) are neural-networks designed to process inputs that are two-dimensional images. While a regular fully connected layer makes use of matrix multiplications. Convolutional nets uses a variation of *discrete convolution* [13]. Convolution is an operation between two functions $\boldsymbol{x}$ and $\boldsymbol{w}$ of the same dimensionality defined as follows

$$s(i) = (\boldsymbol{x} * \boldsymbol{w})(i) = \sum_a x_a w_{a-i}. \tag{2.7}$$

Here, the "$*$"-operator denotes the convolution operator. In neural-networks, the first argument $\boldsymbol{x}$ of (2.7) is known as the input, and the second argument $\boldsymbol{w}$ is called the kernel. Usually, the input is a multidimensional array (a.k.a a tensor), and the kernel is a multidimensional array of parameters. Convolution is also an operation that is conveniently defined on multiple axes. Suppose that the input is a two dimensional image $\mathbb{X}^{(\mu)}$, then the kernel $\mathbb{W}^{(1)}$ also has to be two-dimensional. A consequence of this is that the output is also two dimensional. The activation $S_{ij}^{(1,\mu)}$ for neuron (i,j) in this convolutional layer is calculated according to eq. (2.8). Here $g(.)$ is the activation function. In the context of convolutional neural-networks, $S_{ij}^{(1,\mu)}$ is also known as a *feature map* [35]

$$S_{ij}^{(1,\mu)} = g\left[(\mathbb{X}^{(\mu)} * \mathbb{W}^{(1)})(i,j)\right] = g\left[\sum_m \sum_n x_{i+m,j+n}^{(\mu)} w_{mn}^{(1)}\right]. \tag{2.8}$$

Note the change of sign in the indices used in (2.8) versus (2.7). Equation (2.8) is an alternative form for writing convolution that is more widely used when implementing convolutional-nets [13]. Some literature describes them as filters that scan and perform the convolution operation in eq. (2.8) on smaller parts of the input at a time. Figure 2.3 illustrates the convolution operations that occur in convolutional-layers.

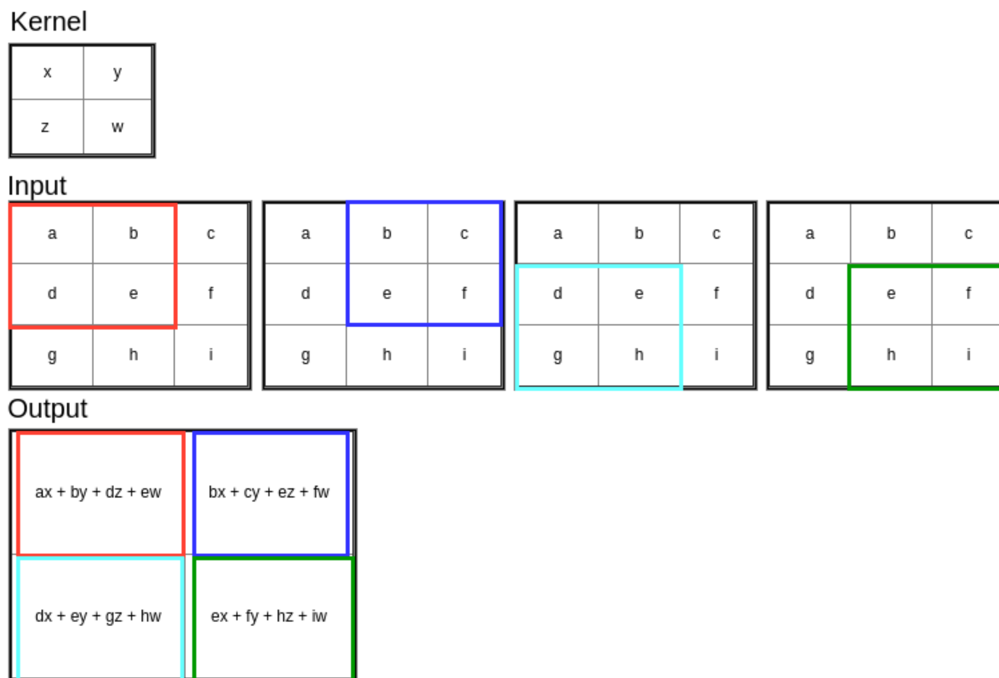

Figure 2.3: Illustration of convolution in CNNs. Given a 3×3 input, using a Kernel of size 2×2, the feature maps are calculated according to the formulas in the figure. Based on figure 2.3 in [5] which is based of a figure used by [13].

Additionally, the outputs from convolutional-layers also filtered through a so-called pooling-layer. A pooling-layer takes the output from its connected feature maps and summarize them to a single number. Examples of pooling layers include the max-pooling layer which, outputs the maximum of its connected feature maps, and the l2-pooling layer that computes the root-mean-square value of the feature map outputs [35].

2.2 Classification problems

Previously, we described machine-learning problems as approximating the function that maps a given input $\boldsymbol{x}^{(\mu)}$ to its correct target $\boldsymbol{t}^{(\mu)}$. In machine-learning, classification problems are defined on a finite set of N classes [1] where each input $\boldsymbol{x}^{(\mu)}$ is supposed to be assigned to their respective correct class by the algorithm. The task of recognizing handwritten digits from the MNIST-dataset is an example of a classification task [25].

Since each class is a discrete-element in the finite-set of N classes (or labels), they can be represented numerically as integers. Let $c^{(\mu)} = 1, ..., N$ denote the integer representing the correct class for input $\boldsymbol{x}^{(\mu)}$. An approach to classification problems in machine-learning is to approximate the discrete probability distribution of the

[1] Hence, classification problems. These are also called labels

conditional probability $y_i^{(\mu)} = P(c^{(\mu)} = i|\boldsymbol{x}^{(\mu)})$ such that $y_i^{(\mu)}$ is largest when i corresponds to the correct class for input $\boldsymbol{x}^{(\mu)}$ (i.e. when $i = c^{(\mu)}$), and smaller otherwise. A machine-learning model that uses this prediction scheme is called a probabilistic-classifier [43].

Training neural-networks as probabilistic classifiers requires the labels $c^{(\mu)}$ to be encoded into the targets $\boldsymbol{t}^{(\mu)}$. This encoding is done by modelling the targets as N-dimensional vectors using

$$t_i^{(\mu)} = \begin{cases} 1 & \text{if } i = c^{(\mu)} \\ 0 & \text{otherwise} \end{cases}. \tag{2.9}$$

This kind of unit vectors are sometimes referred to as one-hot-vectors in computer science [18].

Once an output $\boldsymbol{y}^{(\mu)}$ is obtained from a trained neural-network, a class is assigned to the pattern $\boldsymbol{x}^{(\mu)}$ that was fed to the network. Usually this is done by computing the one-hot-vector $\hat{\boldsymbol{t}}^{(\mu)}$ by

$$\hat{t}_i^{(\mu)} = \begin{cases} 1 & \text{if } i = \arg\max_i \boldsymbol{y}^{(\mu)} \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

This is equivalent of calculating the integer $\hat{c}^{(\mu)} = \arg\max_i \boldsymbol{y}^{(\mu)}$ directly. In which case, the ultimate goal of training a probabilistic-classifier is to be able to assign any given input $\boldsymbol{x}^{(\mu)}$ its correct class. Mathematically, this means that a well-trained algorithm is expected to calculate $\hat{\boldsymbol{t}}^{(\mu)}$ such that

$$\forall \mu, i : \hat{t}_i^{(\mu)} = t_i^{(\mu)}. \tag{2.11}$$

Equivalently, $\forall \mu : \hat{c}^{(\mu)} = c^{(\mu)}$ is expected of a well-trained algorithm if $\hat{c}^{(\mu)}$ is computed. While these two conditions are equivalent, $\hat{\boldsymbol{t}}^{(\mu)}$ introduces more mathematical conveniences in terms of expressing classification error (and classification accuracy) in later sections in this chapter.

According to [35], the outputs of a neural-networks can be transformed into probabilities by using the softmax activation function in the output layer. This function is defined as

$$y_i = \frac{e^{\alpha b_i}}{\sum_{c=1}^{N} e^{\alpha b_c}}. \tag{2.12}$$

Here b_i stands for the local-field of neuron i in the output layer $b_i = \sum_j w_{ij} v_j - \theta_i$. The parameter α is usually set to unity ($\alpha = 1$). Apart from transforming the outputs of a neural net into probabilities, the output of softmax y_i is largest when

the local-field b_i is the largest, and smaller otherwise. This property also ensures that neural nets can be used as probabilistic classifiers.

2.3 Training the neural-network

In order for an neural-network to correctly predict the label of an input pattern $\boldsymbol{x}^{(\mu)}$, correct weights $\mathbb{W}^{(\ell)}$ and thresholds $\boldsymbol{\theta}^{(\ell)}$ are required. One question remains, how do we calculate the correct weights and thresholds?

Given a classification problem with training set $T = \{(\boldsymbol{x}_T^{(\mu)}, \boldsymbol{t}_T^{(\mu)}) | \mu = 1, ..., p\}$ consisting of p training samples. The problem of finding the correct weights can be seen as an optimization problem where the parameters are adjusted iteratively to minimize a loss function, also known as the *energy function* [35, 13]. For each iteration, the parameters are updated using *gradient-descent*.

2.3.1 Energy functions

An energy function H is a function defined by the inputs, targets, and the weights and thresholds of the neural-network. For classification problems, the function H needs to fulfill the following:

1. H has to be differentiable with respect to the parameters in the network.
2. H decreases when the classification error (probability of a neural-network model making an incorrect prediction) decreases.

The first rule is needed since it allows us to update any parameter σ (denoting any weight connection w_{ij} or threshold θ_i) in the network using the gradient-descent update rule defined in (2.13). The parameter η in (2.13) is the step length. In the context of neural-networks training, η is also known as the learning rate

$$\sigma^{(t+1)} = \sigma^{(t)} - \eta \frac{\partial H}{\partial \sigma}. \tag{2.13}$$

The second property ensures that the classification error of the network decreases as we adjust the parameters of the network to minimize the energy function. According to [35, 25] and [42], the negative log-likelihood function in eq. (2.14) is the preferred energy function when training neural-network models to perform classification tasks. Negative log-likelihood is also known as the *cross-entropy loss function* [42]. This is also the energy function used in this thesis

$$H = -\sum_{\mu}\sum_{i} t_i^{(\mu)} \log_2 y_i^{(\mu)}. \tag{2.14}$$

2.3.2 Backpropagation

The procedure of finding the gradients of the energy function with respect to the network parameters is known as error-backpropagation [28]. Suppose that we have a multi-layer perceptron with L hidden layers, given an input pattern $\boldsymbol{x}^{(\mu)}$, the neuron state update rules in each of the layers would be that of (2.15),(2.16) and (2.17)

$$V_i^{(\ell,\mu)} = g(b_i^{(\ell,\mu)}), \qquad b_i^{(\ell,\mu)} = \sum_i w_{ij}^{(\ell)} x_j^{(\mu)} - \theta_i^{(\ell)}, \qquad \ell = 1, \tag{2.15}$$

$$V_j^{(\ell,\mu)} = g(b_j^{(\ell,\mu)}), \qquad b_j^{(\ell,\mu)} = \sum_k w_{jk}^{(\ell)} V_k^{(\ell-1,\mu)} - \theta_j^{(\ell)}, \qquad 1 < \ell < L, \tag{2.16}$$

$$y_k^{(\mu)} = f(b_k^{(O,\mu)}), \qquad b_k^{(O,\mu)} = \sum_k w_{kl}^{(O)} V_l^{(\ell,\mu)} - \theta_k^{(O)}, \qquad \ell = L. \tag{2.17}$$

To calculate the update gradient $\dfrac{\partial H}{\partial w_{mn}^{(O)}}$ for the weights $w_{mn}^{(O)}$ that connects layer L to the output layer, we differentiate the energy function H by using the chain rule according to (2.18) where $\frac{\partial H}{\partial y_i^{(\mu)}}$ is the derivative of the energy function w.r.t. to neuron $y_i^{(\mu)}$, $\frac{dy_i^{(\mu)}}{db_i^{(O,\mu)}}$ is the derivative of the output w.r.t. to its local-field, and finally, $\frac{db_i^{(O,\mu)}}{dw_{mn}^{(L)}}$ is the derivative of the local-field w.r.t. the weight $w_{mn}^{(L)}$

$$\frac{\partial H}{\partial w_{mn}^{(O)}} = \sum_{\mu}\sum_{i} \frac{\partial H}{\partial y_i^{(\mu)}} \frac{dy_i^{(\mu)}}{db_i^{(O,\mu)}} \frac{db_i^{(O,\mu)}}{dw_{mn}^{(L)}}. \tag{2.18}$$

Using the equations (2.15),(2.16) and (2.17), we can calculate each of the derivatives in (2.18) into (2.19) and (2.20). Here $f'(.)$ is the derivative of the activation function f of the output layer, and δ_{im} denotes the Kronecker delta (i.e., $\delta_{im} = 1$ if $i = m$, otherwise 0)

$$\frac{dy_i}{db_i} = f'(b_i), \tag{2.19}$$

$$\frac{db_i}{dw_{mn}} = \delta_{im} V_n^{(L,\mu)}. \tag{2.20}$$

Inserting (2.19) and (2.20) into (2.18), the final equation for calculating the gradients with respect to the weight becomes eq (2.21)

$$\frac{\partial H}{\partial W_{mn}^{(O)}} = \sum_{\mu} \underbrace{\sum_{i} \frac{\partial H}{\partial y_i^{(\mu)}} f'(b_i)\delta_{im}}_{\Delta_E^{(O,\mu)}} V_n^{(L,\mu)} = \sum_{\mu} \Delta_E^{(O,\mu)} V_n^{(L,\mu)}. \tag{2.21}$$

A consequence of the equations for the local-field, differentiating the local-field w.r.t. thresholds would always be -1. Thus the update gradient for the thresholds becomes (2.22).

$$\frac{\partial H}{\partial \theta_m^{(O)}} = \sum_{i\mu} \frac{\partial H}{\partial y_i^{(\mu)}} f'(b_i)\delta_{im}(-1) = -\sum_{\mu} \Delta_E^{(O,\mu)}. \tag{2.22}$$

Next, the parameters between layer L and $\ell = L-1$ are to be updated. The gradients for the weights in the layer are obtained by applying the chain rule repeatedly [35]. To find the gradients to the next set of weights, we reapply the chain rule 3 times. This is done using equation (2.23),(2.24), and (2.25)

$$\frac{\partial H}{\partial w_{mn}^{(L)}} = \sum_{\mu} \sum_{i} \frac{\partial H}{\partial y_i^{(\mu)}} \frac{dy_i^{(\mu)}}{db_i^{(O,\mu)}} \frac{db_i^{(O,\mu)}}{dw_{mn}^{(L)}} \quad , \tag{2.23}$$

$$\frac{\partial b_i^{(O,\mu)}}{\partial w_{mn}^{(L)}} = \sum_{j} w_{ij}^{(O)} \frac{dV_j^{(L,\mu)}}{dw_{mn}^{(L)}} \quad , \tag{2.24}$$

$$\frac{\partial V_j^{(L,\mu)}}{\partial w_{mn}^{(L)}} = \frac{\partial V_j^{(L,\mu)}}{\partial b_j^{(L,\mu)}} \frac{db_j^{(L,\mu)}}{dw_{mn}^{(L)}} = g'(b_j^{(L,\mu)})\delta_{jm} V_n^{(\ell,\mu)} \quad . \tag{2.25}$$

Similarly, to compute the gradients for the consecutive layers, we apply the chain rule repeatedly to find the update gradients for those layers. This is possible due to the layered structure of multi-layer perceptrons [35]. Once the gradients for the layers are computed, the weights and thresholds are updated using the update rule (2.13). A direct implication of eqs. (2.19) and (2.25) is that the activation functions must be differentiable.

2.3.3 Stochastic gradient-descent and mini batches

In order to find the correct weights and thresholds for deep neural-network, an iterative process that uses gradient-descent is used to update the weights. The gradients are calculated using error backpropagation described in section 2.3.2.

In the beginning, weights and thresholds are initialized using a chosen distribution [35]. Training samples $\boldsymbol{x}_T^{(\mu)}$ are then fed through the network. Using an energy function H, the update gradients of the weights and thresholds in each layer is

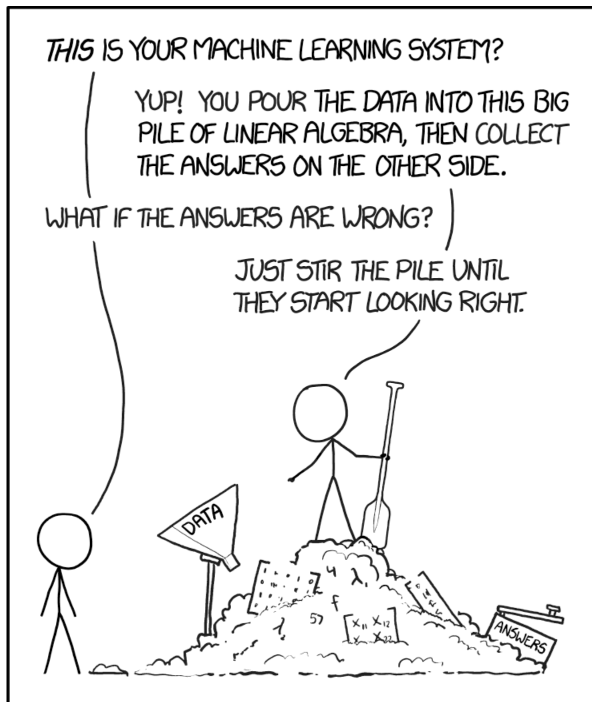

Figure 2.4: A satirical, but very informative description of machine-learning. Reproduced from xkcd.com/1838 under the creative commons attribution-noncommercial 2.5 license.

calculated by applying chain rule error backpropagation. Then the parameters in the neural net model are updated using eq (2.13). Albeit satirical, figure 2.4 works as a summary of the process of training deep neural nets.

As the energy function H is not necessarily convex in the space of the weights and thresholds [22], stochastic methods are used to mitigate the possibilities of the training to converge in a local optimum in H w.r.t. to the network parameters. One of these methods is to use a stochastic gradient-descent algorithm where the training patterns are either sampled one by one, or by simply shuffling the order of the training patterns between each iteration of feeding all p patterns, one such iteration is also known as an epoch [35]. In most cases, this algorithm minimizes the risk of the weights getting stuck on saddle points, and local optima [26].

According to [35], it is also possible to feed all training samples in one go. This is called batch training. While this is more convenient when writing proofs, in general, the iterative stochastic approach is more favoured due to the latter being less prone to converge prematurely into a local optimum [13, 35]. The number of operations remains the same in both cases, batch training have the advantage of being faster in terms of computational time, due to parallel computation algorithm applied to matrix- and vector-operations [35].

A middle ground between batch and full stochastic gradient-descent is to use mini batches. The training samples are randomly permuted and divided into mini batches

of size m_B. The energy function is then normalized across the mini batch. According to [35], this can speed up the training significantly while still mitigating the risk of the training to converge prematurely.

Stochastic gradient-descent can be augmented using an adaptive learning rate with regulated momentum depending on the gradient norm. One way of doing this results in the optimization algorithm known as *Adam*. This optimization algorithm were proposed by Kingma et al. in ref. [22].

2.4 Scheme for estimating predictive uncertainty

This section present the scheme for estimating the predictive uncertainty that was proposed by Lakshminarayanan et al. [25]. Given a classification problem with N labels and the set $T = \left\{(\boldsymbol{x}_T^{(\mu)}, \boldsymbol{t}_T^{(\mu)})|\mu = 1, ..., p\right\}$ of p training samples and corresponding targets. The scheme proposed in ref. [25] consist of the following three components

1. Choosing a proper scoring rule as training criterion.
2. Use *adversarial training* to smooth the predictive distribution.
3. Train an *ensemble*

These three components are described in detail in the following subsections.

2.4.1 Proper scoring rule

Lakshminarayanan et al. [25] described a proper scoring rule as a function that rewards better-calibrated predictions over worse. While this is similar to the properties of energy functions listed in sect. 2.3.1, a scoring function S assigns higher values to better-calibrated predictions instead of lower.

Suppose that $S(\boldsymbol{y}^{(\mu)}, (\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)}))$ is a scoring rule that evaluates the predictive distribution made by a trained network $y_i^{(\mu)} = P\left(c^{(\mu)} = i|\boldsymbol{x}^{(\mu)}\right)$. Here, $c^{(\mu)}$ is the integer representing the correct class of input pattern $\boldsymbol{x}^{(\mu)}$ (see sect. 2.2), $y_i^{(\mu)}$ is output i of a neural-network with N outputs. Most importantly, S needs to satisfy (2.26) and the equivalence implication in (2.27)

$$S[\boldsymbol{y}^{(\mu)}, (\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})] \leq S[\boldsymbol{t}^{(\mu)}, (\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})], \tag{2.26}$$

$$S[\boldsymbol{y}^{(\mu)}, (\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})] = S[\boldsymbol{t}^{(\mu)}, (\boldsymbol{x}^{(\mu)}, \boldsymbol{t}^{(\mu)})] \iff \boldsymbol{y}^{(\mu)} = \boldsymbol{t}^{(\mu)}. \tag{2.27}$$

In their scheme, the networks are trained using the energy function $H = -S$

According to [25], many common energy functions used for training neural-networks fulfill the property required for a proper scoring rule. This includes the negative log-likelihood function in (2.14) that is used when training neural nets as a probabilistic classifier.

2.4.2 Adversarial training

Adversarial examples are datapoints that are specially engineered to trick a machine-learning algorithm into making an incorrect decision [39, 31]. Given an arbitrary input pattern $\boldsymbol{x}^{(\mu)}$ and an energy function H, an efficient way of generating an adversarial example $\boldsymbol{x}'^{(\mu)}$ is to use eq. (2.28). This method is known as the fast gradient sign method [14]

$$\boldsymbol{x}'^{(\mu)} = \boldsymbol{x}^{(\mu)} + \epsilon \mathrm{sign}(\nabla_{\boldsymbol{x}^{(\mu)}} H). \tag{2.28}$$

The idea behind adversarial training (AT) is to generate adversarial examples during training and then treat them as additional samples in the training set [14]. This is done by generating the adversarial example $\boldsymbol{x}'^{(\mu)}$ (using for instance (2.28)) for each given input $\boldsymbol{x}^{(\mu)}$ and then perform the gradient-descent procedure with error backpropagation using the generated $\boldsymbol{x}'^{(\mu)}$ as input to the network. According to [14], training a machine-learning algorithm this way results in improved robustness of the classifier.

2.4.3 Ensembles: training and predictions

In machine-learning, ensembles are models that combine multiple machine-learning models to produce better results [29]. Deep ensembles refer to ensembles consisting of deep neural nets.

In general, there are two ways of training an ensemble. One is randomization methods were each model is trained independently of each other. The other approach is the use of boosting, where the ensemble members are trained sequentially [51]. In this thesis, I mainly used randomization methods since this is the training model favored by [25]. Another motivation for the use of randomization over boosting is that it allowed me to train each ensemble member independent of each other, which means that they could be trained in parallel.

Given an ensemble size K, the training scheme use by Lakshminaryanan et al. [25] can be boiled down to these steps

1. Initialize each of the K members' weights and thresholds independently.

2. Each member is then given the whole training set shuffled randomly.
3. Train each of the ensemble members using their shuffled training samples as input.
4. An alternative to 3), use adversarial training described in sect 2.4.2.

Another way of training randomization ensembles is to use bagging (or bootstrapping), where each member is given a subset of the training data [7]. I favored the training procedure used by [25] because according to [29, 25] it is outperforming bagging methods.

Training aside, there exist numerous ways of modeling the prediction made by an ensemble [29, 30, 51]. Suppose that we have an ensemble of size K, with $\boldsymbol{y}^{(k,\mu)}$ being the output of member k, then one way of modeling the ensemble prediction $\boldsymbol{Y}^{(K,\mu)}$ is to average the predictions made by each member using

$$Y_i^{(K,\mu)} = \langle y_i^{(k,\mu)} \rangle \iff \frac{1}{K}\sum_{k=i}^{K} y_i^{(k,\mu)}. \tag{2.29}$$

In their work, Lakshminaryanan et al. [25] modeled the ensemble output using the averaging model in eq (2.29). According to [51], this scheme is more favored when facing regression problems. Zhou et al. [51] and Hansen et al. [17] both suggested that a prediction scheme more resembling a voting system is better suited for classification problems. Hansen et al. [17] described a vote by member k to be the one-hot-vector $\hat{\boldsymbol{t}}^{(k,\mu)}$ calculated by

$$t_i^{(k,\mu)} = \begin{cases} 1 & \text{if } i = \arg\max_i \boldsymbol{y}^{(k,\mu)} \\ 0 & \text{otherwise.} \end{cases} \tag{2.30}$$

Note that this is exactly the same as classifying operation in (2.10). Using this method, the ensemble prediction $\boldsymbol{Y}^{(K)}$ becomes

$$Y_i^{(K,\mu)} = \frac{1}{K}\sum_{k=1}^{K} \hat{t}_i^{(k,\mu)}. \tag{2.31}$$

In this thesis, both ensembles using averaging and voting are used to compare their impact with regards to performance and predictive uncertainty estimation.

2.5 Scoring metrics

In this thesis, I mainly use two measurement metrics to evaluate the scheme for estimating predictive uncertainty. These two metrics are information entropy and classification error. Entropy is used to measure predictive uncertainty. This is done in order to analyze the relation between entropy and classification error.

2.5.1 Information Entropy

Information entropy is a metric proposed by Claude Shannon [21, 44]. Let X denote a discrete random variable with N observable outcome with $P(X = i), i = 1, ..., N$ denoting the probability to observe outcome i. In ref. [44], Shannon defined the information entropy $Z(P(X))$ as

$$Z(P(X)) = -\sum_{i=1}^{N} P(X = i) \log_2 P(X = i). \tag{2.32}$$

An important property of $Z(P(X))$ is that it is a symmetric function [44]. Furthermore, the global maximum of $Z(P(X))$ occurs when X is uniformly distributed (i.e. $P(X = i) = 1/N$) and 0 when X is distributed such that there exist an event j that can be observed with union probability (i.e. $P(X = j) = 1$), and 0 otherwise ($P(X \neq j) = 0$). In other words, information entropy is a function of the broadness of the distribution, which also falls under our intuitive understanding of uncertainty as we are more uncertain about the outcome when observing an event distribution that is broader compared to a sharply peaked one [21]. This is the main reason I use entropy for measuring predictive uncertainty of neural-network predictions.

Suppose that $\boldsymbol{y}^{(\mu)}$ is the output from feeding the input pattern $\boldsymbol{x}^{(\mu)}$ through a neural-network model. The entropy of the network output $\boldsymbol{y}^{(\mu)}$ can then be computed by

$$Z(\boldsymbol{y}^{(\mu)}) = -\sum_{i=1}^{N} y_i^{(\mu)} \log_2 y_i^{(\mu)}. \tag{2.33}$$

2.5.2 Classification error

Classification error is an estimation of the probability of the neural-network making an incorrect decision. Assuming a classification problem with N labels, then the classification error can be calculated using eq. (2.34) where the vector $\hat{\boldsymbol{t}}^{(\mu)}$ is the predicted target by the classifier that is calculated using (2.10) [35]

$$C = \frac{1}{2p} \sum_{\mu=i}^{p} \sum_{i=1}^{N} |t_i^{(\mu)} - \hat{t}_i^{(\mu)}|. \tag{2.34}$$

Opposing to classification error is classification accuracy, which stands for the probability of the input patterns being correctly classified by the neural net. This implies that classification accuracy C_{acc} can be calculated as $C_{acc} = 1 - C$ [35].

3

Description of simulations

In this chapter, I describe the series of experiments that were performed to analyze the predictive uncertainty estimation scheme described in 2.4. This chapter starts off by explaining the basic setup for the experiments. The choice of ensemble size and network structures are specified and motivated in this chapter. The chapter aims to give the readers enough information to reproduce my results. In general, these analyses are carried out by deconstructing the methods to its components. Each component were analyzed individually in terms of their strength and contribution to the scheme in it's entirety.

3.1 Datasets and experiments setup

As specified in section 1.1, this thesis is about analyzing the scheme proposed by [25] using the image-classification-problem of recognizing handwritten digits using the MNIST dataset. Therefore the training samples from MNIST were used to train the neural-networks. Apart from using the 10000 testing samples from MNIST, a separate dataset consisting of about 1000 self-collected handwritten digits were used in our experiments.

For most of our experiments, ensembles of size 20 were used. This is because Lakshminarayanan et al. [25] observed that their measurements converge on large ensemble sizes in their experiments. The said convergence happens around the size of 20. I also observe that increasing ensemble size past 20 does not affect the results. If not specified, the ensembles in the simulations consisted of multilayered-Perceptrons with 3 hidden layers with 200 neurons in each layer. This is the network structure that was used by [25] in their simulations. For some of our experiments, we used ensembles of convolutional neural-networks with 2 convolutional layers with 16 neurons in the first convolutional layer, and 32 in the second layer.

Each network in the ensemble was trained using the full training set from MNIST according to the procedure described in section 2.3. The Adam optimizer was used as the training algorithm [22] since [25] also used Adam in their experiments. Since Lakshminarayanan et al. [25] listed adversarial training as optional, we only briefly compared the results of using adversarial training with the results without. In a

majority of the experiment, adversarial training was not used.

As stated in section 2.5, the predictive uncertainty was measured using information entropy. Classification error was also measured in order to see the relation between entropy and classification error. In their experiments, Lakshminarayanan et al. [25] used the negative log-likelihood function (presented in eq (2.14)) to measure uncertainty. I decided to use entropy since the knowledge of the targets was not required when using entropy. While NLL does capture uncertainty [42], another reason for us to favor entropy over NLL is that NLL is not a bounded function and goes to infinity as the predicted probability of the correct class goes to 0.

3.2 Testing digits and distortions

As mentioned in the previous section, two testing datasets were used in our experiments. The first one being the testing samples that are apart of the MNIST dataset; the second one is a dataset of 1000 self-collected, bilevel images, hereby referred to as the CTHMNIST dataset in this report. These digits were mainly collected from bachelor students of the IT-student union division of Chalmers university of technology. There are two reasons for creating the CTHMNIST dataset:

1. It allowed us to evaluate how the method proposed by Lakshminarayanan et al. performs on out-of-distribution inputs that are similar to the inputs in MNIST.
2. Since the original images were available to us, it was easier to apply different types of distortions to them without having to upscale the images, since scaling up an image and then scaling down the image doesn't guarantee that the same image is reproduced, and may produce additional distortions through artifacts from the scaling algorithm algorithms [3].

Lakshminarayanan et al. [25] performed some experiments with out-of-distribution inputs using the NotMNIST dataset[32] as input to classifiers trained on the MNIST dataset. We opted to use the CTHMNIST dataset since the inputs are much closer to the MNIST dataset versus the NotMNIST dataset, which consists of letters.

Initial experiments involving only ensembles with a single member were performed on CTHMNIST to produce initial comparisons between attributes of CTHMNIST and the MNIST dataset. In the next few experiments, different distortions were applied to the digits in CTHMNIST to test the performance of [25] with distorted input. Mainly two types of distortion were used, namely change of line-thickness, and salt and pepper noise. The original CTHMNIST digits are first downscaled and fitted to a (200×200) pixels box with a padding of 40 pixels applied using similar methods described in [50] before applying the distortions. The digits were then downscaled to match the (28) dimension of the MNIST images before they were fed to the neural-net classifiers.

3.2.1 Line thickness

The line-thickness of handwritten digits depends on the thickness of the pen (or brush) that was used to write these, and the real-life dimensions (sizes). Therefore, neural-net classifiers trained on the MNIST digits might fail to recognize handwritten digits that are not part of the MNIST dataset simply due to the digits having a different line thickness [35].

In these experiments, the line thickness of the CTHMNIST digits were calculated and adjusted using the algorithm described by Kozielski et al. [23]. We applied the algorithm on the images individually so that we ended up with 29 datasets of distorted CTHMNIST digits with the line thickness adjusted to target values ranging from 2-30. These mutated digits were fed to the neural-net classifiers to measure how the classification error and entropy changed as a function of Line thickness.

A detailed description of the algorithm by [23] can be found in Appendix A.1. Additionally, we also measured the classification error and entropy of ensembles consisting only of one member network to compare the ensemble approach to models not using ensembles in terms of performance and predictive uncertainty estimation.

3.2.2 Salt and Pepper Noise

The next type of distortion we explore is salt and pepper noise. We define salt and pepper noise as a distortion method where the binary values of randomly selected pixels flipped using a binary NOT operation. Since CTHMNIST consists of greyscale images due to the interpolation algorithms used for downscaling, the selected none zero pixels are set to 0, and the selected zero (white pixels) are set to 255.

Salt and pepper noise were applied in two ways. First, the CTHMNIST digits were adjusted to a line thickness with the lowest classification error, the salt and pepper noise were then applied to the adjusted images by randomly selecting non zero pixels and set them to zero. Then once again from the images of optimal line thickness, we do the opposite by randomly selecting zero pixels and set them to 255.

3.3 Predictive uncertainty and incorrect decisions

In these last few experiments, the relation between entropy and incorrect decision making in deep learning models was explored. An example of incorrect decision making, in this case is to classify an image of a 4 as 9. While this example is pretty harmless, incorrect decision making in other areas might have horrible consequences.

These last few experiments were performed by classifying the images using trained deep learning classifiers. Then the correctly classified images were separated from

the incorrectly classified images. Finally, the entropy of the outputs for both cases was evaluated.

This experiment is performed using the MNIST test set, CTHMNIST digits without applying line thickness adjustments, and CTHMNIST adjusted to the line thickness with minimal classification error. Once again, the results produced by deep learning models with only one neural-net classifier were compared to those produced by ensembles. This is done using both multi-layer perceptrons and convolutional-nets. Additionally, we also briefly analyzed the impact of using adversarial training and ensembles using member voting as prediction scheme, as well as looking at the regions of high average entropy using only ensembles of perceptrons.

3.4 Implementation details

The experiments were implemented using python. This is because python as a programming language provides convenient libraries such as SciPy [8] and NumPy [48, 8] that are useful for high level numerical and linear algebraic operations similar to that of MatLab. Available in python is also Matplotlib, which was used to visualize the results.

Most importantly, the deep-learning library Keras is mainly a python library[9]. Keras provides many useful tools for building and training neural-network based models and algorithms. Keras also makes use of Google's Tensorflow as a backend [16, 1]. The models used were implemented using Keras functional API.

4

Results and Discussions

Dataset	Mean LT (px)	LT Variance (px)	Mean Accuracy	Mean entropy
MNIST test	1.48	0.126	0.98	0.024
CTHMNIST	1.38	0.21	0.87	0.106

Table 4.1: Comparisons of basic attributes of CTHMNIST and the MNIST test set. The line-thicknesses (LT) are calculated using the methods described by [23] on 28 by 28 images. The CTHMNIST digits are preprocessed, only using the procedure described in [50]. The mean entropy is calculated by averaging the entropy of each digit that were fed to the network. Both the mean accuracy and mean entropy in this table are calculated by averaging the results over 20 independent trials.

4.1 Comparison of MNIST and CTHMNIST

Table 4.1 shows the calculated line-thickness of both the MNIST test set and CTHMNIST digits in their 28-by-28 dimensions. Without applying any line-thickness manipulation or normalization, the results in table 4.1 demonstrate the difference in some attributes. At least from what can be observed, based on the lower accuracy and higher entropy is that the digits in the CTHMNIST dataset are harder to classify for a single convolutional net trained using MNIST. The higher mean entropy observed here vaguely indicates that there is a correlation between entropy and classification error.

Samples of the numbers 6,9,4, and 7 from both datasets can seen in figure 4.1. From what we can see, there are no clear differences between these samples, though we do see that the MNIST samples in fig 4.1 contain more greyscale pixels than those from CTHMNIST.

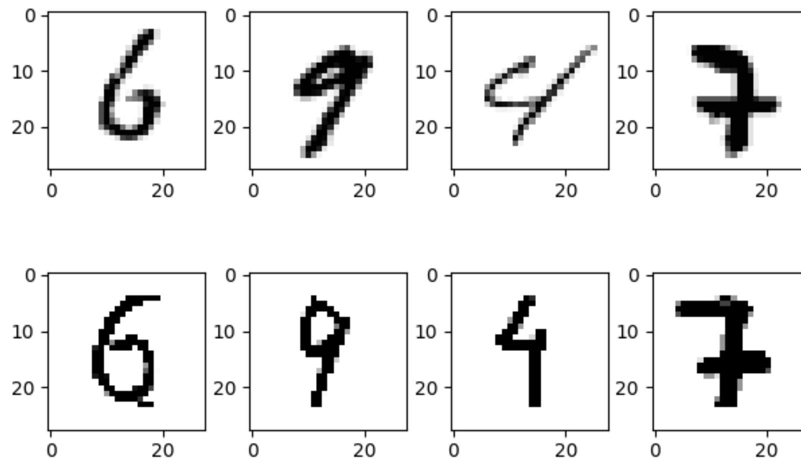

Figure 4.1: Samples of both the MNIST test set and CTHMNIST. The top consists of samples from the MNIST test set, and the bottom row consists of samples from CTHMNIST of the same numbers.

4.2 Distortions using line-thickness

Figure 4.2 shows the classification error and average information entropy (entropy averaged over the input) of ensembles in the size of 5,20,50, and 100 are plotted as a function of line-thickness. We observe that increasing the ensemble size past that of 20 does not significantly improve nor deteriorate classification error. Rather, the metric seems to have converged as the ensemble size were increased past 20. When looking at the plot of entropy (right panel of fig. 4.2), we observe that entropy seems to be increasing with ensemble size. However, just like in the case of classification error, the increase in entropy as a function of ensemble size seems to have converged when the ensemble sizes goes past the size of 20. These results are consistent with the observations reported in ref. [25].

The resulting images of line-thickness manipulations using [23] can be seen in figure 4.3. From this figure, we observe that while the image of 4 continues to be fairly recognizable at the line-thickness of 30, both 8 and 9 have started to lose their shapes at the same line-thickness as 4. From the results in figure 4.3, we can see that both 8 and 9 were starting to lose their characteristic loops at high line-thickness. Incidentally, we can also observe that the independent dots in the images have started to merge with the digits at a certain line-thickness.

The previously described observations with regards to the convergence of our metrics in terms of ensemble size, are also observed in the case of convolutional net ensembles, as seen in figure 4.4. A common observation in both 4.2 and 4.4 is that

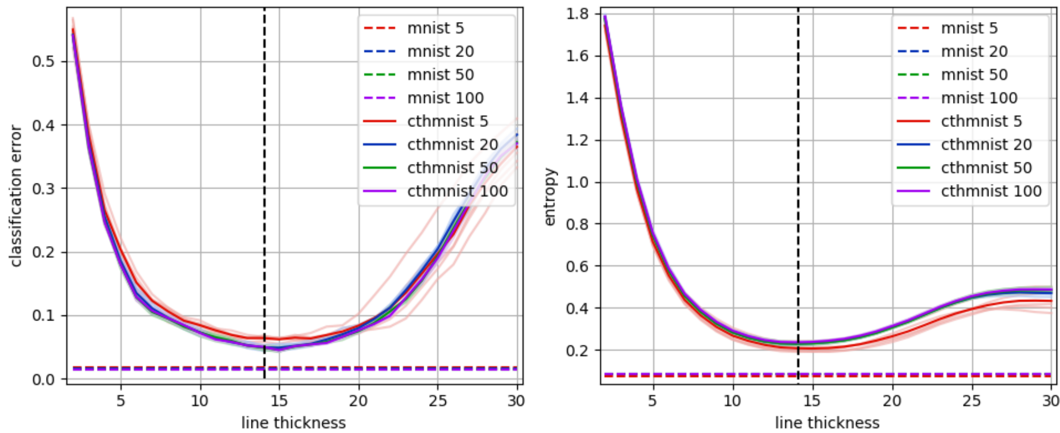

Figure 4.2: The classification error and entropy from an ensemble consisting of multi-layered perceptrons of size 5, 20, 50, and 100. The left panel plots the classification error as a function of line-thickness. The right panel plots the average entropy over the mutated CTHMNIST dataset as a function of line-thickness. These results are produced over 5 independent trials. Here the darker lines represent the average over the runs, while the lighter lines show the different fluctuations from each run. The horizontal dashed lines are the average classification error and average entropy respectively over the trials for the MNIST test set, and the vertical black dashed line is an approximated average line-thickness of the MNIST test set scaled up to that of 280-by-280.

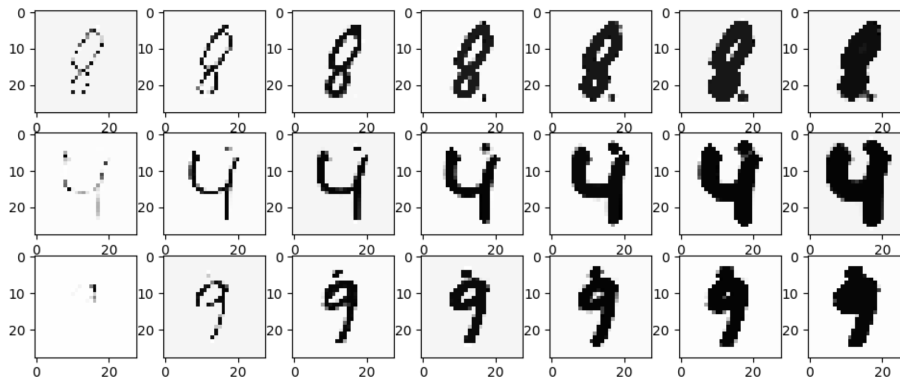

Figure 4.3: The change of sample images of an 8,4 and a 9 over line-thicknesses. The columns are ordered by line-thickness. Starting from the left most columns, the figure illustrates these same images adjusted to the line-thicknesses in the order 2, 5, 10, 15, 20, 25, and 30.

there seems to be a strong correlation between mean entropy and classification error, as both of these metrics seem to increase and decrease in similar ways. Interestingly when comparing 4.4 to 4.2, we can see that the classification seems to form a plateau over the line-thicknesses between 10 and 20.

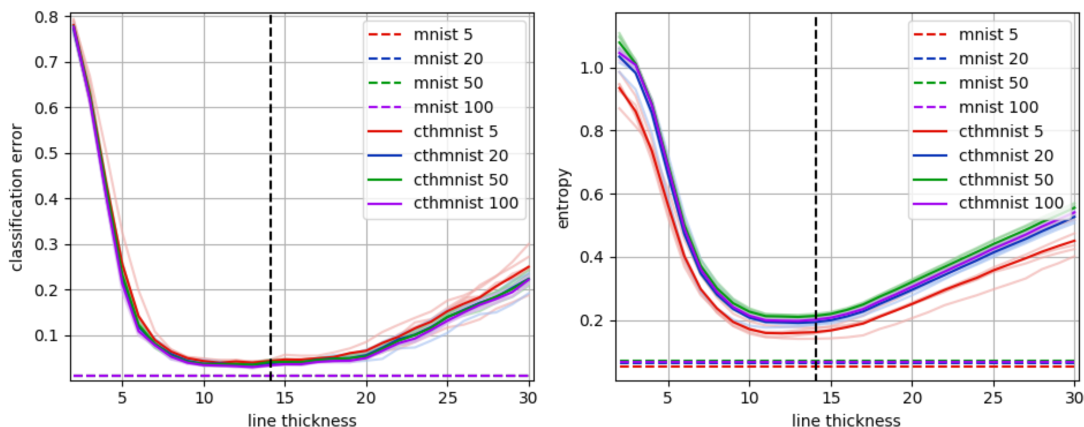

Figure 4.4: Classification error and average entropy of ensembles consisting of convolutional nets. This figure illustrates the results from running the same experiment that produces 4.2 using convolutional nets instead of multi-layer perceptrons. The rest of the settings for this figure remains the same as 4.2.

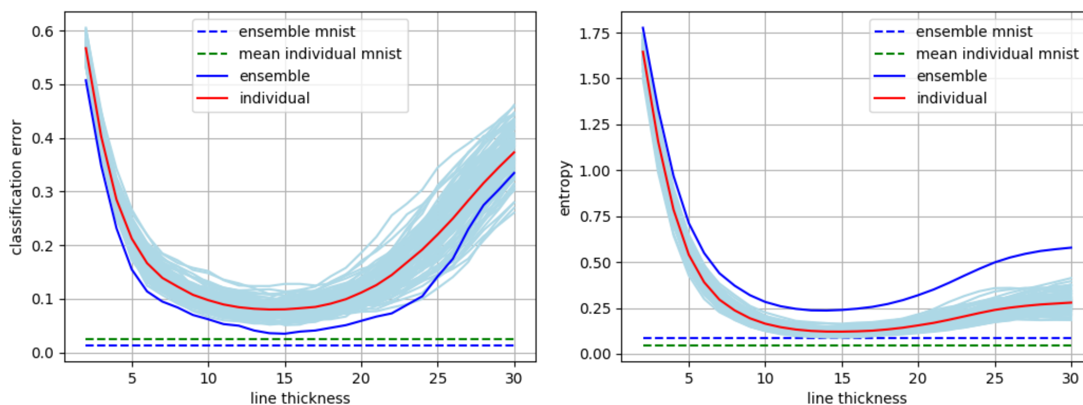

Figure 4.5: Performance comparison between individual Multi-layer Perceptron classifiers and an ensemble of 100 MLPs. The left panel plots the classification error as a function of line-thickness, and the right panel plots the Average entropy over the CTHMNIST dataset. In both panels, light blue colored lines represent the results of individual MLP classifiers over 100 trials. This is compared to the darker blue line, which corresponds to the results produced by a size 100 ensemble during 1 single trial. The red lines represent the average performance of each individual ANN at each line-thickness.

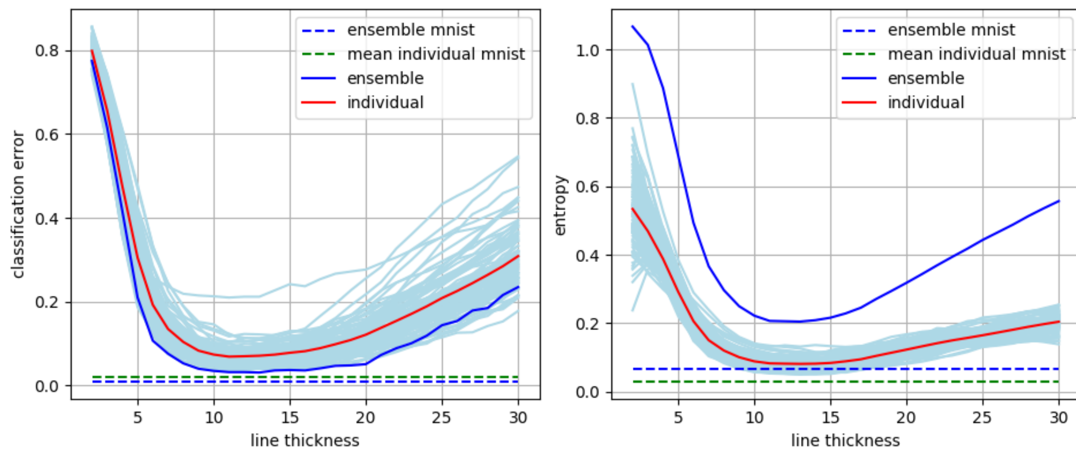

Figure 4.6: Same experiments as figure 4.5, produced with CNNs and ensemble of CNNs instead MLPs.

We also compared the behavior of individual neural-network models (i.e., Ensemble size of 1). The figures 4.5 and 4.6 illustrate a comparison between the performance of individual networks of a size $K = 100$ ensemble and the ensemble as a whole. See the respective figure captions for a more detailed description of the content in these figures. The main difference between these two is the usage of multi-layered perceptron in the first one, and convolutional neural nets in the latter.

The results from the experiments where adversarial training and voting were used can be found in the appendix. With respect to line-thickness, neither of these methods had any significant impact on the task at hand in our experiments. The results remain almost the same. In the case of adversarial training, this is in line with the discovery by Lakshminarayanan et al. [25].

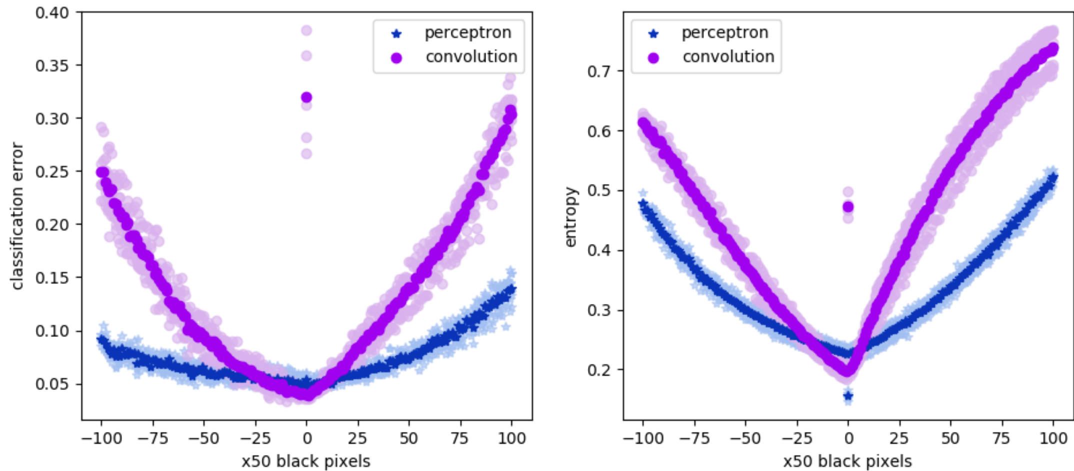

Figure 4.7: Classification error and entropy produced by 2 ensembles over 5 trials each with the inputs distorted with salt and pepper noise. One ensemble consists of MLPs, and the second consists of CNNs. The noise is applied to the digits adjusted to that of line-thickness with the least classification error. Based previous observed results from figure 4.3, this line-thickness is 14. The noise is applied by first selecting a number of black pixels, turning them white, and then from the unmodified image, choosing a number of white pixels turning them black. Each step on the x-axis corresponds to a multiple of 50 pixels.

4.3 Salt and pepper noise.

For our next few experiments, we do not compare ensemble sizes. All of these experiments were be performed using the standard ensemble size of $K = 20$. The results for salt and pepper noise experiments can be found in figure 4.7. Figure 4.8 shows how a picture of number 8 changes with varying degrees of salt and pepper noise. See figure captions for more information on the figures.

The strong correlation between mean classification error and entropy was still observable in these experiments. We do note that the steeper curves in the figures, especially in the case of convolutional nets. This could be due to the salt and pepper noise messing up with the convolution operation in a way, or that convolutional nets are more sensitive to this kind of distortion in general.

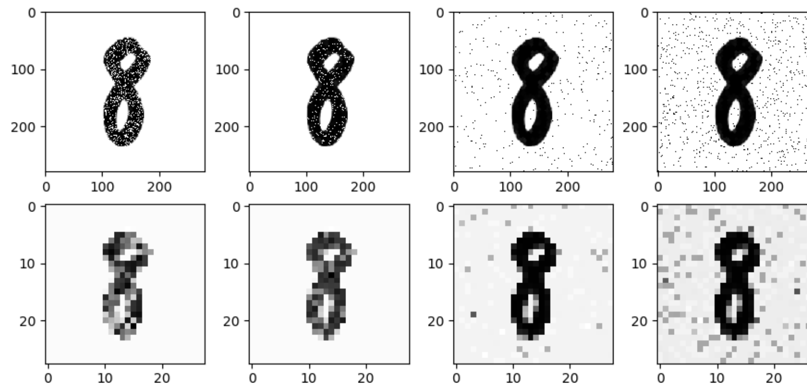

Figure 4.8: Images of an 8 Distorted with salt and pepper noise applied the same way it is described in figure 4.7. The first row is the 280-by-280 sized image of the digits just after the noise is applied. Bottom row images are the results of scaling down the images to 28-by-28, which is the images given to NN classifiers. The leftmost and rightmost columns correspond to removing resp. adding 2000 black pixels, and the middle-left and middle-right columns correspond to removing resp. adding 800 black pixels.

4.4 Predictive uncertainty and incorrect decisions

The first few experiments were performed using only a single neural-network model. Figure 4.9 shows the results for multilayered perceptrons, and figure 4.10 shows the results for convolutional nets.

The ensemble-based solutions can be found in 4.11 and 4.12. More results involving voting and adversarial training can be found in Appendix A.2.

In both of these cases, we can observe a peek around the entropy of 0 for correct classification in the histograms. While the entropy distribution for incorrectly classified inputs is less concentrated around 0, for the cases of single neural-networks, the peek around 0 still exists. This is not the case for ensembles, where we can see in all four figures that the entropy distribution is concentrated on larger values.

In general, these results indicate that high entropy would likely mean that the prediction might be wrong in both cases of ensembles and non-ensemble based models. However, the entropy distributions produced by an ensemble clearly shows that the two different cases are distinct in terms of uncertainty.

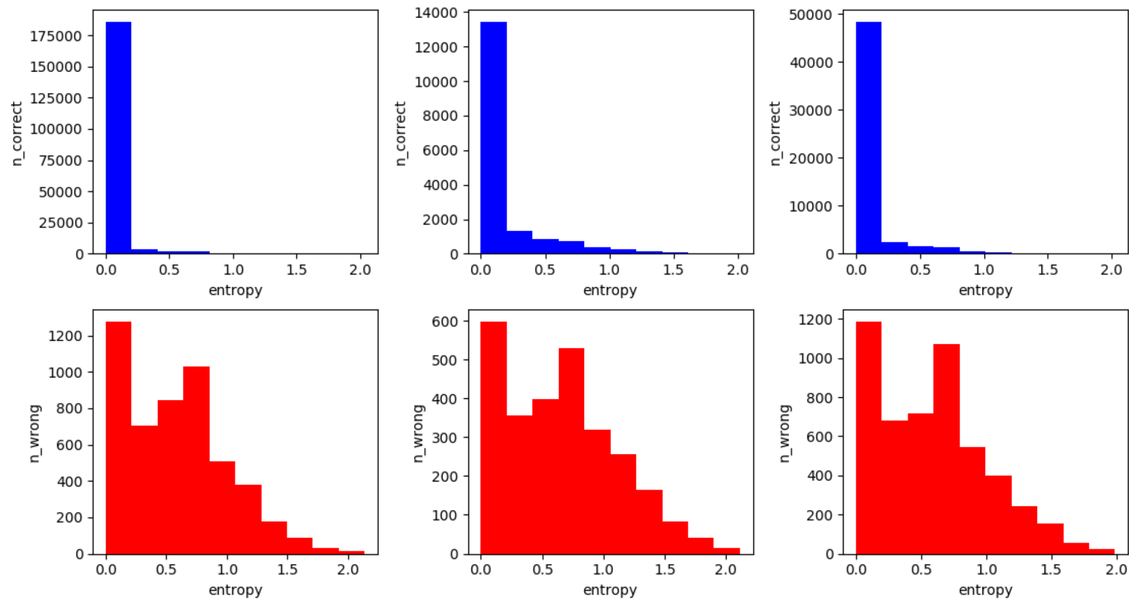

Figure 4.9: Histogram of Information entropy distribution of correctly and incorrectly classified data points. The blue histograms correspond to correctly classified input, and the red ones are incorrectly classified inputs. These values are produced by using a single multi-layer perceptron classifier. The left-most column is produced using the MNIST test-set, center column CTHMNIST without any adjustments to line-thickness, and the right one is produced using the CTHMNIST digits adjusted to the vicinity of optimal line-thickness, which ranges from 13-15.

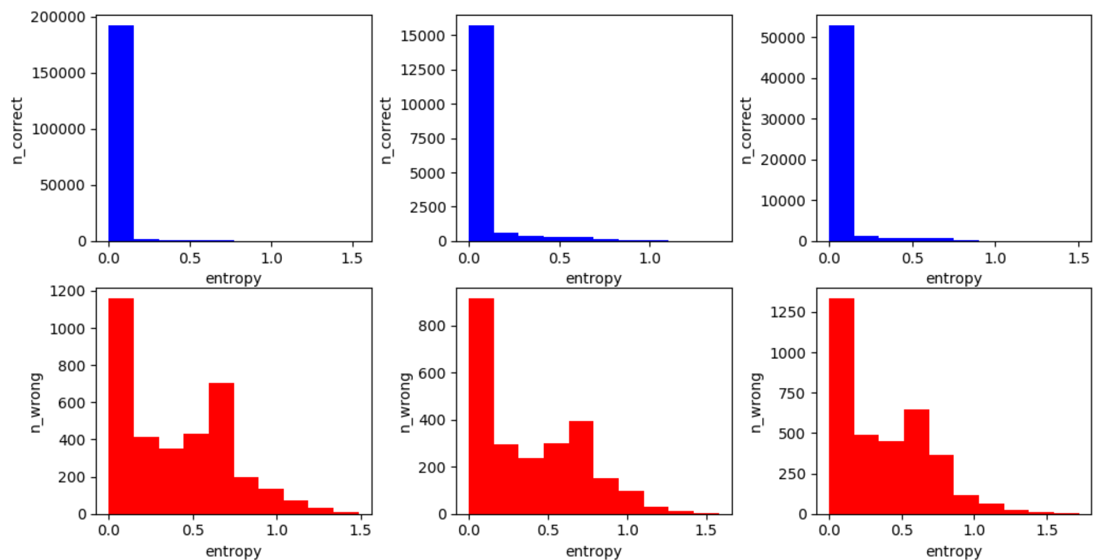

Figure 4.10: Same histograms as in figure 4.9. This one is produced using a single convolutional neural-network trained using MNIST.

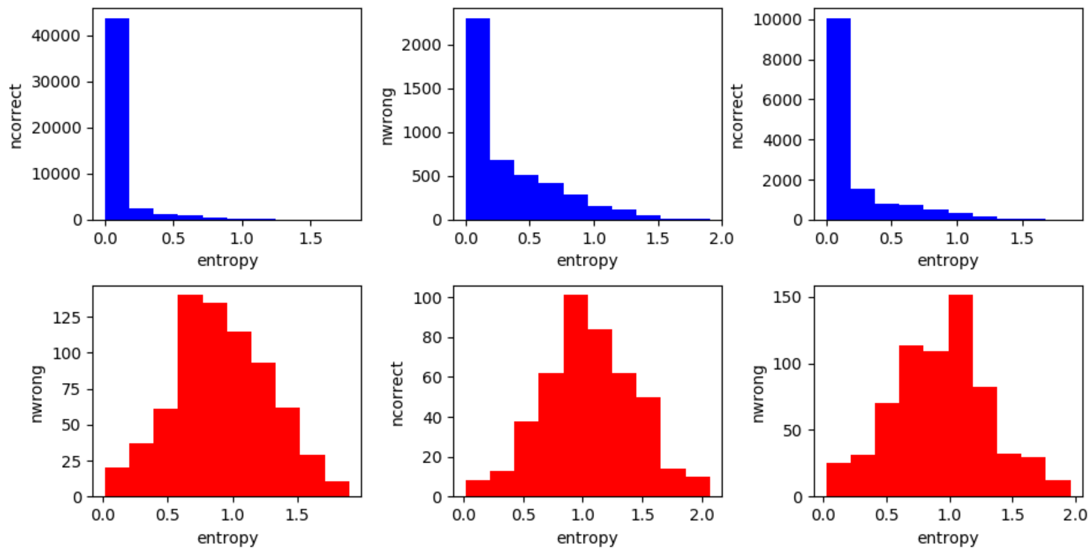

Figure 4.11: The experiments from figure 4.9 and 4.11, but performed using an ensemble of 20 multi-layer perceptrons.

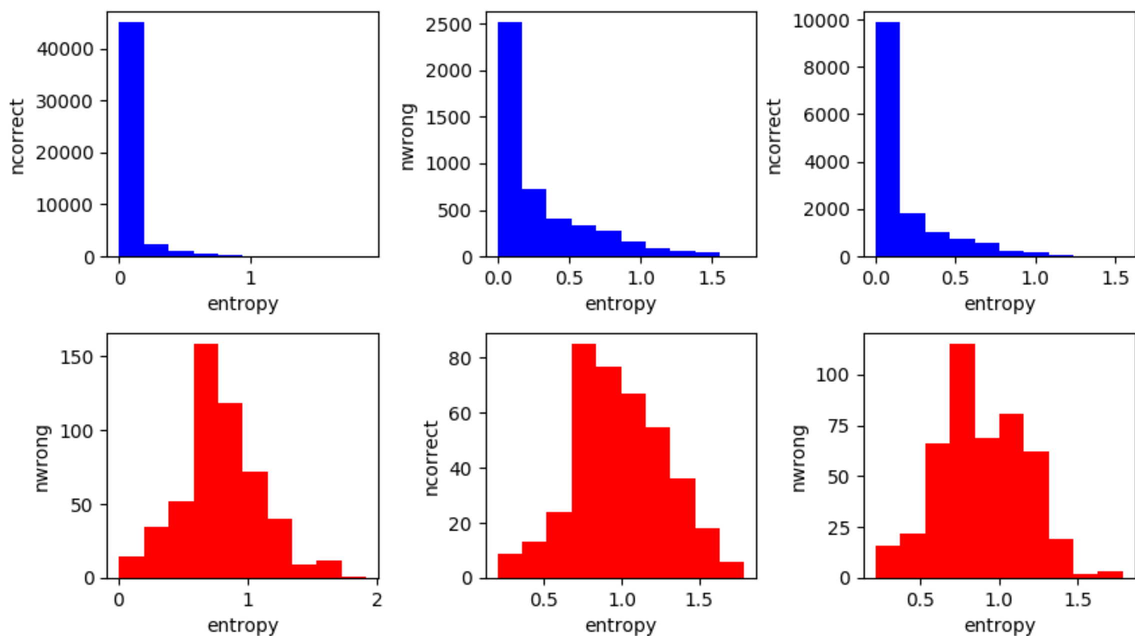

Figure 4.12: The experiments from figure 4.9 and 4.11, but performed using an ensemble of 20 CNNs.

5

Summary and conclusions

The main goal of this thesis is to understand and analyze a method for estimating predictive uncertainty in neural-networks. This method was proposed by Lakshminarayanan et al. [25] and involves using an ensemble of neural-networks. The neural-networks are trained using energy functions with specific properties. Optionally, adversarial training can be used to improve the performance of the ensemble in terms of classification error.

The proposed method is analyzed by deconstructing it into its main components. The analyses were carried out using classification of handwritten digits using the MNIST dataset as training data set. We focused our analysis on out-of-distribution inputs and distorted inputs. This is done by using both the MNIST test set and a separate dataset of handwritten digits collected from IT-bachelor students, some Ph.d. students and researchers from Chalmers. We used change of line-thickness and salt and pepper noise as the two distortion types for our experiments. The predictive uncertainty is measured using classification error and information entropy.

Throughout our experiments, we observed that the classification error gets worse as the inputs diverge from that of the MNIST. We also found that entropy also expresses this behavior, which indicates that average entropy is strongly related to classification error. This is observed both with ensembles and a single neural-network. In general, however, ensemble solutions generate higher entropy values regardless of whether voting or prediction averaging is used as ensemble-prediction scheme.

While low entropy usually indicates correct predictions in our experiments regardless of whether or not the classifier is an ensemble, we observed that the output of single neural-network classifiers have low entropy levels even when the prediction is incorrect. This is not true when using ensembles, where the outputs generally express higher entropy values when the prediction is incorrect. This goes to show that the uncertainty measurements from ensembles are much better suited for forecasting incorrect predictions when it comes to classifying handwritten digits.

To summarize and conclude this report, we demonstrated that the proposed method for estimating the predictive uncertainty in neural-networks classifiers does indeed produce well-calibrated results in this area. The major strength of this model is

the use of ensembles, which we found to be the most important component in this scheme. This thesis demonstrated that uncertainty estimations from ensembles are more satisfactory than that of single neural-networks. This comes with the strength of ensembles being better at making accurate decisions overall in this problem.

5.1 Future work

While the usage of ensembles proved to be more effective when it comes to making predictions and capturing predictive uncertainty, ensemble based models are still resource-heavy in terms of computational time and memory. Worth exploring is to find a model that behaves similar to ensembles but is computationally cheaper to use.

As mentioned before in sect 1.2, the usage of Bayesian networks is the more streamline approach to estimating uncertainty. However, it was argued in [25] that these methods still have shortcomings when it comes to implementation and training. It is still interesting to compare the performance of the ensemble approach proposed in [25] to that of Bayesian neural-networks.

Another future research could be the application of these methods in detecting different types of uncertainty, and the cause of these. This can be done while also comparing the ensemble approach to that of the aforementioned Bayesian neural nets in the same regard.

Bibliography

[1] Martın Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. "Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning". In: *Nature biotechnology* 33.8 (2015), p. 831.

[3] A. Amanatiadis and I. Andreadis. "Performance evaluation techniques for image scaling algorithms". In: *2008 IEEE International Workshop on Imaging Systems and Techniques*. Sept. 2008, pp. 114–118. DOI: 10.1109/IST.2008.4659952.

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. "Concrete problems in AI safety". In: *arXiv preprint arXiv:1606.06565* (2016).

[5] Oscar Bark, Andreas Grigoriadis, Jan Pettersson, Victor Risne, Adele Siitova, and Henry Yang. "A deep learning approach for identifying sarcasm in text". Bachelor's Thesis. 2017.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (2016).

[7] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[8] Eli Bressert. *SciPy and NumPy: an overview for developers.* " O'Reilly Media, Inc.", 2012.

[9] Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek.* MITP-Verlags GmbH & Co. KG, 2018.

[10] Terrance DeVries and Graham W Taylor. "Learning Confidence for Out-of-Distribution Detection in Neural Networks". In: *arXiv preprint arXiv:1802.04865* (2018).

[11] Cicero Dos Santos and Maira Gatti. "Deep convolutional neural networks for sentiment analysis of short texts". In: *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. 2014, pp. 69–78.

[12] Yarin Gal and Zoubin Ghahramani. "Dropout as a bayesian approximation: Representing model uncertainty in deep learning". In: *international conference on machine learning*. 2016, pp. 1050–1059.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[15] Alex Graves. "Practical variational inference for neural networks". In: *Advances in neural information processing systems*. 2011, pp. 2348–2356.

[16] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[17] Lars Kai Hansen and Peter Salamon. "Neural network ensembles". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 10 (1990), pp. 993–1001.

[18] David Harris and Sarah Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[19] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[20] John J Hopfield. "Neurons with graded response have collective computational properties like those of two-state neurons". In: *Proceedings of the national academy of sciences* 81.10 (1984), pp. 3088–3092.

[21] Edwin T Jaynes. "Information theory and statistical mechanics". In: *Physical review* 106.4 (1957), p. 620.

[22] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[23] Michal Kozielski, Jens Forster, and Hermann Ney. "Moment-based image normalization for handwritten text recognition". In: *Frontiers in Handwriting Recognition (ICFHR), 2012 International Conference on*. IEEE. 2012, pp. 256–261.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[25] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. “Simple and scalable predictive uncertainty estimation using deep ensembles”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6402–6413.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[28] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann. 1988, pp. 21–28.

[29] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David Crandall, and Dhruv Batra. “Why M heads are better than one: Training a diverse ensemble of deep networks”. In: *arXiv preprint arXiv:1511.06314* (2015).

[30] Hui Li, Xuesong Wang, and Shifei Ding. “Research and development of neural network ensembles: a survey”. In: *Artificial Intelligence Review* 49.4 (2018), pp. 455–479.

[31] Daniel Lowd and Christopher Meek. “Adversarial learning”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM. 2005, pp. 641–647.

[32] *Machine Learning, etc: notMNIST dataset*. `http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html`. Written and published by Yaroslav Bulatov in september 2011.

[33] Amit Mandelbaum and Daphna Weinshall. “Distance-based Confidence Score for Neural Network Classifiers”. In: *arXiv preprint arXiv:1709.09844* (2017).

[34] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: `10.1007/BF02478259`. URL: `https://doi.org/10.1007/BF02478259`.

[35] B. Mehlig. “Artificial Neural Networks”. In: *CoRR* abs/1901.05639 (2019). arXiv: `1901.05639`. URL: `http://arxiv.org/abs/1901.05639`.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[37] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. “Deep learning for healthcare: review, opportunities and challenges”. In: *Briefings in bioinformatics* 19.6 (2017), pp. 1236–1246.

[38] Radford M Neal. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media, 2012.

[39] Anh Nguyen, Jason Yosinski, and Jeff Clune. “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2015, pp. 427–436.

[40] Soujanya Poria, Erik Cambria, Devamanyu Hazarika, and Prateek Vij. “A deeper look into sarcastic tweets using deep convolutional neural networks”. In: *arXiv preprint arXiv:1610.08815* (2016).

[41] Tomáš Ptáček, Ivan Habernal, and Jun Hong. “Sarcasm detection on czech and english twitter”. In: *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers.* 2014, pp. 213–223.

[42] Joaquin Quinonero-Candela, Carl Edward Rasmussen, Fabian Sinz, Olivier Bousquet, and Bernhard Schölkopf. “Evaluating predictive uncertainty challenge”. In: *Machine Learning Challenges Workshop.* Springer. 2005, pp. 1–27.

[43] Simon Rogers and Mark Girolami. *A first course in machine learning.* Chapman and Hall/CRC, 2016.

[44] Claude Elwood Shannon. “A mathematical theory of communication”. In: *Bell system technical journal* 27.3 (1948), pp. 379–423.

[45] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), p. 484.

[46] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[47] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. “Deeptest: Automated testing of deep-neural-network-driven autonomous cars”. In: *Proceedings of the 40th international conference on software engineering.* ACM. 2018, pp. 303–314.

[48] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. “The NumPy array: a structure for efficient numerical computation”. In: *Computing in Science & Engineering* 13.2 (2011), p. 22.

[49] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. “Starcraft ii: A new challenge for reinforcement learning”. In: *arXiv preprint arXiv:1708.04782* (2017).

[50] Christopher J.C. Burges Yann LeCun Corinna Cortes. *MNIST Handwritten digits Database.* `http://yann.lecun.com/exdb/mnist/index.html`. 1999.

[51] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. “Ensembling neural networks: many could be better than all”. In: *Artificial intelligence* 137.1-2 (2002), pp. 239–263.

A

Appendix 1

A.1 Algorithm for modifying the line thickness

Let $I(x, y)$ denote a two-dimensional image and a structuring element of radius r. Kozielski et al. [23] described the operation of making the lines in the image thicker as a morphological dilation. A dilated image $I'(x, y)$ fulfills is calculated using eq. (A.1)

$$I'(x, y) = \max_{r_x, r_y : d(r_x, r_y) < r} I(x + r_x, y + r_y) \text{ for } r \geq 0. \tag{A.1}$$

If $r \leq 0$, then the operation becomes that of a thinning operation that is a variation of morphological erosion. Described in eq (A.2)

$$I'(x, y) = \min_{r_x, r_y : d(r_x, r_y) < r} I(x + r_x, y + r_y) \text{ for } r \geq 0. \tag{A.2}$$

Given these operations, the line thickness τ of for example an image $I(x, y)$ of a handwritten digit can be calculated using (A.3) where $G(I)$ denotes the image gradient of $I(x, y)$ which is defined by [23] as the difference $I_{thick} - I_{thin}$ between a thinned image I_{thin} and a thickened image I_{thick} by a structuring element of radius 1

$$\tau = 2 \frac{\sum_x \sum_y I(x, y)}{\sum_x \sum_y G(I)(x, y)}. \tag{A.3}$$

Suppose that an image $I(x, y)$ is supposed to be adjusted to a target line thickness T, then the algorithm can be described using the following steps.

1. calculate the radius of a structuring element as $r = T - \tau$ (using the current line thickness).

2. Apply dilation or erosion on the image using either eq. (A.1) or (A.2).

3. Calculate the new line thickness τ_{new} using (A.3).

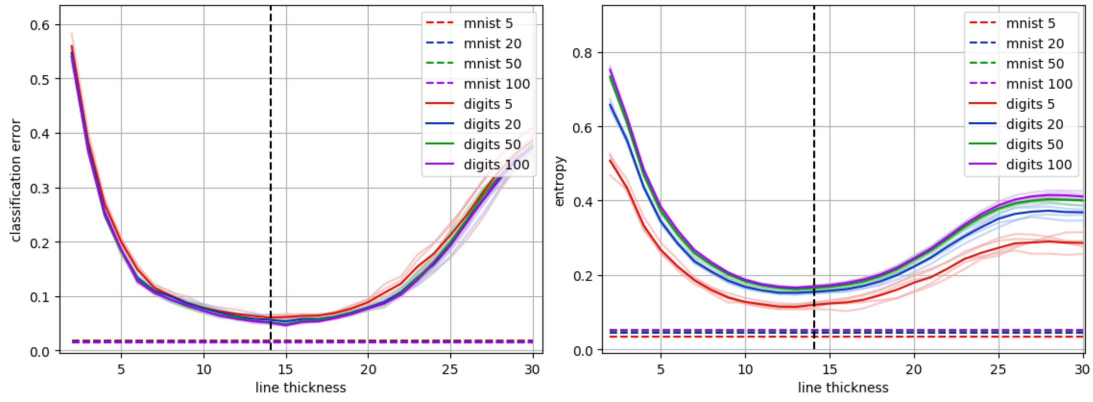

Figure A.1: The results from using an ensemble that uses member voting instead of Averaging the member predictions.

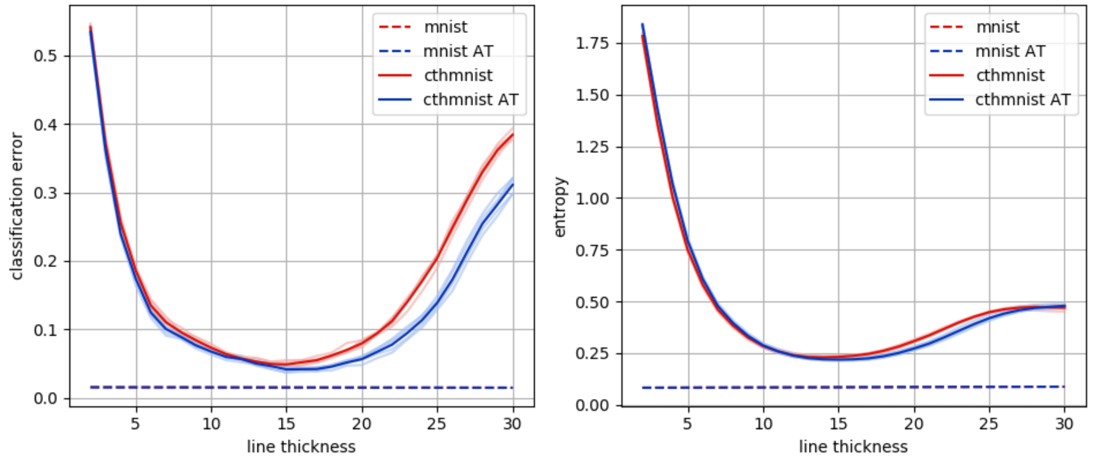

Figure A.2: Comparisons of between two size 20 ensembles of multi-layered perceptrons where one is trained using adversarial training. These results are produced over 5 independent trials. Once again, the darker colored lines are the average over these independent runs, and the lighter once are the fluctuations.

These steps are repeated until $|\tau_{new} - T| \leq \epsilon$ where ϵ stands for an error diversion. Throughout our experiments, ϵ is set to 0.9.

A.2 Additional results and discussions

See figure A.1 and A.2.

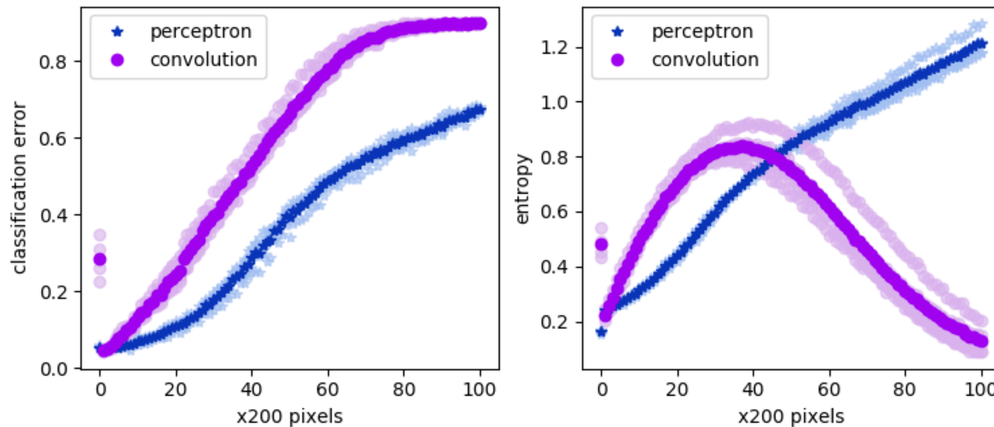

Figure A.3: Classification error and average entropy produced by an ensemble of CNNs and an ensemble of MLPs with inputs distorted with Salt and pepper noise. The noise is applied by randomly selecting a number of pixels and flipping their values. The randomly selected white pixels turn black, and the black pixels turn white. In this plot, the noise is applied to CTHMNIST digits adjusted to line thickness with the least classification error. The darker markers represent the average over the trials, and the lighter ones represent the fluctuations over the trials

A.3 Unexplainable anomaly with convolutional neural-networks

In a supplement experiment, we randomly select multiples of 200 pixels to be flipped using the operation described previously in section 3.2.2. In the process, we noticed the anomalies shown in figure A.3. We observed that as the amount of salt and pepper noise increased, we noticed that the uncertainty, however, in ensembles of convolutional neural-networks were approaching the limits of 0.

As this happens over multiples of trials, this means that the ensemble members seem to all agreeing upon a common label. However, this is clearly unexpected, as the classification error was still increasing.

A.4 Additional results with distributions

See figure A.6, A.7, A.8 and A.9, and their figure captions.

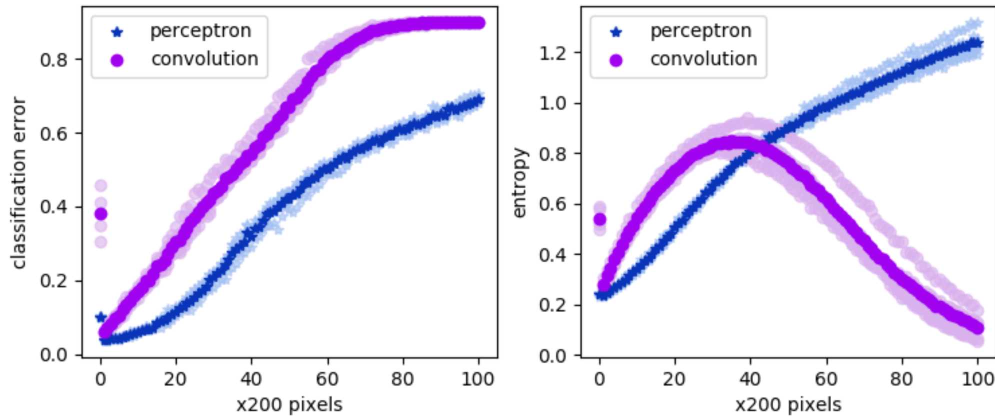

Figure A.4: Results from the same experiments as figure A.3. The difference here is that the noise is applied to the unmodified CHTMNIST digits rather then the once adjusted to optimal line thickness.

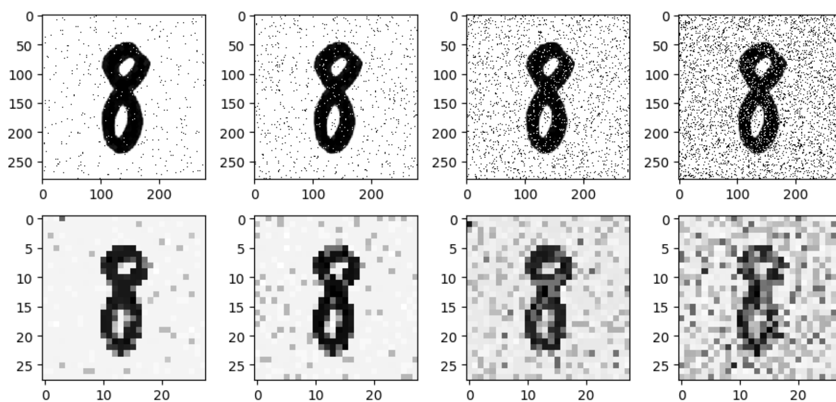

Figure A.5: Images of an 8 Distorted with applied salt and pepper noise. The noise is applied the same way as those that gave the results for figures A.3 and A.4. Sorted according to the number of pixels that were selected to flip their values. From the left, the number of pixels in each column is 1000, 2000, 5000, 10000.

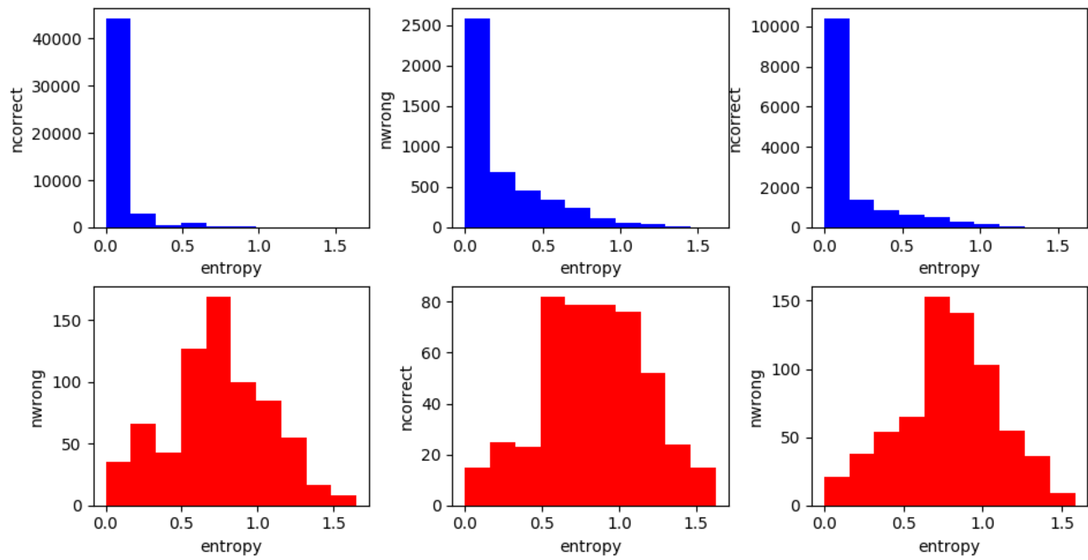

Figure A.6: Histograms from the same experiments that produced the previous figures, this time using a 20 member MLP ensemble that uses voting rather than prediction averaging.

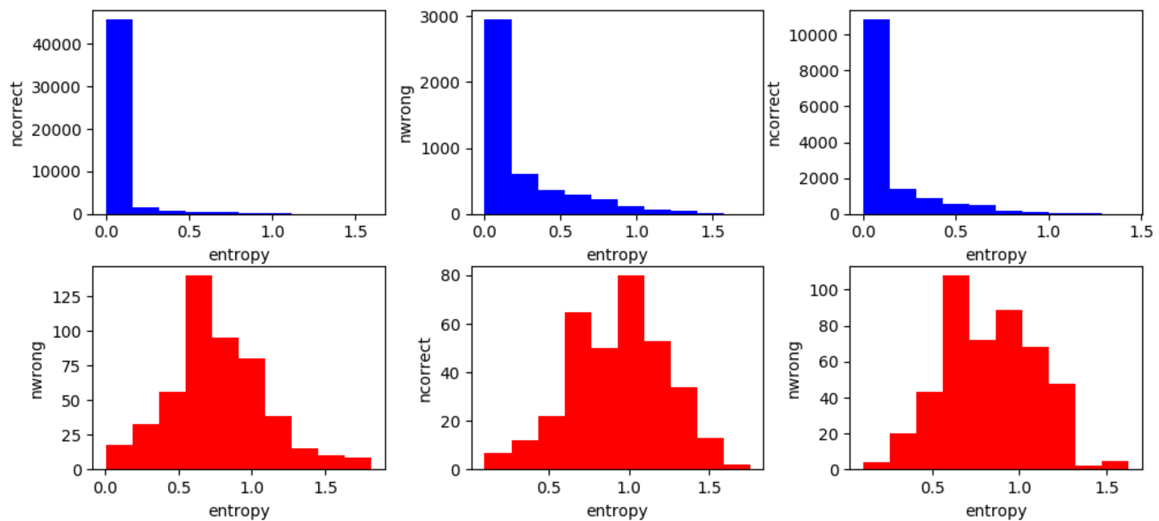

Figure A.7: Histograms of the distribution of entropy like in the previous figures. This figure is produced using ensembles of MLPs trained using adversarial training.

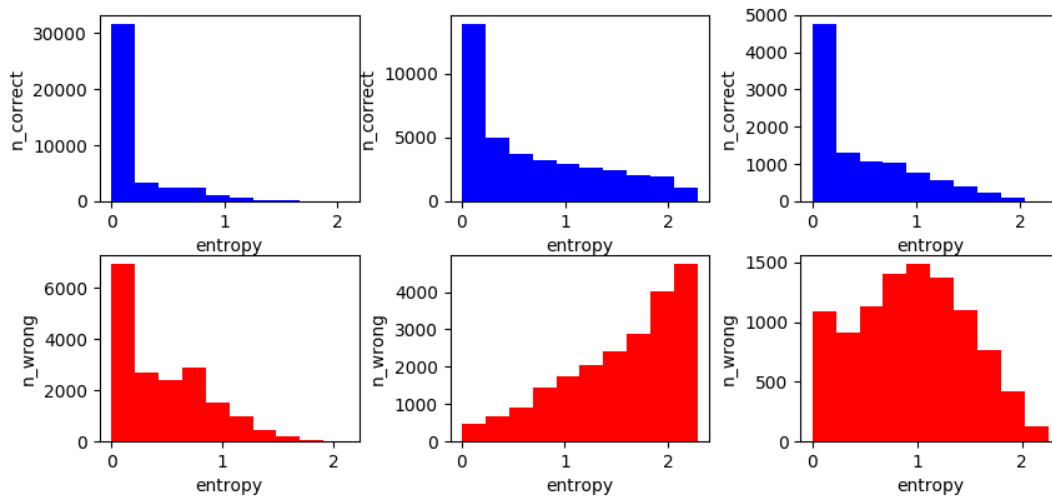

Figure A.8: Histogram of the distribution of Entropy of heavily distorted inputs. The left column represents the entropy distribution of the CTHMNIST dataset with high line thickness(between 26 and 30), the center column with low line thickness (between 2 and 6) and the third column is made with CTHMNIST with salt and pepper noise with 12000 pixels flipped in the dimensions of 280-by-280. These distributions were produced by a single MLP classifier over 20 trials.

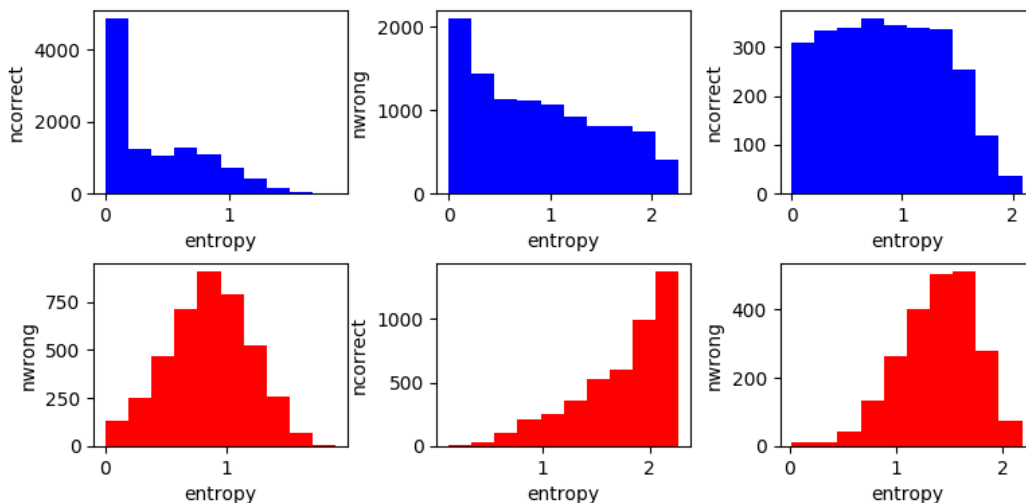

Figure A.9: Histogram using the same datasets as A.8, produced over 5 trials using an ensemble of 20 MLPs instead of a single neural network. The columns remain in the same order as they were in A.8.