Uncertainty Estimation in Deep Neural Networks for Image Classification

Master's Thesis

F.G. Drost
s2934833

August 17, 2020

Internal Supervisor: Dr. M.A. Wiering (Artificial Intelligence, University of Groningen)
External Supervisor: MSc. T. Rozeboom (ZiuZ Visual Intelligence, Gorredijk)

Artificial Intelligence
University of Groningen, The Netherlands

Acknowledgement

I would like to thank my supervisors Marco Wiering and Tiemen Rozeboom for their support and ideas during this research project. I would further like to thank the members of the ZiuZ research team, Chandler Hatton, Ruben Sluiman - Neurink, Joost Calon, Dries Pruimboom, Ioannis Giotis and Faik Karaaba for their support and insights during the time I worked on this project. Next, I would like to thank Mark Nauta and Gerrit Baarda for allowing me the opportunity to conduct this research at ZiuZ Visual Intelligence. Furthermore, I would like to thank Marie Stadel for supporting me in every single way during the project, as well as Ivar de Haan for sharing his insights with me and providing some comedic relief. I would also like to thank my parents for emotionally and financially supporting me throughout my studies. I would also like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

Abstract

For this thesis, we investigated four methods for adding uncertainty estimations to the output of a deep neural network used for image classification: Two stochastic regularisation techniques, Dropout and Batch Normalisation, an Ensemble and a novel method named Error Output. All methods, except Error Output, perform multiple different predictions for a single example to obtain an uncertainty estimate. Error Output trains additional outputs to estimate its error. We trained and evaluated the performance of the four methods on two separate datasets. We used an additional dataset, on which the networks were not trained, to evaluate the uncertainty estimations. We found that the training time significantly increased for Ensemble, but not for the other methods. The inference time increased significantly for all methods except Error Output. Our results show that the uncertainty estimation on its own did not improve our ability to detect wrong predictions. We did, however, find that Dropout, Batch Normalisation and Ensemble, when increased inference time and memory requirements allow it, provide useful methods to lower the uncertainty of predictions. Overall, the addition of an uncertainty estimate proves useful for detecting untrained classes and the mean prediction improves prediction quality in general.

Contents

1 Introduction **1**
- 1.1 Deep Learning . . . 2
- 1.2 Model Uncertainty . . . 2
  - 1.2.1 Application of Model Uncertainty . . . 3
- 1.3 Model Uncertainty in Deep Learning . . . 5
  - 1.3.1 Bayesian Neural Networks . . . 5
- 1.4 Image Recognition . . . 6
- 1.5 Research Questions . . . 6
- 1.6 Thesis Structure . . . 7

2 Neural Networks **9**
- 2.1 Artificial Neural Networks . . . 9
  - 2.1.1 Perceptron . . . 9
  - 2.1.2 Multi-layer Perceptrons (MLPs) . . . 10
- 2.2 Supervised Learning . . . 11
  - 2.2.1 Classification . . . 11
  - 2.2.2 Regression . . . 11
  - 2.2.3 Training and Validation . . . 12
- 2.3 Likelihood and Loss . . . 12
  - 2.3.1 Classification . . . 13
  - 2.3.2 Regression . . . 14
- 2.4 Optimisation . . . 14
  - 2.4.1 Parameter Initialisation . . . 14
  - 2.4.2 Gradient Descent . . . 15
  - 2.4.3 Stochastic Gradient Descent . . . 16
  - 2.4.4 ADAM . . . 16
- 2.5 Activation Functions . . . 16
- 2.6 Regularisation . . . 18
- 2.7 Convolutional Neural Networks . . . 19
  - 2.7.1 Pooling . . . 20

3 Uncertainty Estimation for Neural Networks **21**
- 3.1 Types of Uncertainty . . . 21
- 3.2 Methods for Approximating Bayesian Models . . . 21
  - 3.2.1 The Gaussian Process . . . 22
  - 3.2.2 Stochastic Regularisation Techniques . . . 23
  - 3.2.3 Monte Carlo Dropout . . . 23
  - 3.2.4 Monte Carlo Batch Normalisation . . . 25
  - 3.2.5 Ensemble . . . 27
  - 3.2.6 From Mean and Standard Deviation to Uncertainty . . . 28

3.2.7 Addition of an Error Output . . . 29

4 Experimental Setup **31**
4.1 Datasets . . . 31
4.1.1 Messidor-2 . . . 31
4.1.2 CIFAR10 . . . 32
4.1.3 MNIST . . . 33
4.2 Networks . . . 33
4.2.1 Monte Carlo Dropout . . . 34
4.2.2 Monte Carlo Batch Normalisation . . . 35
4.2.3 Ensemble . . . 35
4.2.4 Error Output . . . 36

5 Results **39**
5.1 Model Performance . . . 39
5.2 Uncertainty . . . 40
5.2.1 Uncertainty Method 1 - Standard Deviation . . . 40
5.2.2 Uncertainty Method 2 - Probability Density Function . . . 41
5.2.3 Scatter Plots . . . 41
5.2.4 Monte Carlo Dropout . . . 42
5.2.5 Monte Carlo Batch Normalisation . . . 43
5.2.6 Ensemble . . . 45
5.2.7 Error Output . . . 47
5.3 Discussion . . . 48
5.3.1 Performance . . . 48
5.3.2 Uncertainty . . . 50

6 Conclusion **53**
6.1 Summary of Results . . . 53
6.2 Recommendations for Future Research . . . 54

Appendices

A Mean Squared Error plotted against the Probability **55**

List of Figures

2.1 The XOR problem visualised. No linearly separable line can be drawn to separate both responses. . . . 10
2.2 An illustration of how the gradient descent algorithm using the derivatives of a function can be used to follow the function downhill to a minimum (from *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio [5]) . . . 16
2.3 ReLU v/s Logistic Sigmoid . . . 18
2.4 Example of a 2-D convolution, where the output is restricted to positions where the kernel lies entirely within the image. The result is a 3 by 2 matrix of summations. (From *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio [5].) . . . 20

4.1 Examples of the images in the Messidor-2 dataset . . . 32
4.2 Examples of the ten classes in the CIFAR10 dataset . . . 32
4.3 Examples of the ten classes in the MNIST dataset . . . 33
4.4 Architecture of the VGG16 . . . 34

5.1 Scatterplots for MCDO when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2 . . . 42
5.2 Scatterplots for MCDO when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2 . . . 42
5.3 Scatterplots for MCDO when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10 . . . 43
5.4 Scatterplots for MCDO when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10 . . . 43
5.5 Scatterplots for MCBN when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2 . . . 44
5.6 Scatterplots for MCBN when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2 . . . 44
5.7 Scatterplots for MCBN when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10 . . . 44
5.8 Scatterplots for MCBN when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10 . . . 45
5.9 Scatterplots for Ensemble when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2 . . . 45
5.10 Scatterplots for Ensemble when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2 . . . 46
5.11 Scatterplots for Ensemble when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10 . . . 46
5.12 Scatterplots for Ensemble when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10 . . . 46

5.13 Scatterplots for Error Output when the uncertainty is raw output of the error nodes and the network is trained on Messidor-2 47
5.14 Scatterplots for Error Output when the uncertainty is computed via the 'true' error and the network is trained on Messidor-2 47
5.15 Scatterplots for Error Output when the uncertainty is raw output of the error nodes and the network is trained on CIFAR10 48
5.16 Scatterplots for Error Output when the uncertainty is computed via the 'true' error and the network is trained on CIFAR10 48

A.1 Effect of mean squared error on probability on Messidor-2 test set 55
A.2 Effect of mean squared error on probability on CIFAR10 test set 56

List of Tables

4.1 Division of examples in the Messidor 2 dataset 32
4.2 Division of examples in the Messidor 2 dataset splitted into a training, validation and test set. The training set is the only set normalised through augmentation. 32
4.3 Division of examples in the CIFAR10 dataset 33
4.4 Dropout Rates per layer added to VGG16 34
4.5 Overview of the different architectures used for the Ensemble, together with their respective performance on the ImageNet validation dataset 36
4.6 Final settings for each method when trained on the Messidor-2 dataset. * Each network inside the Ensemble is stopped after a different number of epochs. † Error Output is trained 3 times in succession. 37
4.7 Final settings for each method when trained on the CIFAR10 dataset. * Each network inside the Ensemble is stopped after a different number of epochs. † Error Output is trained 3 times in succession. 37

5.1 Performance when trained on the Messidor-2 dataset. * The training set caused problems with batch normalisation due to a lack of variance † Accuracy when the complete trained network was used without the MCDO or MCBN method 40
5.2 Performance when trained on the CIFAR10 dataset * Accuracy when the complete trained network was used without the MCDO or MCBN method 40

Chapter 1

Introduction

A critical part of many machine learning (ML) applications is understanding what a model does not know. An output of a network, even with a high probability, should not directly be assumed to be accurate. Many factors can cause a network to respond with a high probability to an example outside of the training set, which can even lead to fatal results [1]. If algorithms can assign an additional uncertainty estimation to their predictions, decisions that may have severe consequences might be reduced. This would allow for example physicians in medical environments to assess when the network is giving unreliable predictions. Leveraging techniques that can assign an uncertainty estimation in the application of deep learning methods to medical image analysis could accelerate acceptance of deep learning applications among clinicians and patients [2].

In computer vision existing approaches to model uncertainty include particle filtering and conditional random fields [3,4]. Deep learning is, however, often mandated to achieve state-of-the-art performance and the previously mentioned approaches to model uncertainty do not apply to deep learning. Deep learning classification models often give normalised score vectors which do not necessarily capture model uncertainty [5]. When these models are exposed to data outside of the distribution it was trained on, the network is forced to extrapolate, which can lead to unpredictable outcomes. Bayesian modelling can, however, capture uncertainty. Bayesian modeling can model two main types of uncertainty, *aleatoric* and *epistemic* uncertainty. Aleatoric uncertainty arises through noise in the observations (e.g. sensor noise). This uncertainty is inherent in the observations and therefore cannot be reduced with more observations. Epistemic uncertainty (model uncertainty) arises through uncertainty in the model parameters and the model's ignorance about which model generates the training data. This can be reduced with additional observations since more data gives a better understanding of the true model that generates the data.

In classification tasks generally aleatoric uncertainty is more important to model since it can't be corrected for with more data [6]. However, medical data almost never provides enough examples (determining what is 'enough' is a hard problem on itself since the variance of the complete data is often unknown), which keeps modelling epistemic uncertainty important as well.

Combining aleatoric and epistemic uncertainty results in predictive uncertainty (the model's confidence in its prediction) which takes into account the noise it can and cannot explain away with more data. Predictive uncertainty is both affected by aleatoric uncertainty (increasing predictive uncertainty in case of a large measurement error) and epistemic uncertainty (increasing predictive uncertainty for inputs that lie far away from the training data). Predictive uncertainty is usually obtained by sampling multiple functions from the model and corrupting them with noise. We can then calculate the variance of these functions on a fixed set of inputs which yields us the predictive uncertainty.

Bayesian machine learning uses models such as Gaussian processes, which define probability distributions over functions to learn what is the most likely (and unlikely) way to generalise from observed data. These probabilistic methods offer, through their provided uncertainty, consequentially confidence bounds which a doctor (or autonomous car) will use in their decision making. These decisions can now include the conclusion that more (diverse) data is needed to train the model, the model itself needs to

be changed or perhaps just that some caution is needed in accepting the output. When and how to draw these conclusions has been extensively studied for Bayesian machine learning [7]. Deep learning models, mostly viewed as *deterministic functions* instead of *probabilistic*, however, require us to sacrifice these conclusions, causing us to wonder whether a deep model is making a sensible prediction or whether it is just guessing at random.

However, with a few small changes, many existing deep learning models can give uncertainty information as well. This subject is of practical interest for the company ZiuZ Visual Intelligence[1], which provided support in forming this thesis. ZiuZ is currently working on the POLAR project[2], which is a collaboration between Amsterdam Medical Center, Medical Center Leeuwarden and ZiuZ Visual Intelligence. The goal of this project is to create a product that can advise doctors during a colon exam on whether a polyp is benign or premalignant. Premalignant polyps can develop into cancer in later stages. In this thesis, we will compare three existing methods to obtain uncertainty estimates from deep learning models, and also introduce a novel method.

1.1 Deep Learning

To explain the concepts of deep learning, we have to start with neural networks (NNs). Artificial NNs are inspired by how biological neurons learn and form networks. NNs are ML models which transform one or more inputs into one or more outputs through a network with one or more layers. Each layer consists of neurons which have an activation function and are connected by weights to neurons in different layers. NNs transform the input(s) through these layers to one or more outputs by propagating activations throughout the network. Standard feedforward NNs perform a sequence of non-linear transformations, but layers can also, independently of other layers, have linear element-wise activation functions. Standard NNs are usually trained by supervised learning, in which the weight parameters connecting the neurons between layers are tuned in such a fashion that the difference between the predicted output of all transformations and the ground truth on the known data is minimised. The ability of NNs to learn complex tasks has made them very successful as an ML method in a wide domain of applications. These include image [8] and speech recognition [9]. Breakthroughs on these tasks have played a major part in the boom of NN research since 2012.

A basic deep learning model can be described as an NN model consisting of many layers (possibly having different activation functions). These layers, or groups of layers, can be seen as building blocks, specialised for a certain task (e.g. detecting a certain feature). The modularity these building blocks provide allows for the creation of large varieties of compositions of these blocks. These different architectures embody the versatility of deep learning models [10]. The advent of deep learning, however, has made it even more difficult to interpret how a model arrived at its predictions. The increased size of the networks tremendously increases the number of parameters of the network, which severely hinders the understanding of the network's inner workings from an outside perspective. Elaborate architectures can be used to limit the number of parameters, but their complicated designs often hinder understanding as well. This causes the view of NNs as black boxes to remain.

1.2 Model Uncertainty

Deep learning models can be used for a wide variety of applications such as stock price predictions, cancer detection from MRI scans, navigation in autonomous vehicles or flower species classification. For an application like flower species classification, a deep neural network would be trained on a large, annotated dataset consisting of many examples of different classes of flowers. After successful training, the network

[1] ZiuZ Visual Intelligence, Gorredijk, The Netherlands

[2] POLAR stands for POLyp Artificial Recognition

should have had its parameters tuned in such a way that it captures the general features that make a certain flower species that specific species. When presented with a new example of a flower, we can expect the network to output the corresponding class with a high probability. But what happens when we present this network with an example of a cat? This is an example of *out of distribution* test data [11]. After training, the model has learnt the distribution of each flower species and is able to distinguish them from each other, but a photo of a cat would lie completely outside of the data distributions it was trained on. Since the model will always be forced to output probabilities that sum to 1 (in the case of this classification example with the commonly used Softmax activation on its last layer), we hope that the network will output an even distribution of probabilities for each output, signalling to the user that its confidence is low. However, due to the black box behaviour of a trained deep network, this is rarely what happens. A more probable result would be the model outputting a certain flower species with a rather high probability, because the features detected in the cat most closely relate to the features found in that flower species. How close (or rather far) they relate does not matter, an output shall be generated by the network.

A more desirable network would return the prediction, but also return the additional information that this new example lies outside of the data distribution it has been trained on and therefore has a lower confidence in this prediction.

Many situations can introduce uncertainty, including:

- The example to be predicted lies outside of the observed data distribution,
- Noise in the data (either in the observed data, the new data, or both), leading to *aleatoric uncertainty* (data uncertainty),
- *Uncertainty in the model parameters* (A large number of different models might explain the observed data, which causes uncertainty about which model to pick to predict with),
- *Structure uncertainty* (what model structure (for example deep learning architecture) should we use?)

Uncertainty in the model parameters and *structure uncertainty* can be combined into *epistemic uncertainty* (model uncertainty). *Data uncertainty* and *model uncertainty* can be used to induce the confidence the model has in its prediction, its *predictive uncertainty*.

1.2.1 Application of Model Uncertainty

Uncertainty information is often used in life sciences [12] and is gaining traction in other sciences such as social sciences [13]. In these areas, the importance of being able to quantify the confidence of the model is well understood.

POLAR

For systems that make decisions that affect human life, like the polyp detection in the POLAR project, the importance of understanding the confidence of the model cannot be overstated. This data gives not only insight for the practitioner using the system, but also for the developer. Recognising that the test data is far from the training data gives insight into how the model can be improved by for example augmentation or the gathering of more data.

A practitioner will not be an expert on the workings of the automatic polyp detection and recognition system. When the trained system is deployed, the practitioner will be told that this new system gained a (for example) +99% accuracy on the test data. The practitioner will not know what was included in this test data, and even if the practitioner did, will not know what features of each class the system uses to make its decision. The practitioner will therefore not have a reason to question the classifications provided by the system when the input deviates from the data distribution it was trained on. This could lead to life-threatening consequences and a loss of confidences in the solutions these systems could provide.

Autonomous vehicles

Autonomous vehicles can range from vacuum robots to rockets that can land themselves. They can be divided into two groups, a group of vehicles that is rule-based and a group of vehicles that can learn to adapt to a changing environment. Both groups can make use of machine learning. The first group might make use of feature extraction while the second group might use reinforcement learning (RL) to adapt to a constantly changing environment.

One of the most promising forms of autonomous vehicles are self-driving cars. Self-driving cars make heavy use of sensory input to map the world around them. Low-level feature extraction is used on cameras, LIDAR, RADAR or any other raw sensory inputs [14]. These features, among other things, can be used for image segmentation and object classification. The outputs of these segmentations or classifications are in turn used by higher-level decision-making algorithms. These high-level decision-making processes can again be trained through reinforcement learning or they can be expert systems relying on fixed sets of rules (e.g. yield to vehicles coming from the left).

The hierarchical structure of this system allows mistakes in lower layers to propagate up to the final decision making. For example, mud on a camera sensor can impact the feature extractors, or a new uniquely styled car might not be recognised as a car in a classifier. In the real world many new and unique situations, which the system has not been trained on, can be present. If the outputs of all parts in the system's hierarchy would give an uncertainty alongside the regular output, the high-level decision making might not take potentially dangerous actions but rather prompt the user preemptively to take over control so fatal results can be avoided [1].

Active Learning

Outside of safety, model uncertainty can also be used to improve the process of constructing a successful model. Many machine learning approaches, like deep learning, require large amounts of labelled data to generalise well. It is often the case that more complex tasks require more complex models, which in turn require more data. Gathering this data can be a long and expensive process in which experts have to manually label each example.

A possible solution for this problem could be active learning [15]. The key idea behind active learning is that a machine learning algorithm itself is allowed to choose the training data it learns from. The active learning algorithm may ask an expert to label unlabelled instances that the algorithm deems to be most informative to improve itself. The choice of which instances to deem most informative is done through an acquisition function. Many acquisition functions make use of model uncertainty. When active learning, for example, tests its predictions on unlabelled data, it might have high confidence that a part of the data fits in the data distribution it is currently trained on, but it might also have a high uncertainty for a different part of the data. This last part then gets sent to an expert who could classify it as a completely new class the model was not yet trained on, or as an existing class after which the model will expand its representation of the data distributions for those particular classes.

Reinforcement Learning

Reinforcement learning (RL) algorithms learn a task by trial and error [16]. They try a certain action, evaluates their new states and updates their decision-making process accordingly. If the action did not result in reward, or in penalty (for example a robot vacuum cleaner bumping into a wall), the algorithm will learn over time to not take those actions. This process, however, can be very time consuming, certainly if the training takes place in the real world. Too many mistakes might break the robot, or too much unnecessary exploration might wear out the physical system.

A RL system usually uses Q-value functions to try to calculate the quality of each action the agent can take at a given moment. These functions can never calculate the true quality of the actions since the sensor can only model a limited understanding of the environment and the environment can constantly change.

Recent advancements in RL make use of deep learning (resulting in deep RL) to produce impressive results in for example the playing of games [17]. These types of networks use NNs to approximate the Q-value function. Greedy search is often used to select the best action the agent can take with some probability and to otherwise explore. However, when the RL algorithm also uses uncertainty information, the agent can decide when to exploit and when to explore similarly to active learning. Additionally, uncertainty information over the Q-value function can be used to learn faster as well [11].

1.3 Model Uncertainty in Deep Learning

Above we have established some of the many uses for uncertainty estimations. However, most deep learning models do not possess the ability to offer such information.

Regression models for example output a single vector representing the mean of the data, but give no confidence in this vector. Classification models output a probability vector (often a softmax output) which should not be mistaken as model confidence. The sum of these outputs is always 1, so the model cannot output 'no confidence' in all of the possible outputs. Due to the nature of the softmax function, where the data distribution of the train set should be captured between the two ends of the softmax curve, unjustifiably high confidences can be given for points far away from the training distribution. For these new points, the system will usually extrapolate from the most outer reaches of the training distribution (these outer reaches will be the closest to the real distribution of the test set). However, these outer points of the training data have usually received a very high or very low probability due to the shape of the softmax function. Extrapolation will result in the same high probability to be (wrongfully) given to the new test points.

However, the output after passing the complete distribution through a softmax will give us uncertainty information about extrapolated results far from the training data. Deep learning models generally do not use distributions, but rather try to find the optimal point estimate for the weights in every node. Deep learning models are generally underspecified by the data and many different (but all high performing) models with differently tuned parameters can exist. Between these many different models that could explain the data lies the uncertainty that is not captured by only one, point estimate, model. Deep learning models are, however, related to a family of probabilistic models that do use probability distributions, the Gaussian process [18].

1.3.1 Bayesian Neural Networks

If one would place a probability distribution over each weight in a neural network, a Gaussian process can be recovered in the limit of infinitely many weights [19]. The most prominent weaknesses of the Gaussian Process, however, is that it suffers from cubic time complexity $O(n^3)$ (where n is the number of training examples) because of the inversion and determinant of the $n \times n$ kernel matrix [20]. This limits the scalability of the Gaussian Process and makes it unaffordable for large-scale datasets. However, model uncertainty can also be obtained when placing a distribution over a finite number of weights. These models are called *Bayesian neural networks* [21]. Various techniques using the basic ideas of Bayesian neural networks have been introduced over the years, with various degrees of success [22]. Often these models are difficult to work with and introduce many more parameters to be trained, a trade-off many members of the deep learning community are not willing to take.

A practical solution should scale well with large data, as well as complex models and should apply to the widely studied and developed deep learning architectures already available. Several methods have been introduced that satisfy these constraints, including Monte Carlo Dropout (MCDO) [11] and Monte Carlo Batch Normalisation (MCBN) [23]. These two methods make use of stochastic regularisation techniques (SRTs). SRTs are techniques for model regularisation. Deep neural networks are usually very complex models with millions of parameters. This complexity allows them to learn the underlying distribution of

almost all training data but also allows them to overfit on the same data. SRTs penalises the weight matrices of the nodes in a network as to make the model less sensitive to noise and therefore to generalise better. The process of SRTs can be exploited to gain uncertainty by alternating which weights get penalised. Each unique combination of penalised weights forms its own unique network, with therefore a unique output. The mean of multiple of these outputs can be used as the final prediction and the variance can be used as the uncertainty of that prediction.

Exploiting SRTs for uncertainty estimates are practical with large models and data. They work with existing approaches and therefore apply to a wide variety of tasks including image classification, reinforcement learning and active learning.

1.4 Image Recognition

Image recognition (also referred to as *image classification*, or broader *object recognition*) is the computer vision task of determining the category of an object in an image. One of the first applications of image classification was optical character recognition. The widely popular convolutional neural network (CNN) was first developed to aid the object recognition task of handwritten zip code recognition [24]. CNNs do not require handcrafted feature extractors, but can learn useful feature extractors themself. A CNN also allows features of an object to be anywhere in the input instead of expecting them to be at certain locations in the data that it has been trained on. This generalisation is especially useful for image recognition, where we cannot expect every object to be always placed the same in every image. The next big advance in image recognition came through the advent of (feasible) deep learning [8, 25, 26]. In 2012 AlexNet, one of the first deep NNs to utilised a deep CNN trained on GPUs, reached a much lower score on the ILSVRC2012 recognition challenge compared to the previous state-of-the-art [8]. This spawned renewed interest in the field and led to CNNs being applied to numerous tasks, such as computer-aided diagnosis [27], activity recognition [28], facial recognition [29] and automatic image annotation [30].

1.5 Research Questions

In this thesis, we investigate how an uncertainty estimation can be incorporated into existing deep learning architectures. Four methods (all using widely available techniques that require minimal adjustments to existing architectures) will be compared with regards to their performance, memory requirements and accuracy.

The problem this thesis will be examining can be framed into the following research question:

Which method provides the best uncertainty estimations in deep neural networks for the classification of images?

This question can be decomposed into the following sub-questions:

- **Which method yields the best performance?** Here, performance will be measured in total training time, the time required to classify one new image (inference time), the accuracy of the predictions, the accuracy of the uncertainty estimate, the total model complexity and the memory requirements. *Total training time* is for example important in an active learning environment where new classes or data might constantly be added. *Inference time* is important in for example autonomous vehicles where the system needs to be able to respond quickly. The *accuracy* of the predictions or uncertainty should also not suffer under the addition of the uncertainty estimation. The *model complexity* and *memory requirements* should be small enough to run inference on relative cheap/simple hardware when used in for example in a mobile application.

- **Does testing on a different image dataset clearly show uncertainty?** This will test the generalisation of the uncertainty. Ideally, we want the uncertainty estimate to be perfectly able to detect

which data classes were part of the training data and which not. All out of distribution data should therefore return a high uncertainty estimate.

1.6 Thesis Structure

The rest of the thesis is structured as follows: Chapter 2 discusses the theoretical background, training and design of neural networks. Chapter 3 discusses how we can implement uncertainty estimates for neural networks. It starts by explaining different types of uncertainty and the Bayesian approach. After this, it describes the four methods of uncertainty estimates that are the core of this thesis and draws a connection with the Bayesian approach. In Chapter 4 the experimental setup is described, including a description of all the used datasets and how the four methods were implemented. Chapter 5 shows the results and discusses their implications. Finally, Chapter 6 concludes by answering the research questions. Furthermore, we give recommendations for future research.

Chapter 2

Neural Networks

2.1 Artificial Neural Networks

An Artificial Neural Network, often simply referred to as a Neural Network (NN), is a computational model inspired by the way biological neural networks process information. An NN consists of a collection of connected nodes (also referred to as *neurons* or *units*), which loosely model the neurons in a biological brain. Like the synapses in a biological brain, the connections between nodes transmit information from one node to another. Each node can have multiple incoming and outgoing connections. During one *forward-pass* through the network, a certain node can receive one input per incoming connection and can output once to all its outgoing connections. Through an *activation function* (Section 2.5) the nodes determine the strength of these outgoing signals. A node can have a certain threshold that has to be crossed before any output signal is sent. In practice, these signals are real numbers and the activation function often is a *non-linear* function of the sum of inputs.

Each connection has a certain weight. This weight determines the strength of the signal and can be compared to the strength of connections between neurons in a biological brain. For biological neurons, the more often a connection is used, the stronger the connection gets. Similarly, for artificial neurons, changing the weights of the connections allows the network to learn certain patterns. To do so, the network considers examples and adjusts it weights until a certain *goal* (Section 2.3) is met. The weights of connections that bring the output closer to the goal will be increased, while weights that produce wrong results will be decreased.

Typically nodes are grouped in layers, with generally no connections between nodes in a certain layer but only between layers. Each layer can be designed to perform a certain task by applying specific transformations on its input. Signals travel from the first (input) layer to the last (output) layer, but might traverse the network multiple times depending on the *architecture* of the network. The architecture refers to how the initial network is designed, e.g. how many layers the network has, how these layers are designed and how they are connected.

One of the first attempts at creating an artificial NN was done by Farley and Clark [31] in 1954. They simulated a network following *Hebbian* learning. A few years prior, D.O. Hebb created this learning hypothesis based on the mechanism of neural plasticity [32]. His theory is often summarized as ”Cells that fire together, wire together”. This idea sets the basis for how NNs increase the weights of the connections that produce wanted output and decrease those that produce unwanted output.

2.1.1 Perceptron

The first binary classifier, a function which can decide whether or not an input belongs to a specific class, was developed by Rosenblatt as the *perceptron* [33]. The perceptron maps an input vector $\mathbf{x}$ to a binary

output scalar through the following function:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}, \tag{2.1}$$

where input vector $\mathbf{x}$ and weight vector $\mathbf{w}$ have real values, and b is the bias. A binary classifier could for example be used for spam detection. The input vector $\mathbf{x}$ could consist of a list of the frequency certain words are found in an email, which is used to predict the binary target value y of being classified as spam or not. A prediction $\hat{y} = 1$ could mean spam, while $\hat{y} = 0$ not spam. The weight matrix $\mathbf{w}$ (with one weight per input dimension) determines how much the frequency of each word in the input list $\mathbf{x}$ influences the outcome, while the bias b determines the magnitude of the dot product $\mathbf{w} \cdot \mathbf{x}$ that is required for positive classification.

Rosenblatt's *perceptron training rule* provides an iterative method to find the weights that produce a correct classification. For linearly separable problems (where a hyperplane exists in the input dimensions that can separate the data points of the positive and negative class), convergence is theoretically guaranteed.

However, a single-layer perceptron cannot solve input vectors that are not linearly separable [34]. The most prominent example of this limitation is the exclusive-or (XOR) problem. Here, given two input booleans, the perceptron should provide a positive response if, and only if, only one of the inputs is positive. No linear line can be drawn to separate both cases (see Figure 2.1).

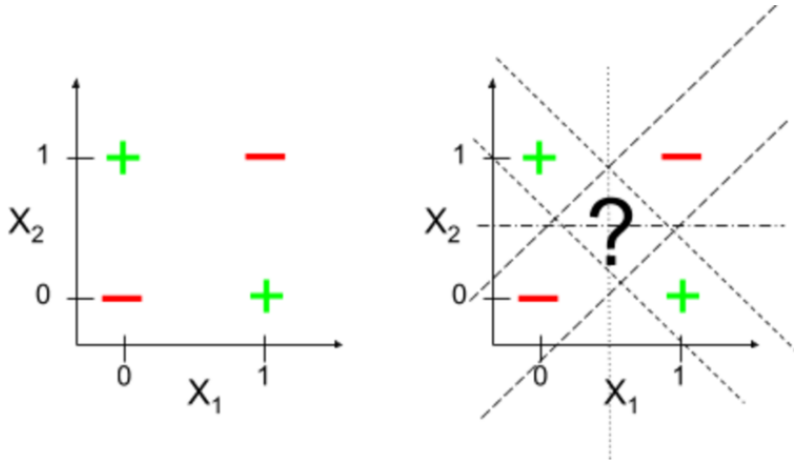


Figure 2.1: The XOR problem visualised. No linearly separable line can be drawn to separate both responses.

2.1.2 Multi-layer Perceptrons (MLPs)

The problem of single-layer perceptrons inability to solve not linearly separable inputs was solved by adding one (or multiple) layers between the input and output layer. The addition of a single extra layer (a *hidden* layer) gives this new multi-layer perceptron the ability to solve the XOR problem and other not linearly separable inputs. MLPs do not require assumptions as to the distribution of the data or the equality of the covariance matrices of the groups to be classified. The number of nodes in each layer is called its *width*, the number of layers in an NN is called its *depth*. Often only the number of hidden layers are counted, since all networks have an input and output layer. Consequently, an NN with a large depth is called a deep NN. Similarly to Equation 2.1, the lth layer of a MLP can be written as the following function:

$$f^{(l)}(\mathbf{x}; \mathbf{W}^{(1)}, \mathbf{b}^{(l)}) = a(\mathbf{W}^{(l)\top}\mathbf{x} + \mathbf{b}^{(l)}), \tag{2.2}$$

where $\mathbf{W}^{(l)}$ is the weight matrix of layer l that performs a linear transformation on the input matrix $\mathbf{x}$ and a represents the pointwise activation function (section 2.5). All weights are arranged in the weight

matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{m \times n}$, m is the number of input dimensions and n is the number of nodes in the layer l. The activation function determines what the output of a node looks like. Equation 2.1 uses a step function as activation function. This activation functions ensures that either the node does not output at all (i.e. 0), or the output is the maximum value it can be (i.e. 1). Many different activation functions can be chosen as we discuss in section 2.5.

The addition of (one or more) hidden layers with non-linear activation functions allows MLPs to approximate non-linear functions. This makes MLPs a universal function approximator, allowing an MLP with one or more hidden layers and a sufficient number of nodes theoretically capable of representing any continuous function [35].

2.2 Supervised Learning

A vast number of machine learning applications use supervised learning. Supervised learning is the process of learning the mapping function from the input $\mathbf{X}$ to the output (e.g. a class or a numerical value) by exposure to labelled examples. Supervised learning can be seen as a teacher *supervising* the learning process. It will provide examples containing a *ground truth* (a label that is known to be correct, for example, given by an expert) as to what the desired output of this example should be. Through a stochastic learning process, the network will eventually be able to approximate a function that can map the input to the correct output [36].

Next to supervised learning, the other two main branches of machine learning are unsupervised learning and reinforcement learning. Unsupervised learning has to the goal to identify structures in unlabelled data, while reinforcement learning studies how artificial agents can learn to adapt their behaviour to their environment by interaction. This thesis, however, will focus on supervised learning.

Supervised learning can be split into two subcategories: *classification* and *regression*.

2.2.1 Classification

During training, a supervised learning classification algorithm (a *classifier*) will be given data points with an assigned category (the ground truth). The classifier then has to assign the correct class to an input value based on the train data. The train dataset can be written as:

$$\mathcal{D} = \{\mathbf{X}, \mathbf{Y}\} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \tag{2.3}$$

where $\mathbf{x}^{(i)}$ denotes the input vector to the classifier, $y^{(i)} \in \{1, ..., C\}$ denotes the corresponding ground truth class from the C different classes. If $C = 2$, we have a binary classifier again. Often the ground truth labels are written as a one-hot vector to allow a classifier to output multiple different classes for a single input (multi-label classification). An one-hot vector is the length of the number of classes and contains a 1 for the representing class it is labelled as, and zeros for all other classes. This changes the description of the dataset as follows:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}, \tag{2.4}$$

where $\mathbf{y}^{(i)}$) is now a one-hot vector.

Multi-label classification could for example be used in the classification of animal species, where we not only want to classify the animal species (e.g. a cat) but simultaneously the animal's gender as a class as well.

2.2.2 Regression

Regression is a predictive statistical process where the network attempts to approximate a function that maps the relation between dependent and independent variables. This is similar to classification, but

instead of outputting one or more classes, the network outputs a continuous number. This number could for instance be a salary, a test score, a predicted stock market price or a life expectancy. The dataset for regression can be described in the same fashion as the dataset for classification with a one-hot vector is described:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \tag{2.5}$$

where $y^{(i)}$) is now a continuous variable.

2.2.3 Training and Validation

All supervised learning algorithms rely on labelled data for training and evaluation of the performance. The validity of these labels is vital because the algorithm will learn to match the input to the label regardless of the label being correctly assigned. The dataset of labelled data is usually divided into three mutually exclusive subsets.

The first, and usually the largest, subset is called the training set. The training set contains the only examples the network can use during training to approximate the function that maps these input examples to the desired output labels. The second subset is called the validation set and contains examples that the network can measure its performance against next to the training set. The difference with the training set is that the network does not have access to these examples like it does for the training set. Therefore the network tries to get a high accuracy score on both the training set and validation set, but can only use the features found in the training set to do so. The validation set can also be used to optimise the hyperparameters of the network and the training procedure, i.e. the parameters that are set before the optimisation starts. The last subset is called the test set and is used to test the performance of the network after training is completed. Since the network has not seen any of the examples in this subset, the network can only perform well on this set when it has learned an accurate approximation of the underlying distribution that explains the data.

Training a feedforward NN through supervised learning can be done through backpropagation [5]. Backpropagation efficiently tries to fits the NN by computing the gradient of the loss function with respect to each weight in the network. It does so through stochastic gradient descent (SGD). We will have a more detailed look at the process of backpropagation in section 2.3 and section 2.4.

2.3 Likelihood and Loss

Training an NN through SGD requires a suitable loss function. As discussed previously, a feedforward NN (MLP) with one or more hidden layers and sufficient many nodes is capable of representing any continuous function. The problem, however, is that, even when it is proven that a suitable configuration of weights for a certain problem exists, there is no mathematical equation that directly leads to these weights.

We can, however, define criteria that evaluate how well a certain configuration solves the current task. A function that captures these criteria is referred to as the *objective* function. Typically, with NNs, we seek to minimise the error. As such, the objective function is often referred to as the *loss* function (or *cost* function) and the value calculated by this function as the *loss*. The loss function calculates the numeric loss for a certain configuration of the model parameters on the test set and often the validation set (when available). If the weights were configured in such a fashion that all examples produce their correct ground-truth label, the loss function would be optimal and the loss would be 0. Different tasks may value their output differently and therefore require a specific loss function. For example, a regression task may consider a range of values around the ground truth as correct or a classification task might value a correct prediction of a certain class more strongly than another class. The task of the loss function is, therefore, to distil all these aspects of the network into a single number in such a way that improvements (a decrease) to

that loss number are a sign of a better network. It is, however, noteworthy that a perfect loss of 0 does not mean that the network has found the perfect configuration for the problem at hand. The loss can only be calculated on a finite subset of the true (but unknown) data-generating distribution of the data to predict. A perfect loss on that finite subset means that a function is found that perfectly describes the examples in the subset, but does not mean that this function can be generalised to the true (often infinite) distribution of examples representing the data. A perfect loss may therefore very well be a sign of over-fitting.

2.3.1 Classification

The output of a classification task is usually a vector of probabilities for each class. Each example presented to the classifier will output predicted probabilities for belonging to each class c as follows:

$$P_{model}(y^{(i)} = c|\mathbf{x}^{(i)};\theta), \tag{2.6}$$

where θ represents the parameters of the network.

Next, the conditional likelihood function describes the probability that the network assigns to a dataset $\mathcal{D}$:

$$\mathcal{L}(\theta,\mathcal{D}) = P(\mathbf{Y}|\mathbf{X};\theta), \tag{2.7}$$

which can be written as:

$$\mathcal{L}(\theta,\mathcal{D}) = \prod_{i=1}^{N} P_{model}(y^{i}|\mathbf{x}^{(i)};\theta). \tag{2.8}$$

This assumes the data to be independently and identically distributed (i.i.d.) [5]. We want to select the parameters θ that have the highest likelihood of explaining the data $\mathcal{D}$. Equation 2.8 shows that for a classification task that this is the case when the network predicts high probabilities to classes that are the ground truth labels $y^{(i)}$, given the input vector $\mathbf{x}^{(i)}$. This product over many probabilities can be inconvenient for a number of reasons. For example, it is prone to numerical underflow [5]. A more convenient equation is the negative log-likelihood (NLL) as follows:

$$\text{NLL} = -\sum_{i=1}^{N} \log P_{model}(y^{(i)}|\mathbf{x}^{(i)};\theta). \tag{2.9}$$

We use the NLL since the range of the positive log-likelihood is $(-\infty, 0]$, whereas the NLL lies in the range $[0,\infty)$ so a perfect network would have a loss of 0. The NLL is also called the *cross-entropy error function* when used for classification problems [37]. We can now write the maximum likelihood estimation (MLE) for the networks parameters as follows:

$$\theta_{\text{ML}} = \arg\max_{\theta} \prod_{i=1}^{N} P_{model}(y^{(i)}|\mathbf{x}^{(i)};\theta) \tag{2.10}$$

$$= \arg\min_{\theta} -\sum_{i=1}^{N} \log P_{model}(y^{(i)}|\mathbf{x}^{(i)};\theta). \tag{2.11}$$

The MLE for θ can derive specific functions that are good estimators for different parameters of the network. Consider a subset of m examples $\mathbb{X} = \{\mathbf{x}^{(1)},...,\mathbf{x}^{(m)}\}$ drawn independently from the true but unknown data-generating distribution $p_{\text{data}}(\text{x})$.

Let $p_{\text{model}}(\text{x};\theta)$ be a parametric family of probability distributions over the same space indexed by θ. This means that $p_{\text{model}}(\text{x};\theta)$ maps any configuration of $\mathbf{x}$ to a real number estimating the true probability $p_{\text{model}}(\text{x})$ [5].

2.3.2 Regression

The most common loss function for regression is the mean squared error (MSE). Using the MSE for a regression problem is in effect the same as using the cross-entropy loss (or the NLL for that matter). Any loss consisting of an NLL is a cross-entropy between the distribution of the training set and the probability distribution defined by the network [5]. For example, the MSE is the cross-entropy between the distribution of the training set and a Gaussian model.

Given a dataset with N examples, the MSE is obtained by taking the average of the squared error between the predicted $\hat{\mathbf{y}}^{(i)}$ and the corresponding ground truth target $\mathbf{y}^{(i)}$:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)})^2. \tag{2.12}$$

The MSE incorporates both the variance and, if not for an unbiased estimator, the bias of the estimator. The error is squared to increase the impact of large deviations compared to the mean absolute error (MAE). Minimising the MAE or the MSE is equivalent to minimising the L1 (also known as the *Manhattan Distance*) norm or the L2 (also known as the *Euclidean*) norm respectively. The MSE is always positive and a number closer to zero represents a better model (in measurable terms defined by the distribution of the (limited) dataset).

2.4 Optimisation

Neural networks are notoriously difficult to optimise. It is not uncommon to invest days to months on hundreds of machines to solve a single instance of a neural network training problem. The loss functions that need to be optimised are non-convex and there are no theoretical guarantees about the performance of the most popular functions available [38].

Because this problem is so important and expensive, a specialised set of optimisation techniques have been developed. In general, these techniques try to find the network parameters θ that significantly reduce a cost function $J(\theta)$. The cost function typically is a performance measure that evaluates on the entire training set as well as a validation set to account for generalisation.

The optimisation of NNs is usually optimising indirectly. Rather than optimising for the best performance P on the training set, we want the best performance on a novel test set. We, therefore, optimise P only indirectly by reducing a cost function $J(\theta)$ in the hope that it will also improve J. For pure optimisation reducing J would be the goal itself.

Typically, the cost function can be written as an average over the training data as follows:

$$J(\theta) = \mathbb{E}_{(\mathbf{x},y)\ \hat{p}_{\text{data}}} L(f(\mathbf{x};\theta), y), \tag{2.13}$$

where L is the per-example loss function, $f(\mathbf{x};\theta)$ the predicted output of $\mathbf{x}$, and $\hat{p}_{\text{data}}$ the training distribution. Note that this cost function is for supervised learning, hence the addition of y.

2.4.1 Parameter Initialisation

Each weight and bias in an NN needs to have a certain value before training can start. If each weight is initialised with the same value, each node connected to the same inputs would contribute to the output in the same way and therefore no optimisation regarding the contribution of individual nodes can be done. Therefore, all parameters of an NN are usually randomly initialised. This way each node can be tuned through backpropagation after each training step (each *epoch*).

Calculating the activation of each layer through matrix multiplication could, however, lead to unexpected problems with randomly initialised weights. A too-large difference between the variance of

activation between layers in an NN could cause the activation gradient for a large enough network to become infinitesimally small. Forcing a standard deviation for each layer of about 1 could still cause this problem [39].

A common initiation method to omit these problems is to initialise a layer's weights to values chosen from a random uniform distribution as proposed by He et al. as follows [40]:

$$w_{lk}\ \mathcal{N}(0, \sqrt{2/n_l}), \tag{2.14}$$

where w_{lk} is the value of the *kth* weight in layer l and n_l is the dimensional of the input to the nodes in layer l. He et al. initialise all biases as zero.

2.4.2 Gradient Descent

Gradient descent is an algorithm to minimise the loss function. Suppose we have a function $y = f(x)$, with y and x real numbers. The derivative, that gives the slope of $f(x)$ at the point x, is denoted as $f'(x)$. This specifies how to scale a small change in the input to obtain the corresponding change in the output: $f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$. The derivative can help us minimise a function because it can tell how to change x to make a small improvement in y. For example, $f(x - \varepsilon sign(f'(x)))$ is less then $f(x)$ for a small enough ε. From this, we can derive that we can reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative. This is called gradient descent [41]. See Figure 2.2 for an example of gradient descent.

When the derivative becomes zero ($f'(x) = 0$), no information can be gathered as in what direction to move. These points are known as *critical points*, or *stationary points*. When at a critical point $f(x)$ is lower than all neighbouring points, and $f(x)$ can no longer be decreased, a *local minimum* is reached. If the value of $f(x)$ at this point is the lowest value $f(x)$ can get, a *global minimum* is reached. Not every local minimum is a global minimum. Large NNs often have many local minima that can be difficult to leave or very flat areas that can be hard to traverse. We therefore often try to find a local minimum with an acceptable low value of f, since finding the actual global minimum might be an impossible task.

NNs often have multiple inputs, so we have to use *partial derivatives*. The partial derivative $\frac{\partial}{\partial x_i} f(\mathbf{x})$ measures how f changes when only the variable x_i increases at point $\mathbf{x}$. The gradient of f is now a vector containing all the partial derivatives, denoted by $\Delta_{\mathbf{x}} f(\mathbf{x})$. The increased dimensions now mean that a critical point is a point where the derivative of every element of the gradient is zero.

We can now find a new, better, point proposed by the gradient descent algorithm as follows:

$$\mathbf{x}' = \mathbf{x} - e\Delta_{\mathbf{x}} f(\mathbf{x}), \tag{2.15}$$

where e is the *learn rate*. The learn rate is a positive scalar determining the size of the step towards the new point. If the learn rate is too big, the NN cannot fine-tune to find a good solution. Set it too small and the NN will get stuck in the very first local minima while training very slow as well.

The learn rate can also vary during training, as with *learn rate annealing*. Here, the learn rate starts high to avoid local minima and to speed up training, while slowly decreasing (cooling off) to fine-tune the parameters as a good solution nears.

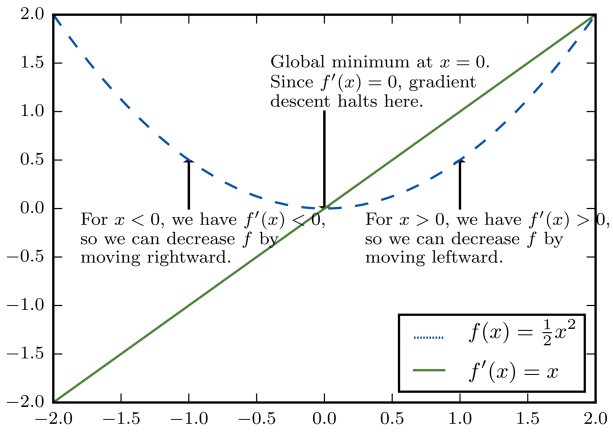


Figure 2.2: An illustration of how the gradient descent algorithm using the derivatives of a function can be used to follow the function downhill to a minimum (from *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio [5])

2.4.3 Stochastic Gradient Descent

Almost all NNs are trained by a variation on gradient descend: *stochastic gradient descent* (SGD). Since the non-linearity of an NN causes most loss function to become non-convex, they are usually trained using an iterative, gradient-based optimiser. This optimiser merely tries to drive the cost function to a low value, instead of convex optimisation algorithms that have a global minimum guarantee (e.g. for support vector machines).

SGD applies to non-convex loss functions, has no convergence guarantee and is, as we saw in section 2.4.1, sensitive to the initialisation of the parameters. SGD uses a mini-batch (typically varying between 1 and hundreds of examples per batch) instead of the complete training set as with gradient descent. The larger the mini-batch, the more computations and memory is required, but also the more precise the update is. Larger mini-batches may, however, due to their increased precision, end in steeper local minima [42]. Flatter local minima are often more desirable because they are less specific to the training data and therefore generalise better.

The addition of *momentum* may accelerate the SGD process and can help to avoid local minima. A momentum term is added to the gradient that increases the size of the update in the direction of the last few updates.

2.4.4 ADAM

Adam (derived from ”adaptive moments”) is a popular adaptive learning rate optimisation algorithm [43]. Adam can be seen as a variant on the combination of Root Mean Square Propagation (RMSprop) algorithm and SGD with momentum. RMSprop improves the SGD procedure by keeping track of an exponentially decaying second moment (the pointwise square) of the gradient and dividing this gradient by the square root of this moving average [44]. Adam also corrects for the bias in the second moment during early episodes of training [5]. Adam is generally regarded as being robust to the choice of hyperparameters. The algorithm for ADAM can be seen in Algorithm 1.

2.5 Activation Functions

The activation function computes the output of the unit in each hidden layer of an NN. An example of an activation function can be seen in Equation 2.2. Many different variations exist, and NN architectures

Algorithm 1: The Adam algorithm

Require: Step size ε (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho 1$ and $\rho 2$ in [0, 1). (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilisation (Suggested default: 10^{-8})
Require: Initial parameters θ
Initialise 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$
Initialise time step $t = 0$
while *While condition* **do**
  Sample a mini-batch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.
  Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  $t \leftarrow t + 1$
  Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
  Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
  Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
  Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
  Compute update: $\Delta\theta = -\varepsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ (operations applied element-wise
  Apply update:$\theta \leftarrow \theta + \Delta\theta$
end

often use different activation functions for specific layers in the network. A default choice of layer is often the *rectified linear unit* (ReLU) with the activation function:

$$g(x) = max\{0, x\}, \tag{2.16}$$

where x represents the input to the unit. A visual representation of the ReLU function can be seen on the right side of Figure 2.3. This activation is very similar to a linear function with the difference that it outputs a zero for all negative values. This causes a large derivative when the unit is active, with a large but consistent gradient. Compared to activation functions that introduce second-order effect the second derivative of the ReLU is almost zero everywhere and 1 everywhere the unit is active, causing the gradient direction to be very useful for learning [5].

The ReLU activation function has largely replaced the *sigmoid* activation function:

$$\sigma(x) = \frac{1}{1 + \varepsilon^{-x}}, \tag{2.17}$$

A visual representation of the sigmoid function can be seen on the left side of Figure 2.3. The activation from the sigmoid function ranges in an s shaped curve between 0 (for large negative values) and 1 (for large positive values). The derivative of the sigmoid function is bell-shaped with its maximum for $x = 0.5$. A drawback of sigmoid is that the function becomes saturated for very small or very large input values, with a derivative close to zero. As we saw in Section 2.4.1, very small derivatives can cause the backpropagation algorithm to break for NNs with many layers. When the derivatives are too small, no information about the contribution of previous layers can be extracted anymore. This causes the algorithm to not be able to determine in which direction to change the parameters. This phenomenon is known as the vanishing gradient problem.

The ReLU activation function holds three advantages over the sigmoid activation function.

Firstly, the computation is faster since ReLU is in effect a linear operation.

Secondly, it does not saturate, so it does not cause the vanishing gradient problem.

Finally, ReLUs can introduce sparsity. Sparsity arises with the activation is smaller or equal to zero. More optimisation operation on sparse matrices exist which improve the performance and robustness of the NN due to for example the disentanglement of information [45].

The output layer of classifiers usually uses the *softmax* activation function [5]. The softmax function computes the probabilities associated with a multi-class distribution. The softmax function is defined as:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{(x_i)}}{\sum_{j=1}^{n} e^{(x_j)}}. \tag{2.18}$$

Here, the standard exponential function gets applied to each element x_i of the input vector $\mathbf{x}$ and then normalised by dividing by the sum of all these exponentials. The normalisation ensures that the sum of all components in the output vector is 1.

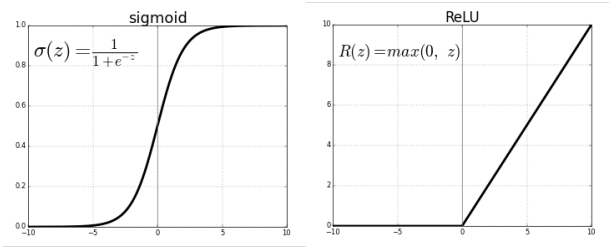


Figure 2.3: ReLU v/s Logistic Sigmoid

2.6 Regularisation

A core problem of training NNs is how to train an NN that not only performs well on the training data, but also on new inputs. An NN that performs well on new inputs is said to generalise well but is over-fitted when it only performs well on its training data. The parameters in an over-fitted model are too specifically tuned for the details of the examples in the training set, instead of tuned to capture the general underlying pattern that is shared by all examples of a certain class. Strategies that are explicitly designed to reduce the test error, possibly at the expense of the training error, are known as regularisation techniques. Many regularisation techniques exist. Some add extra terms to the objective function that can be seen as a soft constraint on the parameter values. These terms add constraints and penalties that can improve generalisation when chosen carefully. Some use specific kinds of prior knowledge, while others may express a general preference for a simpler model. Others again, known as ensemble methods, combine multiple hypotheses that explain the data [5].

For deep learning, most regularisation techniques are based on regularisation estimators. Regularisation of an estimator works by trading increased bias for reduced variance, preferably trading a small increase in bias for a large decrease in variance. Three situations can occur when training:

1. Training excludes the true data-generating process. This causes underfitting and introduces bias.
2. Training matches the true data-generating process.
3. Training includes the true-generating-process but also many other possible generating processes. Variance rather than bias dominates the estimation error.

The goal of regularisation is to get from situation three to situation two.

In practice, however, an NN might not include the target function or the true data-generating process (or even a close approximation). The true data-generating process is rarely available for checking if the model includes this process or not. For a complex task that most (deep) NNs are used for, like for example

image classification, the true data-generating process will almost certainly be outside the parameters of the network. Finding the true data-generating process for these tasks might include simulating the complete universe to capture all variances.

Acknowledging this means that the best model that we can find usually is a large model that has been regularised properly. In section 3.2.2 till section 3.2.4 we will go over some of the most popular regularisation techniques currently used.

2.7 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a variation on NNs that are specialised for processing data with a grid-like topology. This includes time-series data (a 1-D grid taking samples at regular time intervals) or image data (2-D grid of pixels). CNNs use the linear *convolution* operation instead of general matrix multiplication for at least one of their layers.

For a 2-D grid of pixels, like the image data we use in this thesis, we can write the convolutional operation as follows:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{2.19}$$

where i and j denote the indexing of the resulting matrix, I is a two-dimensional input image and K the two-dimensional *kernel* of m by n. The resulting matrix is often called the *feature map*. The multi-dimensional input array I and the multi-dimensional kernel array K of parameters that are adapted by the learning algorithm, are referred to as tensors. The convolutional operation normally is an infinite summation, but since each element of the input and kernel must be explicitly stored separately, we assume them to be zero everywhere except for the finite points for which we store the values. This results in Equation 2.19. A visual example of a convolution operation can be seen in Figure 2.4.

CNNs improve NNs in a few aspects [5]:

- Convolutional layers have *sparse interactions* with their input.
- CNNs *share parameters* for more than one function in the network.
- Convolutional layers are *equivariance*.
- Convolutions allow the input to be of variable size.

First, when the kernel of a convolutional layer is smaller than the input, not every input node interacts with every output node (*spares interactions*). This allows convolutional layers with a relatively small kernel size (tens to hundreds of pixels big) to detect small, meaningful features such as high contrast areas throughout an input image consisting of thousands or millions of pixels. This allows for fewer parameters in the network which reduces its memory requirement and improves its statistical efficiency and performance.

Secondly, every member of the kernel is used at every position of the input (except maybe the edges, unless a padding technique is used) (*parameter sharing*). This allows the network to learn only one set of parameters for a feature that can occur at any location in the input, instead of learning it separately for every location in the input. Since the kernel size k is usually several orders smaller than the input size, the convolutional operation is many times more efficient than dense matrix multiplication in terms of memory requirements and statistical efficiency.

Thirdly, if the input to a convolutional layer changes, the output changes in the same way (the layer is *equivariance* to translation). For images, convolution creates a 2-D map of where certain features appear in the input. If the object in the images was to be shifted, the representation in the output will be shifted an equal amount.

Lastly, convolutions allow for the processing of inputs with different sizes.

Above we saw that the centre of the kernel cannot reach to the edge pixels since the edges of the kernel would then be outside of the input. A solution can be through *padding*. Padding the input puts extra data (usually zeros, but for example, a repetition of the values at the border is also possible) around the input, which allows the kernel to traverse the full dimensions of the input.

The kernel can shift across the input one or more data points at a time. This is called the *stride*. Increasing the stride results in a smaller dimensional output.

2.7.1 Pooling

A typical CNN consists of three stages [5]. First, the layer performs several convolutions in parallel to produce a set of linear activation. Second, each of these activation is passed through a non-linear activation function (see Section 2.5). Thirdly, a *pooling* function is used to modify the output further.

Pooling is used to reduce the dimensionality of the output of a convolutional layer since features become more abstract in later layers of the network and precise location plays a less important role. *Max pooling*, one of the most common pooling techniques, reduces the dimensionality of a layer by only retaining the maximum output within a certain rectangular neighbourhood [46]. Other pooling techniques, such as the average value of a certain neighbourhood, are used as well.

Pooling helps to make the representations approximately invariant to small translations of the input, at the cost of losing information about the exact location. This is useful when we care more about the presence of a certain feature than we care about its exact location. Pooling can, however, introduce unwanted effects when the separate features are present in the input, but the geometric relations between them are not [47].

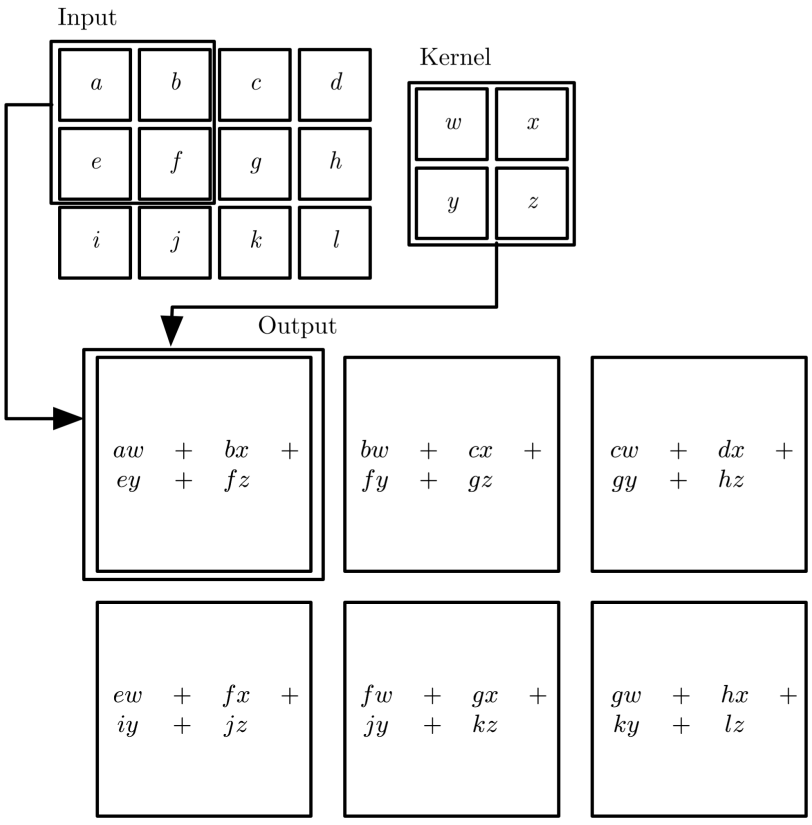


Figure 2.4: Example of a 2-D convolution, where the output is restricted to positions where the kernel lies entirely within the image. The result is a 3 by 2 matrix of summations. (From *Deep Learning* by Aaron Courville, Ian Goodfellow, and Yoshua Bengio [5].)

Chapter 3

Uncertainty Estimation for Neural Networks

3.1 Types of Uncertainty

Bayesian modeling can model two main types of uncertainty, **aleatoric** and **epistemic** uncertainty.

Aleatoric uncertainty arises through noise in the observations (e.g. sensor noise). This uncertainty is inherent in the observations and therefore cannot be reduced with more observations. Aleatoric uncertainty can be categorised in to *homoscedastic* and *heteroscedastic* uncertainty. Homoscedastic uncertainty stays constant for all inputs (all observations exhibit the same finite variance, i.e. the task-dependent uncertainty), heteroscedastic uncertainty can differ for each input.

Heteroscedastic uncertainty is usually the most important for computer vision tasks since these tasks often rely on sharp distinct features to make accurate classifications. When these features are harder to detect (e.g. the wrong focus of the camera causes object corners to blend into each other to form featureless parts in the image), the output of a model will be highly uncertain.

Epistemic uncertainty (*model uncertainty*) arises through uncertainty in the model parameters and the model's ignorance about which model generates the training data. This can be reduced with additional observations since more data gives a better understanding of the true model that generates the data.

In classification tasks, aleatoric uncertainty is more important to model since it cannot be accounted for with more data [6]. However, medical data might not always provide enough examples, which keeps modelling epistemic uncertainty important as well.

Combining aleatoric and epistemic uncertainty results in predictive uncertainty, the model's confidence in its prediction, taking into account the noise it can and cannot account for with more data. The predictive uncertainty is both effected by aleatoric uncertainty (increasing predictive uncertainty in case of a large measurement error) or epistemic uncertainty (increasing predictive uncertainty for inputs that lie far away from the training data). Predictive uncertainty is usually obtained by sampling multiple functions from the model and corrupting them with noise. We can then calculate the variance of these functions on a fixed set of inputs which yields the predictive uncertainty.

3.2 Methods for Approximating Bayesian Models

A Bayesian Neural Network (BNN) puts a prior distribution over its weights [48–50]. This can be for example a Gaussian prior distribution: $W \sim \mathcal{N}(\mu, \sigma^2)$. BNNs replace the weight parameters point-estimate networks use with distributions over these parameters. Instead of optimising the weight parameters directly, BNNs average over all possible weights in the distribution (*marginalisation*). Given a training dataset $\mathbf{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1:N}$ with sample-label pairs $(\mathbf{x}_i, \mathbf{y}_i)$, Bayesian inference is used to calculate the posterior over the weights $p(\omega|\mathbf{x}, \mathbf{y})$, with parameters ω. The posterior holds the set of plausible model parameters for the data. To evaluate the posterior, $p(\omega|\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x}, \omega)p(\omega)/p*(\mathbf{y}|\mathbf{x})$ needs to be calculated. However, the marginal probability $p(\mathbf{y}|\mathbf{x})$ cannot be evaluated analytically, only approximated. This

approximation can be calculated through e.g. variational inference [51].

For classification, the posterior distribution of the model parameters $p(\omega|\mathbf{D})$ can be approximated as follows:

$$p(\mathbf{y}|\mathbf{x},\mathbf{D}) = \int f_\omega(\mathbf{x},\mathbf{y})p(\omega|\mathbf{D})d\omega, \tag{3.1}$$

where $f_\omega(\mathbf{x},\mathbf{y})$ is an inference function.

The predicted label $\hat{\mathbf{y}}$ can be obtained by sampling $p(\mathbf{y}|\mathbf{x},\mathbf{D})$ or by taking its maxima.

3.2.1 The Gaussian Process

The Gaussian process is a stochastic process such that every finite collection of data has a multivariate normal distribution. For a given dataset, the Gaussian process assigns a probability to each of the functions that can explain the data. The goal of the Gaussian process is to, stochastically, learn the multivariate normal distributions underlying the train data Y. From this multivariate normal distribution, we want to predict the function values (or class in a classification problem) at specific points in the test set X. To get to this prediction, X also needs to be modelled as a multivariate normal distribution. The joint probability distribution $P_{X,Y}$ then spans the space of possible function values for the function (or class) that we want to predict.

Both X and Y are Gaussian distributions and are therefore *closed under conditioning* and *marginalisation*. This means that the resulting distributions after performing operations on X or Y are also Gaussian distributions. Through *marginalisation* we can extract the marginalised probability distributions from the multivariate probability distribution $P_{X,Y}$ as follows:

$$X \sim \mathcal{N}(\mu_X, \Sigma_{XX}) \tag{3.2}$$
$$Y \sim \mathcal{N}(\mu_Y, \Sigma_{YY}), \tag{3.3}$$

where $\mathcal{N}$ represents the Gaussian distribution, μ the mean vector and Σ the covariance matrix. From this formula we can determine that each partition of X and Y only depends on its corresponding entries in μ and Σ. If we want to know the probability density of a point x, we need to consider all possible outcomes of Y that can jointly lead to this result. This can be shown by the following equation:

$$p_X(x) = \int_y p_{X,Y}(x,y)dy = \int_y p_{X|Y}(x \mid y)p_Y(y)dy. \tag{3.4}$$

Conditioning is used to determine the probability of one variable depending on another variable. This operation, similarly to marginalisation, also yields a modified Gaussian distribution. Conditioning is defined as:

$$X \mid Y \sim \mathcal{N}(\mu_X + \Sigma_{XY}\Sigma_{YY}^{-1}(Y-\mu_Y), \Sigma_{XX} - \Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{YX}) \tag{3.5}$$
$$Y \mid X \sim \mathcal{N}(\mu_Y + \Sigma_{YX}\Sigma_{XX}^{-1}(X-\mu_X), \Sigma_{YY} - \Sigma_{YX}\Sigma_{XX}^{-1}\Sigma_{XY}), \tag{3.6}$$

where the new mean only depends on the conditioned variable, while the covariance matrix is independent from said variable.

To perform regression on the training data, we need to use *Bayesian inference* [52]. Bayesian inference uses the training data to update the current hypothesis.

To set up the Gaussian distribution we have to define the mean μ and covariance matrix Σ. We can define the covariance matrix Σ through the *covariance function* k, also known as the *kernel* of the Gaussian process. Kernels are widely used in machine learning and many different kernels exist for the Gaussian process. Often μ is assumed to be 0 to simplify the necessary equations for conditioning.

When the multivariate Gaussian distributions have been found they can be used to estimate function values. Each test point is treated as a random variable and the corresponding multivariate Gaussian distribution has the same number of dimensions as the number of random variables. To make a prediction we draw samples from this distribution. The i-th component of the resulting vector corresponds to the i-th test point.

A single-layer fully-connected neural network with an independent and identically distributed prior over its parameters is equivalent to a Gaussian process, in the limit of infinite network width [19]. By evaluating the corresponding Gaussian process, exact Bayesian inference can be performed on this infinite width neural network. The standard Gaussian Process, however, has cubic time complexity $O(n^3)$ [20]. This limits the scalability of the Gaussian Process and makes it unusable for deep learning. However, model uncertainty can also be obtained through Bayesian Neural Networks (BNNs) [21]. BNNs place a distribution over a finite number of weights, therefore vastly reducing the computation required. Various techniques using the basic ideas of Bayesian neural networks have been introduced over the years [22], with various degrees of success. Often these models are difficult to work with and introduce many more parameters to be trained, making it again unsuitable for large-scale deep learning.

We will use two methods that do not suffer from these drawbacks: Monte Carlo Dropout [11] (MCDO) and Monte Carlo Batch Normalisation [23] (MCBN). These two methods make use of stochastic regularisation techniques.

3.2.2 Stochastic Regularisation Techniques

Stochastic regularisation techniques (SRTs) are used to regularise a network by injecting stochastic noise into the model. This often increases generalisation on novel data by preventing over-fitting. Almost every network trained with an SRT can output, given some input $\mathbf{X}^*$, a predictive mean $\mathbb{E}[\mathbf{y}^*]$ and a predictive variance $Var[\mathbf{y}^*]$. The predictive mean is the expected model output given the input and the predictive variance reflects how confident the model is in this prediction.

To gain these results we obtain the output of the network with input $\mathbf{X}^*$ while treating the SRT as we would during training time (i.e. obtain a random output through a stochastic forward pass) [11]. This process can be repeated T times to sample independent and identically distributed outputs. These are empirical samples from the *approximate posterior*. We can get an empirical estimator for the predictive mean and variance of the approximate posterior as follows:

$$\mathbb{E}[\mathbf{y}^*] \approx \frac{1}{T}\sum_{t=1}^{T} \hat{\mathbf{y}}_t^*(\mathbf{x}^*) \tag{3.7}$$

$$Var[\mathbf{y}^*] \approx \tau^{-1}\mathbf{I}_D + \frac{1}{T}\sum_{t=1}^{T} \hat{\mathbf{y}}_T^*(\mathbf{x}^*)^T \hat{\mathbf{y}}_t^*(\mathbf{x}^*) - \mathbb{E}[\mathbf{y}^*]^T\mathbb{E}[\mathbf{y}^*], \tag{3.8}$$

where τ^{-1} is observed noise (the inverse variance, also often called precision), I_D the expected log likelihood's derivative with respects to the dataset D of observed d-dimensional vectors.

3.2.3 Monte Carlo Dropout

Gal (2016) showed that a typical optimisation of NNs with Dropout layers is equivalent to Bayesian learning via variational inference with a specific variational distribution [11]. Dropout [53, 54] is one of the most popular SRTs for deep learning models.

Dropout samples i binary vectors for each layer excluding the output layer, with the same dimension as the respective layer. The elements in these vectors z_i take value 1 with probability $0 \leq 1 - p_i \leq 1$ for the amount of layers i. A proportion p_i of all elements in the input x to a certain layer are set to zero as follows: $\hat{x} = x \odot z_i$ [1]. A proportion $p_i + 1$ of the output h of the next layer is again set to zero: $\hat{h} = h \leq z_i + 1$. This

[1] $\odot$ is the element-wise product

process is repeated for every layer until the output can be computed: $\hat{y} = \hat{h}W_i$.

For each different input, forward pass and backward pass through the network's new binary vectors z_i are calculated. When the model is being evaluated all connections are used and no sampling through Dropout is applied.

It has been found that optimising any NN through Dropout is equivalent to a form approximate inference in a probabilistic interpretation of the model. In other words, the optimal weights found through Dropout are equal to the optimal variational parameters in a Bayesian Neural Network, making it a Bayesian Neural Network by process. The uncertainty can be obtained by performing several stochastic forward passes through the model and sampling the mean and variance. The inference in this technique is called *Monte Carlo Dropout* (MCDO) and can be applied to any pre-trained network with Dropout layers.

Dropout has, however, some shortcomings. First, the test time is scaled by the number of forward passes through the network (training time is nearly identical to similar models in the field) [11]. Secondly, variational inference is known to underestimate predictive variance through the objective which penalises placing mass where there is no mass [55]. Several solutions exist with varying practicality, but uncertainty underestimation also does not seem a real concern in practice [11].

Thirdly, the Dropout probability together with the weight configuration of the network determines the magnitude of the epistemic uncertainty. A fixed probability with higher magnitude weights will cause higher output variance (i.e. higher epistemic uncertainty). Therefore, the model would like to decrease its weights (to zero for zero uncertainty, but then losing its ability to explain data well) and find a balance between the desired output variance and weight magnitude. Allowing for a variable Dropout probability lets the model decrease its epistemic uncertainty by choosing smaller Dropout probabilities as well as by decreasing its weights. The Dropout probability can be determined by grid-searching it to maximise validation log-likelihood [56]. Grid-search is computationally expensive and slow for large models, so a gradient method would be more suitable. However, using a gradient method forces us to define an optimisation objective to optimise the Dropout probability as well. Choosing this is not trivial since our aim is not to maximise the model performance, but to obtain a good epistemic uncertainty. Concrete Dropout [57] offers a suitable objective.

Lastly, the combination of Dropout and Batch Normalisation [58] (a method which speeds up learning by normalising layers inputs and that has a regularisation effect) often leads to worse performance. Performance only seems to increase when the combination is used for very wide networks (e.g. Wide ResNet) [59].

Concrete Dropout

When allowing the Dropout probability to change, a suitable objective needs to be found to allow gradient optimisation. A suitable objective follows Dropout's variational interpretation [11]. Here, Dropout's objective is to approximate the distribution $q_\theta(\omega)$ to the posterior in a BNN with a set of random weight matrices $\omega = \{\mathbf{W}_l\}_{l=1}^{L}$ with L layers and θ the set of variational parameters. The Monte Carlo (MC) optimisation object for Dropout can be written as:

$$\hat{\mathcal{L}}_{MC}(\theta) = -\frac{1}{M}\sum_{i\in S} \log p(\mathbf{y}_i|\mathbf{f}^{\omega(\mathbf{x}_i)}) + \frac{1}{N}\mathbf{KL}(q_t\theta(\omega)\|p(\omega)), \tag{3.9}$$

where θ are the parameters to optimise, N the number of data points, S a random set of M data points, $\mathbf{f}^{\theta(\mathbf{x}_i)}$ the NN's output on input $\mathbf{x}_i$ when evaluated with weight matrix realisation θ, and $p(\mathbf{y}_i|\mathbf{f}^{\theta(\mathbf{x}_i)})$ the model's likelihood, e.g. a Gaussian with mean $\mathbf{f}^{\theta(\mathbf{x}_i)}$. The KL term $\mathbf{KL}(q_t\theta(\omega)\|p(\omega))$ is used as regularisation to prevent the approximated posterior $q_\theta(\theta)$ to deviate too far from the prior distribution $p(\theta)$.

This equation needs to be evaluated through the derivative with regards to the parameter p. This can be done by for example a score function estimation or a pathwise derivative estimator. The score

function estimator, however, has an extremely high variance in practice, making optimisation difficult and not manageable for Dropout [60]. The pathwise derivative estimator has much lower variance and was therefore successfully used for Gaussian Dropout [60]. However, unlike Gaussian Dropout, we want to optimise the parameter of a Bernoulli distribution which cannot be done with the pathwise derivative estimator.

To solve this, Gal, Hron and Kendall (2017), introduced Concrete Dropout, where they replaced Dropout's discrete Bernoulli distribution with the Concrete distribution relaxation [57]. This allowed them to re-parametrise the distribution and to use the low variance pathwise derivative estimator.

Their results showed a slight improvement over regular MCDO, but because of the added complexity, we decided to not implement this improvement over the standard implementation for this comparison.

3.2.4 Monte Carlo Batch Normalisation

Batch Normalisation (BN) [58] is a unit-wise method to normalise the input distribution to each layer in an NN. It does this for each training mini-batch B. BN can be used to mitigate the problem of internal covariate shift, where the initialisation of the parameters and changes in the distribution of the inputs of each layer affects the learning rate of the network. However, more recent findings suggest this might not be the case and BN rather smooths the objective function, which improves performance [61].

Ideally, BN would normalise over the entire training set $\mathbf{D} = \{(x_i, y_i)\}_{i=1:N}$ (where each (x_i, y_i) is a sample-label pair), but to do this with stochastic optimisation methods is impractical. Therefore, for each B of size m, the mean and variance of B is denoted as:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.10}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2. \tag{3.11}$$

For fully connected layers (FC), BN converts the input x^k to each unit k ($k \in [1,d]$, where d is the dimensions of the input) in the layer as follows:

$$\hat{\mathbf{x}}_i^k = \frac{\mathbf{x}_i^k - \mu_B^k}{\sqrt{\sigma_B^{k^2}}}, \tag{3.12}$$

where $k \in [1,d]$, $i \in [1,m]$, μ_B^k the units mean and $\sigma_B^{k^2}$ the units variance.

A deep NN can be trained using mini-batch optimisation as above with the following regularised risk (RR) minimisation:

$$\mathcal{L}_{RR}(\omega) := \frac{1}{M}\sum_{i=1}^{M} l(\hat{\mathbf{y}}_i, \mathbf{y}_i) + \Omega(\omega), \tag{3.13}$$

where the first term $\frac{1}{M}\sum_{i=1}^{M} l(\hat{\mathbf{y}}_i, \mathbf{y}_i)$ represents the empirical loss on the training data and the second term $\Omega(\omega)$ the regularisation penalty on the model parameters ω (this can be seen as a prior).

With a cross-entropy (for classification) as loss l, this is equivalent to minimising the negative log-likelihood:

$$\mathcal{L}_{RR}(\omega) := \frac{1}{M}\sum_{i=1}^{M} ln f_\omega(\mathbf{x}_i, \mathbf{y}_i) + \Omega(\omega). \tag{3.14}$$

The model parameters for each unit in a layer include $\{\mathbf{W}^{1:L}, \gamma^{1:L}, \beta^{1:L}, \mu_{\mathbf{B}}^{1:L}, \sigma_{\mathbf{B}}^{1:L}\}$, where $\gamma^{1:L}$ and $\beta^{1:L}$ are the parameters that are adjusted by stochastic gradient descent (SGD) instead of the weights in each

layer. This prevents SGD to undo the normalisation process of BN. Each output of a layer gets multiplied (*scale*) by γ after which β gets added (*shift*). This *scale* and *shift* makes sure that the transformation inserted in the network can represent the identity transformation. These parameters are learned along with the original model parameters and restore the representation power of the model [58].

To get the learnable parameters $\theta = \{\mathbf{W}^{1:L}, \gamma^{1:L}, \beta^{1:L}\}$, the stochastic parameters $\omega = \{, \mu_{\mathbf{B}}^{1:L}, \sigma_{\mathbf{B}}^{1:L}\}$ need to be decoupled. This results in the following objective at each step of the mini-batch optimisation:

$$\mathcal{L}_{RR}(\theta) := \frac{1}{M} \sum_{i=1}^{M} lnf_{\{\theta, \hat{\omega}_i\}}(\mathbf{x}_i, \mathbf{y}_i) + \Omega(\theta), \tag{3.15}$$

where $\hat{\omega}$ is the mean and variance for samples i's mini-batch at a certain training step as calculated above.

After batch normalising the network, $q_\theta(\omega)$ corresponds to an approximation of the true posterior. From this, the uncertainty can be calculated using the inherent stochasticity of BN [23].

In 2018 Teye et al. showed that training a deep NN with BN is equivalent to approximating inference in a BNN [23]. This finding allows us to make estimates of the model's uncertainty using conventional architectures without modifications to the network or training procedure needed. The mini-batch statistics used for training each iteration depend on randomly selected batch members, which can be exploited to approximate Bayesian inference. The uncertainty of a model trained by BN can be found through a process called *Monte Carlo Batch Normalisation* (MCBN).

The predictive mean for classification can be computed from the approximated predictive distribution $p^*(\mathbf{y}|\mathbf{x}, \mathbf{D})$ as follows:

$$\begin{aligned} \mu_{p^*}[y] &= \int \mathbf{y} p^*(\mathbf{y}|\mathbf{x}, \mathbf{D}) d\mathbf{y} \\ &\approx \frac{1}{T} \sum_{i=1}^{T} f_{\hat{\omega}_i}(\mathbf{x}), \end{aligned} \tag{3.16}$$

where the MC Integral with T samples of ω is taken and $\hat{\omega}_i$ represents a sampled set of the network's stochastic parameters $\omega = \{\mu_{\mathbf{B}}^{1:L}, \sigma_{\mathbf{B}}^{1:L}\}$. As during training, a batch $\mathbf{B}$ is sampled from the training set and the parameters in the BN units are updated.

The Predictive Log Likelihood (PLL) for a test point (y_i, x_i) is taken to estimate the model's uncertainty as follows:

$$\begin{aligned} \text{PPL}(f_\omega(x), (\mathbf{y}_i, \mathbf{x}_i)) &= \log p(\mathbf{y}_i | f_\omega(\mathbf{x}_i)) \\ &\approx \log \frac{1}{T} \sum_{j=1}^{T} p(\mathbf{y}_i | f_{\hat{\omega}_j}(\mathbf{x}_i)), \end{aligned} \tag{3.17}$$

where $\hat{\omega}_j$ represents a sampled set of stochastic parameters from the approximate posterior distribution $q_\theta(\theta)$.

To obtain the uncertainty measures, a network has to be trained similarly to BN, but instead of replacing the network's stochastic parameters $\omega = \{\mu_{\mathbf{B}}^{1:L}, \sigma_{\mathbf{B}}^{1:L}\}$ with population values D from inference, the parameters are updated stochastically every forward pass.

A common approach in Bayesian modeling is using variational approximation (VA) to learn a parameterized approximating distribution $q_\theta(\omega)$ that minimises the Kullback-Leibler (KL) divergence of the true posterior with regards to the approximation. From a VA perspective, training the NN via MCBN amounts to minimising $\text{KL}(q_\theta(\omega)) \| p(\omega|\mathbf{D}))$ with regard to θ. By sampling $\hat{\omega}$ from the training set, and by keeping $\mathbf{B}$ consistent with the mini-batch size used during training, $q_\theta(\omega)$ during inference remains identical to the approximated posterior optimised during training [23].

In summary, BN normalises the input to each layer in an NN by subtracting the mean and dividing by the standard deviation of a mini-batch. The adjustment of these inputs causes the weights to be no longer optimal, so stochastic gradient descent (SGD) will undo the normalisation. To avoid this, BN adds two trainable parameters to each layer, a standard deviation parameter *gamma* and a mean parameter *beta*. The output of a layer is first multiplied by *gamma* after which *beta* is added. BN lets SGD only adjust

these two parameters at each layer and not all the weights of the layer. BN trains each iteration with randomly selected examples in a mini-batch, which can be exploited to approximate Bayesian inference. MCBN can compute the mean classification as well as the uncertainty of a sample by calculating the mean and variance of a mini-batch for multiple inferences. The mini-batch from the training data provides the network's stochastic parameters just as during a training step so MCBN, in essence, checks if the sample falls in the same distribution as the mini-batch of the training data.

3.2.5 Ensemble

Ensembles of NNs, *deep ensembles*, are proven to boost predictive performance in many classification tasks. Ensembles work according to the idea that the average vote of many, slightly different, NNs provides a better prediction than that of a single NN. Generally speaking, two classes of ensembles exist: *randomisation*-based approaches such as random forests [62] and *boosting*-based approaches. Randomisation-based approaches allow the members of the ensemble to be trained in parallel with any interactions, while boosting-based approaches fit the ensemble members sequentially. In 2017, Lakshminarayanan et al. introduced a method to use deep, randomisation-based, ensembles to compute uncertainty for classifications [63]. Here, they treat the many members of an ensemble together as a uniformly-weighted mixture model and use the mean and variance of the resulting predictions as the final prediction and uncertainty respectively.

Ideally, we want all of the different members of the ensemble together to be able to explain the complete dataset. The performance of a single NN can be restrained by several different causes. A certain member of the ensemble might over-fit on the train data and not generalise well, another member might only be able to explain 85% of the data and not the remainding 15%. One member might even completely ignore a class with fewer examples compared to the other classes in an unbalanced dataset.

An ensemble consisting of many members, which by design vary from each other, can solve these problems at least partially. For example, each member can have its own unique architecture, can start with different pre-trained weights or can be trained on a unique part of the train data. Of course, the previously mentioned examples can be combined as well to create a vast ensemble of different networks. However, even if all potential test examples can be classified correctly by at least one of the members, the ensemble will not know which of the many predictions is the right one. Often the majority vote is taken, which means that hard-to-predict examples that are only predicted right by the minority are still not classified correctly.

So instead of only looking at the majority vote (the *mean*), all predictions can be used to determine the *variance* of the predictions. For the "easy" to learn examples, which would be right near the mean of the representing class distributions, the standard deviation of all ensemble members will not be high and therefore the uncertainty will be low. However, for the harder-to-predict examples that are further near the edges of the representing class distributions (the examples that might only be predicted correctly by a minority of members), the standard deviation will be higher and therefore the uncertainty should be higher.

A perfect BNN would have a distribution over its weights that encapsulated every possible explanation of the data. While a standard ensemble does not approximate a distribution over the weights in a single network as MCDO and MCBN do, it does provide many more possible explanations for the same data which can ultimately be used to gain an uncertainty estimate. Taking it a step further, the methods from MCDO, MCBN, or both can also be used on the ensemble members, making it an even closer approximation to a true BNN.

The drawbacks of ensembles, not to mention in combination with MCDO or MCBN, however, are not to be understated. Single deep NN can take hours, if not days to train on large datasets. An ensemble in which each member is trained for maximum variance could take days, if not weeks to successfully train. This makes making small alterations to the networks or the addition of new data or a new class a slow and cumbersome task.

After training the deep networks, millions of parameters need to be stored at increasingly large capacity. Many applications require the fully trained network to be stored offline for reliable and quick access. These networks often run on cheaper, less powerful hardware since the networks do not have to be retrained, but only have to be used for new predictions. The use of ensembles might require these devices to be more powerful and to have more storage. Updating these networks with newly trained weights (for example after the addition of new classes) would take much more time as well and might not be feasible in remote locations where the updates are performed over-the-air or via unstable, slow internet connections.

While a single prediction of a trained network often only takes a fraction of a second, the predictions of a complete ensemble can add up to prediction times that are not feasible in many applications (e.g. autonomous vehicles where split-second decisions can save lives). The addition of MCDO or MCBN on each ensemble member would increase the processing time even more. While this can be partially solved by deploying more powerful hardware, the drawbacks of increased price, power consumption and heat output might not be out-weighted by the potential increase in classification and uncertainty accuracy.

3.2.6 From Mean and Standard Deviation to Uncertainty

The MCDO, MCBN and ensemble method mentioned above all provide multiple predictions per input example. It makes the most sense to take the mean of these predictions as the final prediction, but it is not directly clear what the best way to calculate the is uncertainty.

For the multiple predictions for a single example $\mathbf{X}$ the standard deviation for all predictions of a certain class can be calculated as follows:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i-\mu)^2}, \tag{3.18}$$

where $x_1, x_2, ..., x_N$ are the single predictions in X, μ is the mean value of all predictions in $\mathbf{X}$, and N is the number of predictions per sample.

One method is to return σ as the uncertainty for that particular example. We will later refer to this approach as the *STD* (standard deviation) approach.

Another method is to use the mean μ and standard deviation σ to calculate the probability density function (PDF) of the Gaussian distribution $\mathcal{N}$ (the normal distribution) as follows:

$$pdf(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}. \tag{3.19}$$

The PDF provides the relative likelihood that the value of the random variables in the distribution would equal that of a sample x. We can use this function to calculate an individual likelihood for every probability that we encounter in $\mathbf{X}$. We can then convert this likelihood to an uncertainty through the following formula:

$$uncertainty(x) = 1 - \frac{pdf(x)}{pdf(\mu)}. \tag{3.20}$$

The PDF will return the highest likelihood for the mean value μ, so we assume this point to have 0(%) uncertainty. The further the likelihood of $pdf(x)$ moves from the value of $pdf(\mu)$, the higher the uncertainty will get, till a maximum of 1 (100%) for an example that has 0 likelihood to fall in the distribution. After this uncertainty is calculated for all probabilities in every single prediction, we can get the final uncertainties per class for that example by taking the mean of all individual uncertainties within that class.

We will later refer to this method as the *PDF* approach.

3.2.7 Addition of an Error Output

Neural networks can be used to approximate the real distribution $f(\vec{x})$ of a dataset. We assume that this dataset $d(\vec{x})$ can be modeled by:

$$d(\vec{x}) = f(\vec{x}) + n(\vec{x}), \tag{3.21}$$

where $n(\vec{x})$ is the noise. This noise can bee seen as the error on the target values that moves the target away from their true value $f(\vec{x})$.

In a classification task we can view the outputs of a network $d(\vec{x})$, given an input $y(\vec{x})$, as an estimation $\hat{\mu}(\vec{x})$ of the true mean $\mu(\vec{x})$ which represents the noisy distribution of the class to be classified. $\hat{\mu}(\vec{x})$ in classification tasks is converted to a probability for each of the possible classes.

To also quantify the uncertainty next to the probability, the network should simultaneously estimate the degree of noise of $\hat{\mu}(\vec{x})$ based on the noise observed in the training data. To achieve this, we want to simultaneously learn a function $e(\hat{x})$ that estimates the true error $E(\hat{x})$ of that distribution.

$\hat{\mu}(\vec{x})$ and $e(\hat{x})$ together can give us an approximation of the true distribution given by $\mu(\vec{x})$ and $E(\hat{x})$ [64, 65].

The network will, therefore, have two sets of output units; output units y corresponding to the classes used in the classification, and the error output e. The target for e will be calculated by taking the square of the error of y. The error of y is calculated by subtracting the true target value for y from the predicted value $d(\vec{x})$. For a classification task with multiple outputs, the error and subsequent target are calculated individually for each of the possible output classes. Both sets of outputs must have at least one unique layer that is not shared with the other outputs to allow for their own function to be learned.

To end up with an uncertainty estimation, two methods can be used:

- *Method 1*: The predicted error, as outputted by the network, is directly returned as the uncertainty estimation.
- *Method 2*: The predicted error, as outputted by the network, is first converted to an uncertainty estimation.

For *method 1* no additional computation is required. The estimated error of the error nodes can directly be interpreted as the uncertainty the model has in this prediction.

For *method 2* some additional computations are required. Since the output of the error nodes estimates the true error, we can treat the difference between the estimated error and the true error as the uncertainty of the model. However, for new examples, we do not have access to the true labels and therefore we cannot calculate the true error. To circumvent this problem we can, however, treat the highest prediction of the classification nodes as the 'true' label. Now the corresponding highest probability can be used to calculate the 'true' error.

For example for a classification task class 2 has the highest probability of 0.8. We now treat class 2 as being the 'true' label which allows us to calculate the 'true' (squared) error as follows: $(0.8-1)^2 = 0.04$. Here 1 is subtracted from the probability 0.8 since we assume this to be the true class which would need to receive a probability of 1 of being that class. If our estimated error output would be agreeing with the prediction of the classification outputs, we would expect to see an error output close to 0.04 as well. Assume the corresponding error estimation is 0.1. This would result in an uncertainty of $|0.1-0.04| = 0.06$, which would be 6%.

We can also calculate the uncertainties for the non-highest predicted classes by substituting the 1 above by a 0. If in the example above the true class is 2, it should have a 0 probability of being any other class. The outputted error should reflect this as well.

This could of course also be turned around, where a predicted squared error of 0.1 should have a corresponding probability prediction of $1-\sqrt{0.1} \approx 0.68$. The uncertainty would now be $|0.8-0.68| = 0.12$, or 12%. We did, however, use the first method.

Method 2 relies on the premise that, while training to predict the error, the network finds some new features and patterns in the data that the classifier has not found. For an accurate uncertainty estimate, the error output and classification output must be in unison when the model is predicting on similar data that it has been trained on, but they should disagree when the model is predicting on unknown data.

Error Output is most similar to an ensemble that only contains two networks which are trained on the same dataset. The main difference is that the addition of the Error Output method allows the network to learn its error instead of just the classification. This means that the network is actively learning its uncertainty. While not modelling the underlying distributions of the train data, as a Baseyian approach would do, it will still learn what kind of data it has encountered in the train data. This method captures the epistemic uncertainty.

The method is, however, more vulnerable to over-fitting than the other methods discussed, since it only consists of a single deep network. If, for example, the error outputs cannot generalise the train data, it might give high errors (and therefore uncertainties) to data outside of the train data even when that data is part of the classes it has been trained on.

Chapter 4

Experimental Setup

4.1 Datasets

4.1.1 Messidor-2

As the first dataset to test the methods on we used the medical Messidor-2 dataset [66] with 1748 annotated images of retinas. This dataset has been created to facilitate studies on computer-assisted diagnoses of diabetic retinopathy (DR). The dataset was divided into five different classes (by severity) of diabetic retinopathy. These labels were given by specialists working at hospitals partnered with the Messidor project[1]. The number of images for each class was highly imbalanced, as can be seen in Table 4.1. An immediate problem that arises from this imbalance is that a model which has not learned any representations of the classes but rather is always outputting class 1, will get an accuracy of around 58%. To mitigate this problem we decided to normalise the classes in the training data by five different random augmentations while keeping the imbalance in the validation and test set. We used 70% of the original data as the training set (which additionally got normalised by augmentation), 10% for validation and 20% for testing. The resulting division can be seen in Table 4.2.

The images were high resolution ($>$ 1500 x $>$ 1000 pixels RGB) cutouts of the iris on a black backdrop. All images were resized to 256x256x3 pixels RGB, with black pixels padded to the sides to retain the aspect ratio of the original images while having the iris centred. To counter the effects of the imbalanced dataset, we normalised the train data by performing augmentation. We took the number of examples of the class with most examples (which was 700) and set that amount times 1.2 as the goal amount for every class ($700x1.2 = 840$). We randomly selected examples of a certain class, performed 5 different augmentations on them (random rotation with a max range of 20°, random shearing with a max shear of 20%, random zooming with a max zoom of 20%, a random horizontal flip and a random change of brightness between 50% and 150%) and repeated this process until we had reached the target amount of 840.

The resulting number of examples per class for the train set was: [840, 840, 839, 838, 836].

The number of examples per class for the validation set (where no normalisation or augmentation was used) was: [119, 22, 28, 7, 3].

The number of examples per class for the test set (where no normalisation or augmentation was used) was: [202, 58, 70, 10, 6].

[1] http://www.adcis.net/en/third-party/messidor/

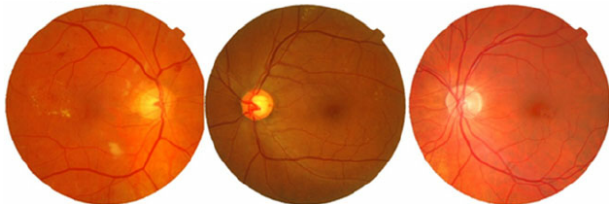


Figure 4.1: Examples of the images in the Messidor-2 dataset

	Class 1: No DR	Class 2: Mild DR	Class 3: Moderate DR	Class 4: Severe DR	Class 5: PDR
Number of examples	1021	270	347	75	35
Percentage of total	∼58.4%	∼15.4%	∼19.9%	∼4.3%	∼2%

Table 4.1: Division of examples in the Messidor 2 dataset

	Class 1: No DR	Class 2: Mild DR	Class 3: Moderate DR	Class 4: Severe DR	Class 5: PDR
Number of training examples	840	840	839	838	836
Number of validation examples	119	22	28	7	3
Number of test examples	202	58	70	10	6

Table 4.2: Division of examples in the Messidor 2 dataset splitted into a training, validation and test set. The training set is the only set normalised through augmentation.

4.1.2 CIFAR10

As the second dataset CIFAR10 [67] was used. This dataset is often used as a benchmark for novel machine learning techniques. The CIFAR-10 dataset consists of 60,000 32x32x3 RGB images in 10 classes, with an even 6000 images per class. There are 50,000 training images and 10,000 test images. The test images were split into a validation set and a test set in a $\frac{1}{3}$ division. The division of examples per set per class can be seen in Table 4.3. No augmentation was performed. Examples of the images and classes in CIFAR10 can be seen in Figure 4.2.

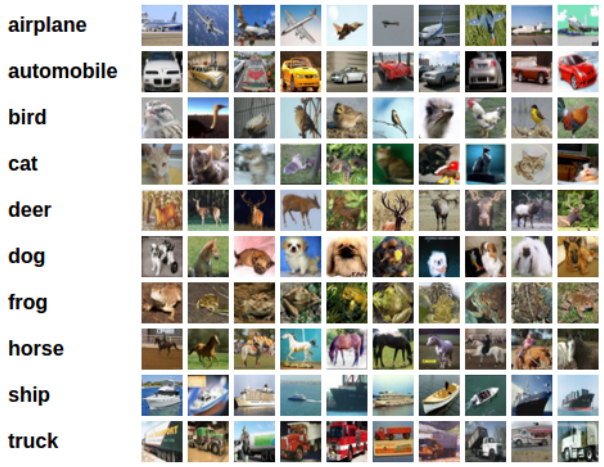


Figure 4.2: Examples of the ten classes in the CIFAR10 dataset

	Class 1: Airplanes	Class 2: Automobile	Class 3: Bird	Class 4: Cat	Class 5: Deer	Class 6: Dog	Class 7: Frog	Class 8: Horse	Class 9: Ship	Class 10: Truck
Number of training examples	5000	5000	5000	5000	5000	5000	5000	5000	5000	5000
Number of validation examples	139	115	122	105	118	122	127	129	131	142
Number of test examples	861	885	878	895	882	875	873	871	869	858

Table 4.3: Division of examples in the CIFAR10 dataset

4.1.3 MNIST

We used the handwritten digit dataset MNIST [68] to test our trained models on *out of distribution* examples (classes the network was not trained on). The dataset consists of the 10 Arabic numerals (0 till 9) all handwritten, centred, greyscale images of 28x28 pixels. The dataset contains 60,000 train images and 10,000 test images.

We did not train any of the networks on the images in MNIST, we only used the test set (after an appropriate resizing to the required input size of the network in question) to test the performance of the different methods on data it has not been trained on. An appropriate response to out of distribution data is to give a high uncertainty. An example of the images and classes in MNIST can be seen in Figure 4.3.

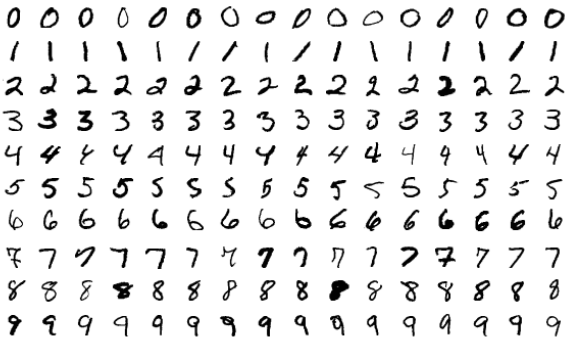


Figure 4.3: Examples of the ten classes in the MNIST dataset

4.2 Networks

We compared three of the four methods (MCDO, MCBN and Error Output) on the same VGG16 architecture [69]. We picked this network for its proven performance on image classification tasks and its wide use as a benchmark architecture, as well as the lack of any SRTs incorporated into the architecture. For the Ensemble method, we used 13 additional architectures which can be seen in Table 4.5. We used the pre-trained weights[2] on the ImageNet dataset [70] as the starting weights of the network (excluding the last 3 Dense layers). We will refer to fully connected layers with a non-linear activation function as Dense layers. These weights are the result of training the network for object recognition. We used the assumption that the basic features found for the object classification task for ImageNet are also useful for our (transfer learning) classification task. The full VGG16 consists of a total of 13 Convolutional layers, 3 Dense layers and 5 MaxPooling layers. The network can be seen as a combination of 6 distinct blocks. The first two blocks consist of two Convolutional layers followed by a MaxPooling layer. The middle three blocks consist of three Convolutional layers followed by a MaxPooling layer. The last block is the top of the network and consists of three Dense layers. Each Convolutional layer uses the ReLU (Section 2.5) activation function. All Dense layers except the very last (output) Dense layer use the ReLU activation function as

[2]https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5

Dropout Layer:	1	2	3	4	5	6	7
Dropout Rate:	0.05	0.1	0.15	0.2	0.2	0.2	0.25

Table 4.4: Dropout Rates per layer added to VGG16

well. The output Dense layer uses the Softmax activation function (Section 2.5). A visual representation of the network can be seen in Figure 4.4.

To allow for a fair comparison we tried to minimise adjustments to the networks for individual methods and we tried to keep the hyper-parameters for training as similar as possible.

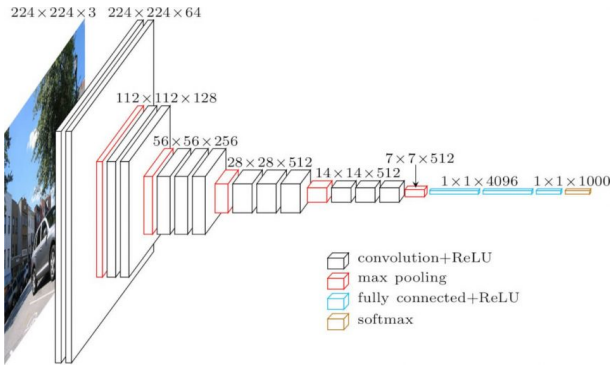


Figure 4.4: Architecture of the VGG16

4.2.1 Monte Carlo Dropout

To perform Monte Carlo Dropout (MCDO) as mentioned in Section 3.2.3, we added a Dropout layer after every MaxPooling and Dense layer (except the last Dense classification layer). A total of 7 Dropout layers were added to the network. Table 4.4 shows the Dropout rates used per layer. The rates are chosen to be fairly low since the use of multiple Dropout layers all with high Dropout rates can result in a network with no connections from input to output anymore. The Dropout rates gradually increase since the low-level features extractors early in the network will generally introduce less variance than the higher-level layers later in the network. This is because a lot of these low-level features, e.g. the detection of a line at a certain angle, will not be unique to the classes the network is trained on.

After training we perform 250 [3] forward-passes through the network per example in the test set. For each forward-pass, the network has different active connections because random Dropout masks are taken (with the same probability as during training, see Table 4.4). The final probabilities for each class are the mean of all 250 predictions, where the highest mean probability gives the final predicted class. We discuss two different methods to compute the uncertainty of these predictions in Section 5.2.

Training and Optimisation

The settings of the final MCDO network trained on Messidor-2 and on CIFAR10, found after stochastically testing various variations of settings, can be seen in Table 4.6 and Table 4.7.

We used early stopping to stop training if no improvements of at least 0.5% were made in the last 5 epochs to the validation accuracy. If the validation loss improved after an epoch, this improved model was saved. When early stopping stopped training, the previous best model was loaded again and used as the final model. We used the categorical cross-entropy function (Section 2.3) as loss function and Adam (Section 2.4.4) as the optimiser.

[3] more will result in more reliable accuracy and uncertainty predictions (to a certain degree), but will negatively influence training and inference time

4.2.2 Monte Carlo Batch Normalisation

To perform Monte Carlo Batch Normalisation (MCBN) as mentioned in Section 3.2.4, we added a batch normalisation (BN) layer after every Convolutional and Dense layer (except the last Dense classification layer). At training time the 'standard deviation' parameter *gamma* (γ) and the 'mean' parameter *beta* (β) of the BN layers will be trained (as can be seen in Algorithm 2), next to all other layers.

For testing, the weights of all layers were frozen. Next, a random mini-batch was sampled from the training data to compute a new mean and variance for the batch normalisation layers of the network. Now, inference was performed on the test set by this new, slightly adjusted network. The resulting predictions were stored. This process was repeated 250 [3] times per example in the test set. The final probabilities for each class were the mean of all 250 predictions, where the highest mean probability gave the final predicted class. We discuss two different methods to compute the uncertainty of these predictions in Section 5.2.

Algorithm 2: Batch Normalisation Transform, applied to activation x over a mini-batch.

Input: Values of x over a mini-batch $\mathcal{B} = \{x_1 ... m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \rightarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2\mathcal{B} \rightarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \rightarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} \qquad \text{// normalise}$$

$$y_i \rightarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Training and Optimisation

The settings of the final MCBN network trained on Messidor-2 and on CIFAR10, found after stochastically testing various variations of settings, can be seen in Table 4.6 and Table 4.7.

We used early stopping to stop training if no improvements of at least 0.5% were made in the last 5 epochs to the validation accuracy. If the validation loss improved after an epoch, this improved model was saved. When early stopping stopped training, the previous best model was loaded again and used as the final model. We used the categorical cross-entropy function (Section 2.3) as loss function and SGD with momentum (Section 2.4.3) as the optimiser.

4.2.3 Ensemble

To perform uncertainty estimations with an Ensemble, as mentioned in Section 3.2.5, we used 14 different network architectures. The architectures used can be found in Table 4.5.

Each architecture was trained 3 times, each time on a unique $\frac{1}{3}$ of the training data. This resulted in a total of 42 trained networks. Due to random initialisation of the weights, the difference in network architecture and the unique part of the training data each of the 3 networks with the same architecture was trained on, each trained network performs slightly different. All 42 networks performed inference on the

complete test set and the resulting predictions were stored. The final predictions were the mean of all the predictions, and the final uncertainties were the standard deviation of all predictions.

Training and Optimisation

The settings of the final Ensemble network trained on Messidor-2 and on CIFAR10, found after stochastically testing various variations of settings, can be seen in Table 4.6 and Table 4.7.

We used early stopping while training each model in the ensemble to stop training if no improvements of at least 0.5% were made in the last 5 epochs to the validation accuracy. If the validation loss improved after an epoch, this improved model was saved. When early stopping stopped training, the previous best model was loaded again and used as the final model in the ensemble. We used the categorical cross-entropy function (Section 2.3) as loss function and Adam (Section 2.4.4) as the optimiser.

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121

Table 4.5: Overview of the different architectures used for the Ensemble, together with their respective performance on the ImageNet validation dataset

4.2.4 Error Output

To perform Error Output as mention in Section 3.2.7, we added two separate Dense output layers to VGG16. Both, the *classification* and the *error* layer had nodes equal to the number of classes. The classification layer used the softmax activation function (since it is outputting a probability for each class), while the error layer used a linear activation function (the errors do not yet represent an uncertainty probability at this point and therefore should not be normalised). Nix et al. found that, when using only one hidden layer, connecting the classification layer y and output layer e to their own set of hidden units works better than connecting them to a shared set of hidden units [65]. Therefore, also the second to last Dense layer was separated into two equally sized Dense layers both with 4096 nodes. Both these layers were connected to the same third-to-last Dense layer, but their output was connected to either the classification layer or the error layer. This ensures that both layers had enough capacity to learn their own function.

Training and Optimisation

The settings of the final Error Output network trained on Messidor-2 and on CIFAR10, found after stochastically testing various variations of settings, can be seen in Table 4.6 and Table 4.7.

The model was trained in three consecutive successions, each with a different loss function.

During the first training round, only the error output was used for training. The loss function, a modified version of the mean squared error, only considered the outputs of the error layer and after each training step, the parameters of the network were adjusted accordingly. Early stopping stopped training when no improvements to the training loss were made for 10 epochs. Recall that the *target* error for the error outputs is the *true* error calculated on the classification outputs. Not training the classification outputs actively results in a high variance in the *true* error, which allows the error outputs to learn how to output high errors in case of uncertainty. The model with the lowest loss on the training data was used for the following training round.

During the second training round, only the classifier output was used for training. The loss function, a modified version of categorical cross-entropy, then only considered the outputs of the classification layer and after each training step, the parameters of the network were adjusted accordingly. Early stopping stopped training when no improvements to the validation loss were made for 10 epochs. The model with the lowest loss on the validation data was used for the following training round. For both datasets the model with the lowest loss was reached after 1 epoch. The weights found during the first training round were also relevant for the classifier.

During the last training round both the classifier and error outputs were used for training. The loss function then summed the error calculated by the categorical cross-entropy function on the classification output and the mean squared error calculated on the error output. Early stopping stopped training when no improvements to the training loss were made for 2 epochs. The model with the lowest loss on the training data was used as the final model.

We used Adam (Section 2.4.4) as the optimiser for each of the three consecutive successions.

	Batch size	Epochs	Pretrained weights	Learn rate	Optimiser
MCDO	64	Early stopping after: 91 Best model picked: 13	ImageNet	0.00001	Adam
MCBN	180	Early stopping after: 77 Best model picked: 74	ImageNet	0.01	SGD (momentum = 0.9)
Ensemble	32	Early stopping after: * Best model picked: *	ImageNet	0.00001	Adam
Error Output	64	Early stopping after: 118/26/31† Best model picked: 109/1/29†	ImageNet	0.00001	Adam

Table 4.6: Final settings for each method when trained on the Messidor-2 dataset.
* Each network inside the Ensemble is stopped after a different number of epochs.
† Error Output is trained 3 times in succession.

	Batch size	Epochs	Pretrained weights	Learn rate	Optimiser
MCDO	32	Early stopping after: 14 Best model picked: 9	ImageNet	0.00001	Adam
MCBN	32	Early stopping after: 18 Best model picked: 3	ImageNet	0.001	SGD (momentum = 0.9)
Ensemble	32	Early stopping after: * Best model picked: *	ImageNet	0.00001	Adam
Error Output	128	Early stopping after: 101/11/13† Best model picked: 91/1/9†	ImageNet	0.00001	Adam

Table 4.7: Final settings for each method when trained on the CIFAR10 dataset.
* Each network inside the Ensemble is stopped after a different number of epochs.
† Error Output is trained 3 times in succession.

Chapter 5

Results

In this chapter, we will present the results from the experiments described in Section 4. Recall that these experiments were designed to answer the following research question:

Which method provides the best uncertainty estimations in deep neural networks for the classification of images?

This research question is decomposed into two sub-questions:

- **Which method yields the best performance?**

  This question will be answered by the results shown in Section 5.1 and Section 5.2.

- **Does testing on a different image dataset clearly show uncertainty?**

  This question will be answered by the results shown in Section 5.2.

5.1 Model Performance

Table 5.1 and Table 5.2 show the performance of each method when trained on Messidor-2 and CIFAR10 respectively. All networks were trained on a node from the Peregrine cluster from the University of Groningen. This node had the following specifications:

- CPU: Intel® Xeon® Gold 6150 @ 2.70GHz (12 cores) (virtualized)
- Memory: 128 GB
- GPU: NVIDIA V100 (32 GB VRAM)

	Training time (HH:MM:SS):	Inference time Messidor-2 test set (HH:MM:SS.ss):	Inference time MNIST test set (HH:MM:SS.ss):	Accuracy on Messidor-2 test set:	Size trained model (MB):
MCDO	Early stopping: 00:41:12 Best model: 00:06:06	00:05:00 Per example: 00:00:00.65	01:20:54 Per example: 00:00:00.49	63.0% (62.1% †)	663
MCBN	Early stopping: 00:44:01 Best model: 00:42:19	00:07:19 Per example: 00:00:00.96	01:18:07 Per example: 00:00:00.47	29.2% * (63.2% †)	634
Ensemble	Early stopping: 06:29:38 Best model: 02:53:12	00:11:10 Per example: 00:00:01.46	00:21:10 Per example: 00:00:00.12	67.3%	73,505
Error Output	Early stopping: 01:21:06 Best model: 00:59:37	00:00:05 Per example: 00:00:00.01	00:00:20 Per example: 00:00:00.002	61.7%	633

Table 5.1: Performance when trained on the Messidor-2 dataset.
* The training set caused problems with batch normalisation due to a lack of variance
† Accuracy when the complete trained network was used without the MCDO or MCBN method

	Training time (HH:MM:SS):	Inference time CIFAR10 test set (HH:MM:SS.ss):	Inference time MNIST test set (HH:MM:SS.ss):	Accuracy on CIFAR10 test set:	Size trained model (MB):
MCDO	Early stopping: 00:05:13 Best model: 00:02:11	00:02:17 Per example: 00:00:00.016	01:30:29 Per example: 00:00:00.543	85.4% (85.1% *)	129
MCBN	Early stopping: 00:07:19 Best model: 00:01:19	00:04:02 Per example: 00:00:00.025	01:21:53 Per example: 00:00:00.490	83.4% (82.8% *)	130
Ensemble	Early stopping: 14:12:51 Best model: 4:36:11	00:08:05 Per example: 00:00:00.056	00:10:47 Per example: 00:00:00.011	79.3%	14,828
Error Output	Early stopping: 00:22:58 Best model: 00:18:48	00:00:03 Per example: 00:00:00.0004	00:00:25 Per example: 00:00:00.003	83.7%	194

Table 5.2: Performance when trained on the CIFAR10 dataset
* Accuracy when the complete trained network was used without the MCDO or MCBN method

5.2 Uncertainty

The methods *MCDO*, *MCBN* and *Ensemble* all return an output matrix $\mathbf{O}$. We used two different methods to convert the information in these output matrices to uncertainty approximations. The structure of $\mathbf{O}$ is as follows: $\mathbf{O} = A, E, C$, where A represents the different model configurations that made a prediction on the complete test set, E represents the examples in the test set and C represents the softmax probabilities for each class.

5.2.1 Uncertainty Method 1 - Standard Deviation

The first method to gain an uncertainty approximation from the output matrix calculates the standard deviation (STD) of all the N predictions made for each of the examples e_i per class c_i by the different

model configurations in A. The STD is calculated as follows:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}, \tag{5.1}$$

where $x_1, x_2, ..., x_N$ are the predicted probabilities for a certain class c and $\bar{x}$ is the mean value of these probabilities.

The resulting matrix is E by C and holds the STDs for each class of each example.

5.2.2 Uncertainty Method 2 - Probability Density Function

For the second method, we use the mean and standard deviations calculated in *method 1* to construct a Gaussian distribution for each class in each example. We then calculate the probability density function (PDF) of the mean for this distribution. This will be the highest probability any example can get of falling in this distribution, and therefore will be given 0 uncertainty. Now the PDF for each individual predicted uncertainty will be computed and compared to the PDF of the mean. The further away this probability is from the PDF of the mean, the higher the uncertainty.

Details about both methods can be read in section 3.2.6.

5.2.3 Scatter Plots

Section 5.2.4 till 5.2.7 show the results of each method when trained on either Messidor-2 or CIFAR10. Section 5.2.4 till 5.2.6 show two figures (with each figure containing three plots) per dataset. The first figure shows the scatter plot when *method 1* (STD) is used to compute the uncertainty. The second figure shows the scatter plot when *method 2* (PDF) is used to compute the uncertainty.

Section 5.2.7 shows two figures per dataset as well. Here, the first and second figure depicts the scatter plot when the uncertainty is calculated via method 1 and method 2 as described in section 3.2.7 respectively.

Per figure three scatter plots are shown.

The leftmost plot (a) shows the highest probability of each example in the test set plotted against the uncertainty of that example. The green dots represent correct predictions, the red dots wrong predictions.

The middle plot (b) shows the mean squared error (MSE) plotted against the uncertainty. The MSE is calculated as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2, \tag{5.2}$$

where Y is the vector of all highest predicted probabilities for a certain example and $\hat{Y}$ is the true label for that example. For example if the highest predicted probability of a certain network instance for a certain example is 0.7, and this prediction is correct (so the true label would be 1), the squared error is $(0.7-1)^2 = 0.09$. The MSE for a certain example would be the mean of the squared errors calculated for each prediction made on that example.

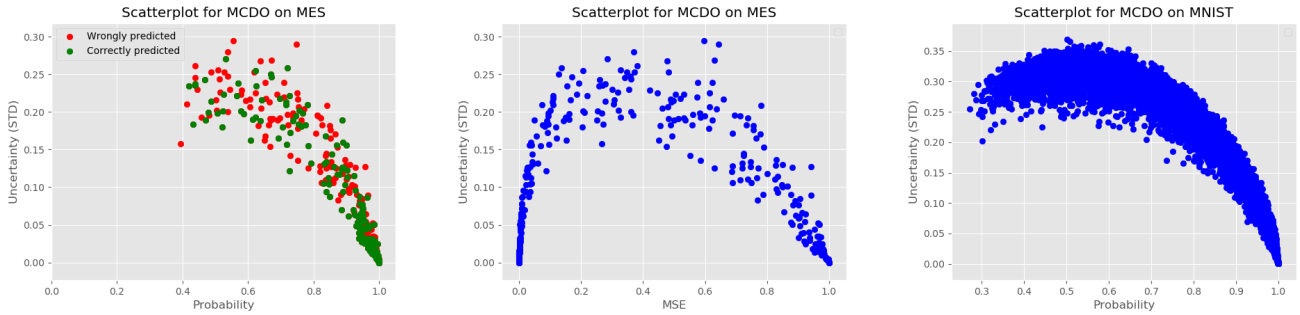
If the middle plot (b) would be showing a linearly increasing correlation, the uncertainty would be increasing when the network's predictions are less accurate as well.

The rightmost plot (c) shows the highest probability for each example in the MNIST test set plotted against the uncertainty. Since none of the networks were trained on MNIST and therefore in no way able to give accurate predictions, the methods should express their confusion through a high uncertainty.

5.2.4 Monte Carlo Dropout

In Figure 5.1 till Figure 5.4 the scatter plot results of MCDO can be seen. Figure 5.1 and Figure 5.2 show the results when the network is trained on the (augmented) Messidor-2 train set, Figure 5.3 and 5.4 show the results when trained on the CIFAR10 train set.

When trained on Messidor-2

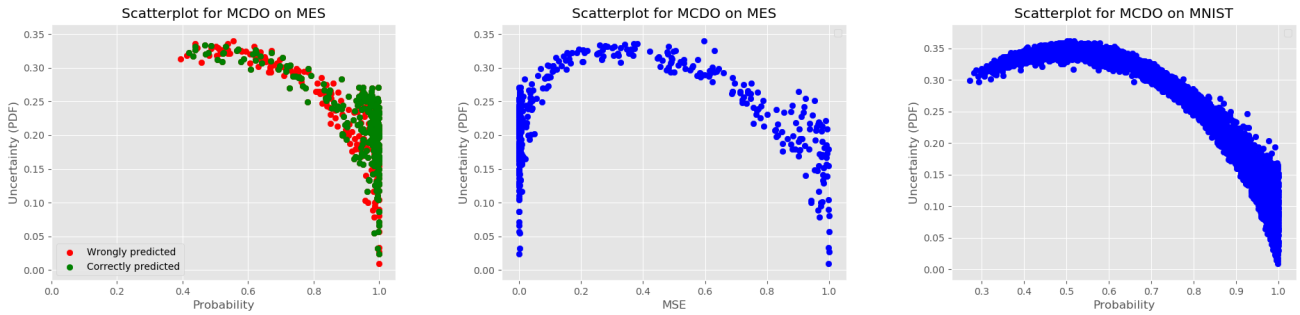


(a) Effect of probability on uncertainty on Messidor-2 test set

(b) Effect of mean squared error on uncertainty on Messidor-2 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.1: Scatterplots for MCDO when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2



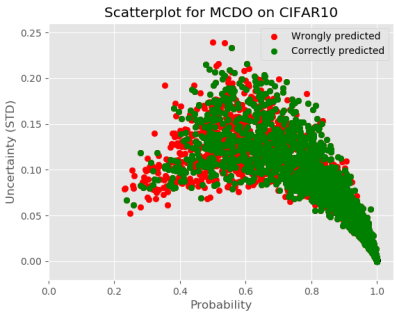
(a) Effect of probability on uncertainty on Messidor-2 test set

(b) Effect of mean squared error on uncertainty on Messidor-2 test set

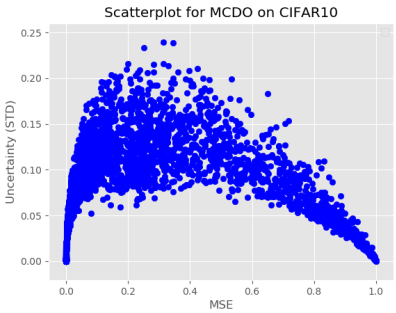
(c) Effect of probability on uncertainty on MNIST test set

Figure 5.2: Scatterplots for MCDO when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2

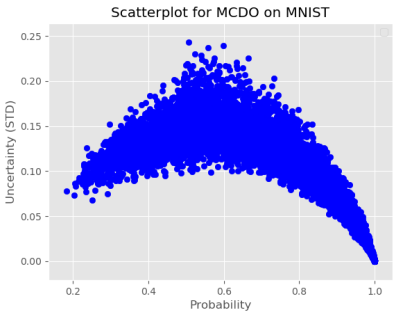
When trained on CIFAR10



(a) Effect of probability on uncertainty on CIFAR10 test set

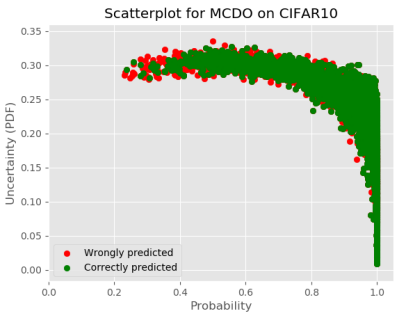


(b) Effect of mean squared error on uncertainty on CIFAR10 test set

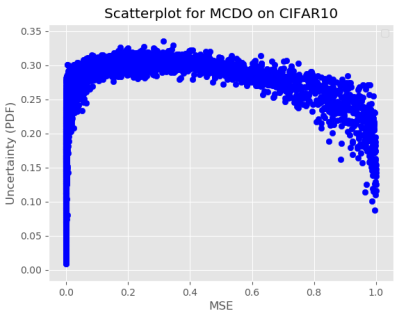


(c) Effect of probability on uncertainty on MNIST test set

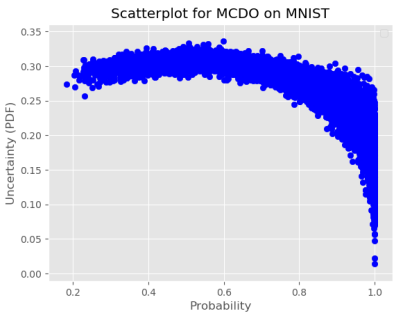
Figure 5.3: Scatterplots for MCDO when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10



(a) Effect of probability on uncertainty on CIFAR10 test set



(b) Effect of mean squared error on uncertainty on CIFAR10 test set



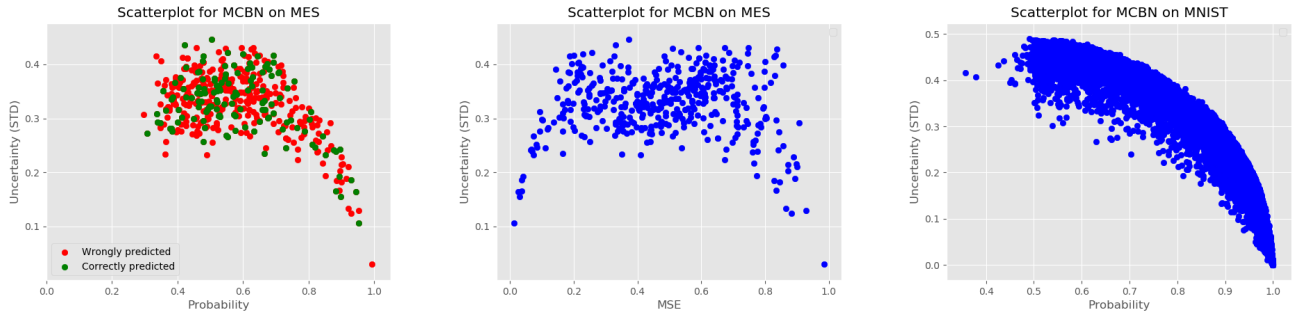
(c) Effect of probability on uncertainty on MNIST test set

Figure 5.4: Scatterplots for MCDO when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10

5.2.5 Monte Carlo Batch Normalisation

In Figure 5.5 till Figure 5.8 the scatter plot results of MCBN can be seen. Figure 5.5 and Figure 5.6 show the results when the network is trained on the (augmented) Messidor-2 train set, Figure 5.7 and 5.8 show the results when trained on the CIFAR10 train set.

When trained on Messidor-2

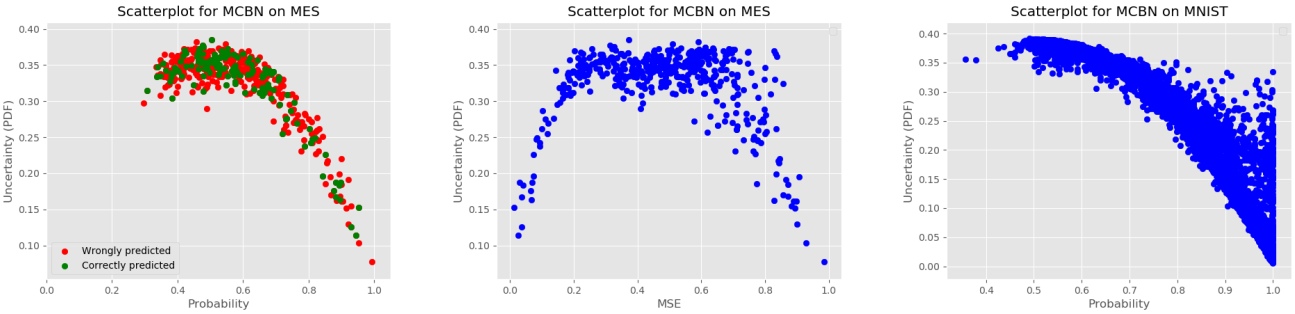


(a) Effect of probability on uncertainty on Messidor-2 test set

(b) Effect of mean squared error on uncertainty on Messidor-2 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.5: Scatterplots for MCBN when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2



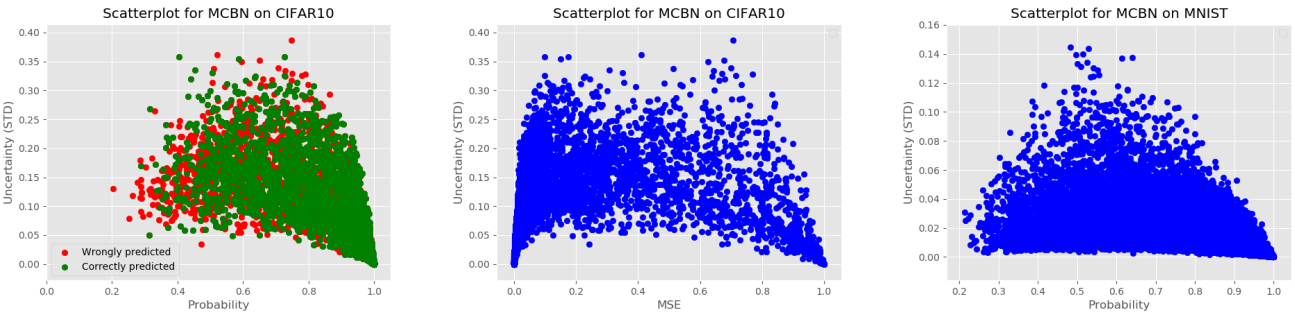
(a) Effect of probability on uncertainty on Messidor-2 test set

(b) Effect of mean squared error on uncertainty on Messidor-2 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.6: Scatterplots for MCBN when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2

When trained on CIFAR10

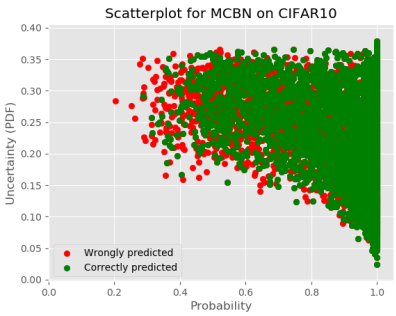


(a) Effect of probability on uncertainty on CIFAR10 test set

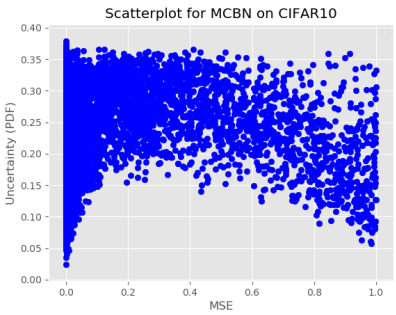
(b) Effect of mean squared error on uncertainty on CIFAR10 test set

(c) Effect of probability on uncertainty on MNIST test set

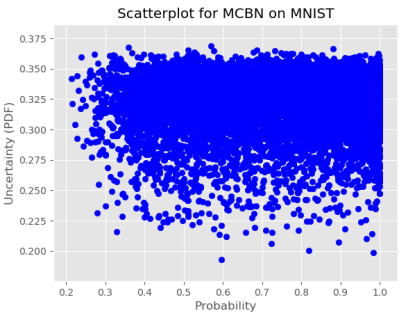
Figure 5.7: Scatterplots for MCBN when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10



(a) Effect of probability on uncertainty on CIFAR10 test set



(b) Effect of mean squared error on uncertainty on CIFAR10 test set



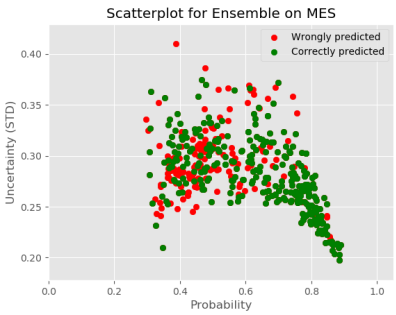
(c) Effect of probability on uncertainty on MNIST test set

Figure 5.8: Scatterplots for MCBN when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10

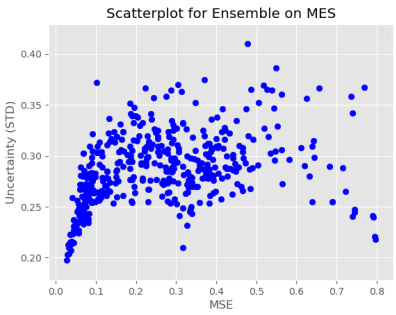
5.2.6 Ensemble

In Figure 5.9 till Figure 5.12 the scatter plot results of Ensemble can be seen. Figure 5.9 and Figure 5.10 show the results when the network is trained on the (augmented) Messidor-2 train set, Figure 5.11 and 5.12 show the results when trained on the CIFAR10 train set.

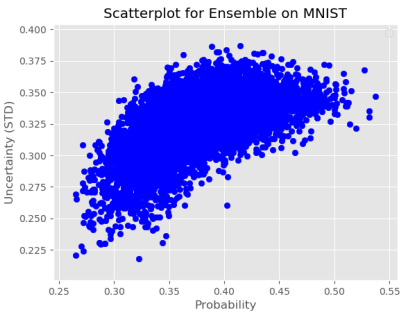
When trained on Messidor-2



(a) Effect of probability on uncertainty on Messidor-2 test set

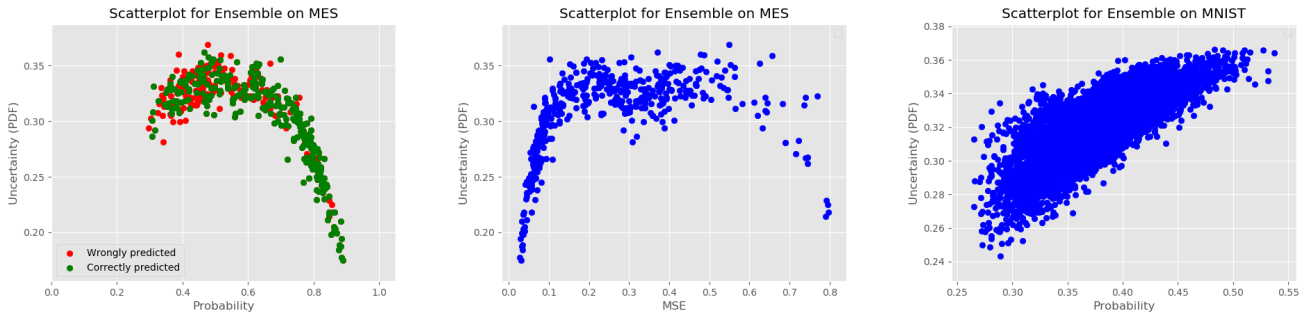


(b) Effect of mean squared error on uncertainty on Messidor-2 test set



(c) Effect of probability on uncertainty on MNIST test set

Figure 5.9: Scatterplots for Ensemble when the uncertainty is computed via method 1 (standard deviation) and the network is trained on Messidor-2



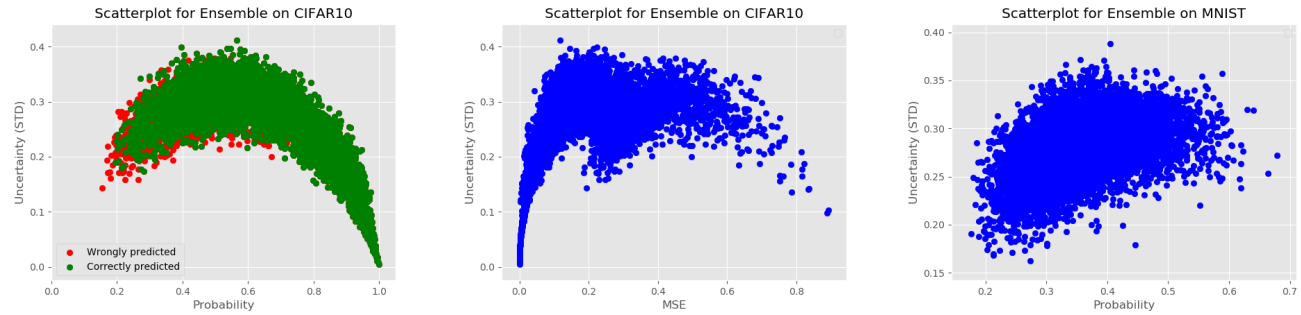
(a) Effect of probability on uncertainty on Messidor-2 test set

(b) Effect of mean squared error on uncertainty on Messidor-2 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.10: Scatterplots for Ensemble when the uncertainty is computed via method 2 (probability density function) and the network is trained on Messidor-2

When trained on CIFAR10

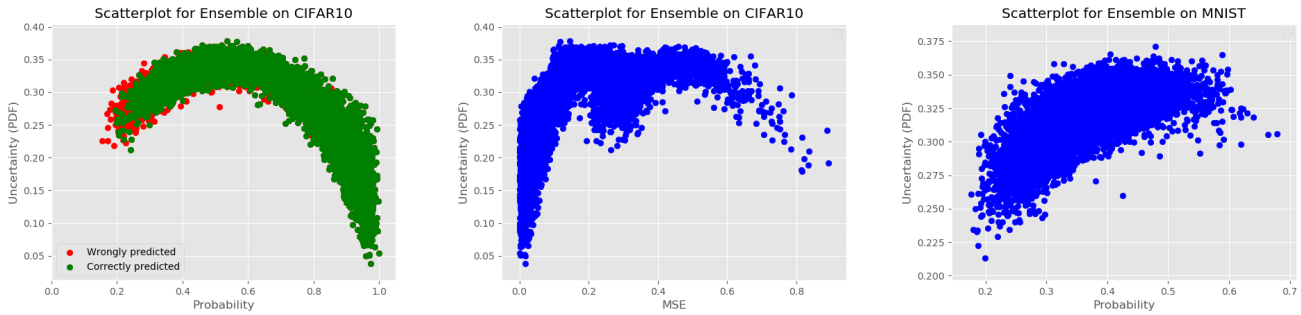


(a) Effect of probability on uncertainty on CIFAR10 test set

(b) Effect of mean squared error on uncertainty on CIFAR10 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.11: Scatterplots for Ensemble when the uncertainty is computed via method 1 (standard deviation) and the network is trained on CIFAR10



(a) Effect of probability on uncertainty on CIFAR10 test set

(b) Effect of mean squared error on uncertainty on CIFAR10 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.12: Scatterplots for Ensemble when the uncertainty is computed via method 2 (probability density function) and the network is trained on CIFAR10

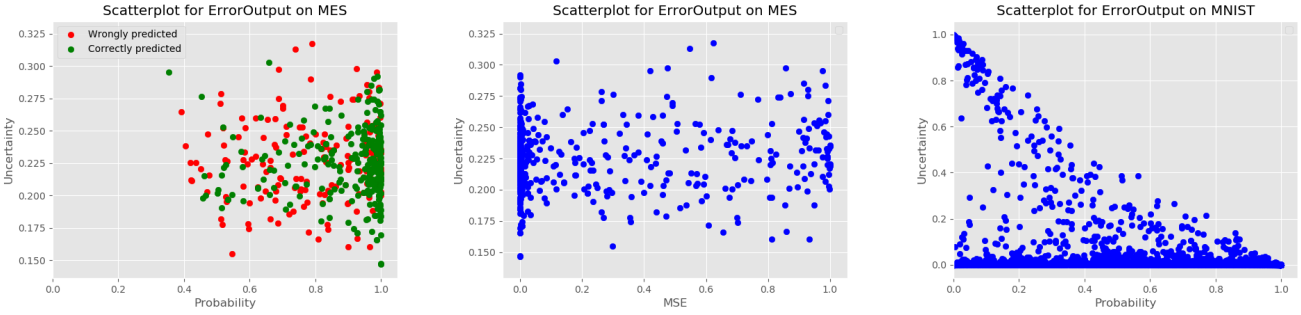
5.2.7 Error Output

As mentioned in section 3.2.7, two methods were used to receive an error estimation from a network trained through the *Error Output* method.

For the first method, as can be seen in Figure 5.13 and Figure 5.15, the raw output from the error outputs is taken as the uncertainty.

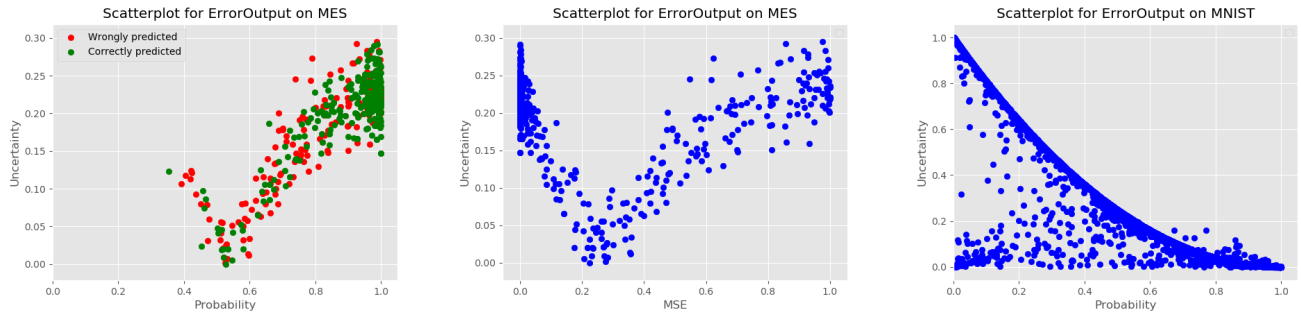
For the second method, as can be seen in Figure 5.14 and Figure 5.16, the raw error is compared to the 'true' error (the error that the predicted probability would yield if the relevant class was correct (for the highest probability of all classes) or wrong (for all other classes)) to gain an uncertainty. If the predicted probability and error are compatible with each other, the uncertainty is low. If not, the uncertainty is high. Section 3.2.7 explains this method in more detail.

When trained on Messidor-2



(a) Effect of probability on uncertainty on Messidor-2 test set (b) Effect of mean squared error on uncertainty on Messidor-2 test set (c) Effect of probability on uncertainty on MNIST test set

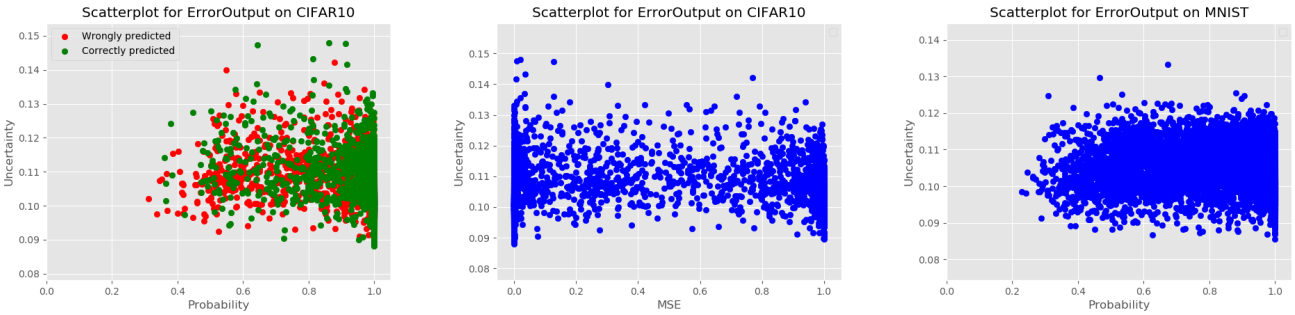
Figure 5.13: Scatterplots for Error Output when the uncertainty is raw output of the error nodes and the network is trained on Messidor-2



(a) Effect of probability on uncertainty on Messidor-2 test set (b) Effect of mean squared error on uncertainty on Messidor-2 test set (c) Effect of probability on uncertainty on MNIST test set

Figure 5.14: Scatterplots for Error Output when the uncertainty is computed via the 'true' error and the network is trained on Messidor-2

When trained on CIFAR10

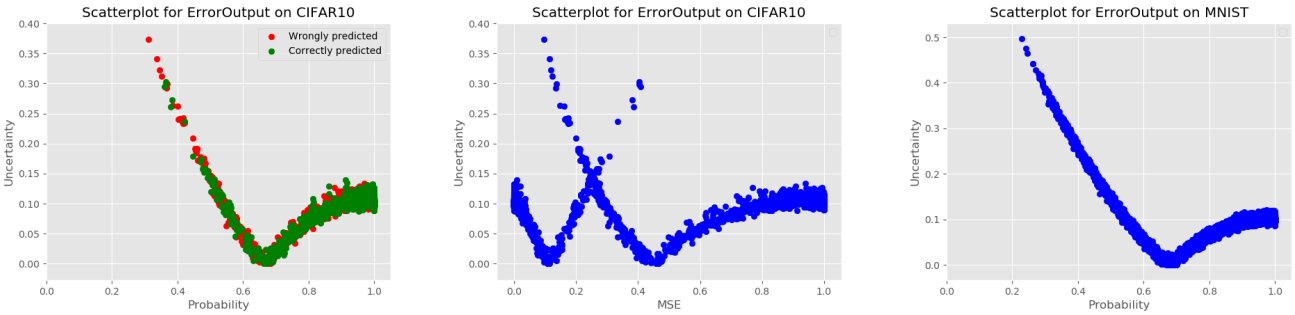


(a) Effect of probability on uncertainty on CIFAR10 test set

(b) Effect of mean squared error on uncertainty on CIFAR10 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.15: Scatterplots for Error Output when the uncertainty is raw output of the error nodes and the network is trained on CIFAR10



(a) Effect of probability on uncertainty on CIFAR10 test set

(b) Effect of mean squared error on uncertainty on CIFAR10 test set

(c) Effect of probability on uncertainty on MNIST test set

Figure 5.16: Scatterplots for Error Output when the uncertainty is computed via the 'true' error and the network is trained on CIFAR10

5.3 Discussion

In this section, we will discuss the findings of the first part of Chapter 5. Per method, we will discuss if the method has been able to answer our research questions of Section 1.5.

5.3.1 Performance

We can see the performance of each method in Table 5.1 and 5.2. We will elaborate on these results in the following sections.

Training time

MCDO was the fastest to train, next was MCBN, followed by Error Output and the slowest method was Ensemble. The training times for MCDO and MCBN are very similar. This is to be expected since only the addition of either Dropout or BN layers is the difference between the two resulting networks. The addition of Dropout generally has less of an impact on training than the addition of BN, a result we see replicated here [71].

Error Output uses two additional Dense layers and trains three consecutive times with different loss functions, causing it to be slower than the MCDO and MCBN. Ensemble was unsurprisingly the slowest since it had to train 42 different networks. Even though the networks were only trained on $\frac{1}{3}$ of the training data individually, Ensemble was still around a factor 10 slower on Messidor-2 and more than a factor 100 slower on CIFAR10 compared to MCDO and MCBN. This is because some of the architectures used in the Ensemble are a lot slower and deeper than VGG16 used for the other methods.

Inference time

Error Output excels at inference time since it only requires one forward-pass per example. Error Output had an average inference time of just 2 to 10 ms, which was around 60 to 200 times faster than the other methods. Ensemble, which had to load and perform inference on 42 networks, was the second-fastest when it had to perform inference on a large set of data (MNIST). When Ensemble had to perform inference on a smaller test set (Messidor-2 and CIFAR10), it was the slowest since it needs to load each of the 42 models before inference.

MCDO was slightly faster than MCBN. Since both methods had to adjust and perform inference on their network 250 times, they were a lot slower than Error Output and Ensemble when dealing with big datasets.

Accuracy

Messidor-2 proved to be a difficult dataset to classify correctly. This is probably due to the non-precise nature of the ground truth labels, where specialists classified the perceived severity of diabetic retinopathy (DR). The difference between, for example, mild and moderate DR is not easily quantifiable and a look at the confusion matrices of each method showed us that, when the wrong prediction was made, it was often only one class away from the true class. Combined with the fact that the dataset was imbalanced (see Table 4.1), the accuracy fell between 61.7% and 67.3%, with the highest accuracy obtained by the Ensemble. The Ensemble method reaching the highest accuracy was in line with other literature [72].

MCBN was not able to receive an acceptable accuracy on the Messidor-2 test set. During the training, it reached high training and validation accuracy, but this was not reflected in the test accuracy of 29.2%. A possible explanation for these bad results would be a lack of variance in the mini-batch (128 random examples from the training data) used to adjust the mean and variance of the BN layers when the mini-batch was too small. Experiments in which only the original data was used (the augmented set could have less variance since multiple examples could come from the same original image) or the mini-batch size was increased to 180 (we could not test higher sizes due to memory limitations) did not resolve this issue and only attained a slightly higher accuracy of 45.3%.

CIFAR10 proved to be a more balanced dataset with more variance between the classes. This resulted in high accuracies between 79.3% for the Ensemble and 85.4% for MCDO. The Ensemble performed notably worse on this dataset, which was probably because it was only able to train each network on $\frac{1}{3}$ of the train data. When each network was trained on the complete train data the accuracy was notably higher, but at an increased training time and a drop in the variance used for the uncertainty estimate.

On CIFAR10, MCBN did not have the same problems as it had on the Messidor-2 test set. CIFAR10 has more variance between the classes which further reinforces the hypothesis that MCBN performs bad on Messidor-2 because of a lack of variance in the mini-batches.

Testing the complete trained network of MCDO and MCBN resulted in a lower accuracy score than with the methods applied (except for MCBN on the Messidor-2 dataset where the complete trained network did not have a suspected variance problem). Since MCDO and MCBN combine the predictions of multiple networks into a single mean prediction we can see that their accuracy increases as it does for ensembles [72].

Model complexity

The model complexity of MCDO and MCBN was nearly equal since they both have an equal amount of layers. The model complexity of Error Output was slightly higher since it added two Dense layers with many trainable weights. Ensemble had by far the highest model complexity since it consisted of 14 different architectures, each trained 3 times.

This was also reflected in the memory requirements of the trained models.

After being trained on Messidor-2 (where the input as 256x256x3) MCDO, MCBN and Error Output were all around 650 MB in size. Ensemble, however, was 73,505 MB in size, resulting in an average size per network of around 1750 MB. Ensemble, therefore, was nearly 115 times bigger in terms of memory than the other methods.

We see similar results when trained on the smaller images of the CIFAR10 dataset (32x32x3). MCDO and MCBN were around 130 MB, Error Output was 194 MB and Ensemble was 14,828 MB (for an average of around 350 MB per network, or around 100 times bigger in total than the other methods).

5.3.2 Uncertainty

We can see the uncertainty estimates in Figure 5.1 till 5.16. We will elaborate these results in the following sections.

Uncertainty on original test data

Figure **a** of Figure 5.1 till 5.16 show us that the uncertainty estimates of none of the methods gave us information to better distinguish between *false positives* (red) and *true positives* (green).

For the Error Output method, the reason was perhaps easy to find. Error Output can only give a high estimated error if it thinks the classifier layer is going to classify this example wrong. To do this, it needs to detect some features in the input that tell it that this is a hard to classify example. But if the Error layer was able to detect these features, why wouldn't the classifier layer be able to do the same thing and use these features to make a correct prediction in the first place? It probably would, and therefore both output layers would almost always be in correspondence with each other. This means that when the error outputs predict a higher error because of uncertainty, the classifier has also already picked up on this and therefore is not giving a high probability to a single class. This idea was reflected in the results as well.

For MCDO, MCBN and Ensemble we can see that when the probability lowers, the uncertainty increases as well. We usually saw the highest uncertainty when the probability was around 0.5. Gal, in his thesis where he introduced MCDO, reported similar results when MCDO was used for image classification [11]. Uncertainties were found to be the highest when the probability was around 0.5 and lower when the probabilities either got higher or lower. The performance of MCBN was in line with previous findings that found the performance of MCDO and MCBN to be very similar [23].

MCDO, MCBN and Ensemble all used the variance of multiple predictions to estimate uncertainty (either through STD or PDF). If the separate predictions did not agree with each other, the uncertainty will be higher. We indeed saw that the uncertainty became higher when the mean probability lowered, but again this did not give us the ability to reliably detect false positives. When the variance was higher, the mean of those predictions with high variance between them will generally be lower than when the variance is small.

Uncertainty versus mean squared error

Figure **b** of Figure 5.1 till 5.16 show us the uncertainty estimates plotted against the mean squared error (MSE) of all predictions (or just the squared error of the one prediction of Error Output). If the uncertainty estimation was perfectly able to increase uncertainty when the classification error was increasing there

would be a linear relation between the two axes that would increase from the left bottom to the right upper corner. We can see that none of the methods shows a perfect linear relationship, but most roughly increase the uncertainty (most of the time very fast) when the MSE increases. MCDO, MCBN and Ensemble also lowered the uncertainty again when the MSE was the highest. This happened when the probabilities for a certain class were high, but this class happened to be the wrong one (for example due to over-fitting while training).

Error output did not show this behaviour but seems to always increase the error when the squared error (not the mean squared error since it was now only one prediction) was high (this only happened when the Error was converted to an uncertainty estimate via method 2, see Figure 5.14b and 5.16b). Error Output seemed to nicely model its own uncertainty since it raised the uncertainty when it actually was completely right (MSE of 0). It did this because it "knows" it is not a perfect classifier and therefore a prediction with a very high probability (e.g. > 0.95) is unusual since it's actual accuracy on the test set might only be for example 80%.

When we compare the uncertainty plotted against the MSE to the probability plotted against the MSE (see Appendix A: Figure A.1 and Figure A.2), we see that only using the probability shows a better linear relation between the two axes. This further diminishes the advantage of the uncertainty estimate.

Uncertainty on untrained data

Figure **c** of Figure 5.1 till 5.16 show us the uncertainty estimates on the MNIST test, data that none of the models is trained on. We would like to see a high uncertainty for every example since none of them can be classified as something useful. We can see only Ensemble was able to give a high uncertainty for every example. MCDO and MCBN gave a high uncertainty, but the uncertainty drops down the closer the estimated highest probability was to 1. It seems that the extra variance that either Dropout or BN layers added to every different version of the network was not comparable to that of the different architectures of Ensemble. MCBN showed strangely low uncertainty over nearly all probabilities in Figure 5.7c, when the uncertainty was calculated via STD and the original model, was trained on CIFAR10. This was, however, solved when the uncertainty was calculated via PDF. This only occurred when trained on CIFAR10 and not when trained on Messidor-2, which probably had to do with the difference in the input size and the variance between the examples in the training sets.

STD vs PDF

Figure 5.1, 5.3, 5.5, 5.7, 5.9 and 5.11 show the plots when the standard deviation (STD) of all predictions was used as the uncertainty estimate, while Figure 5.2, 5.4, 5.6, 5.8, 5.10 and 5.12 show the plots when the probability density function (PDF) was used to calculate the uncertainty estimate. Both methods are explained in Section 3.2.6.

The PDF method caused the uncertainties to, on average, get higher and more squeezed together for all predictions that did not have a high probability estimate. Comparing Figure 5.7c with Figure 5.8c shows most clearly the advantage of the PDF method. Where STD fails to show convincing uncertainty for MNIST, PDF shows it perfectly. This effect was less pronounced for the other methods (or even datasets).

Since the PDF method requires quite some more computational steps compared to STD, the inference time is also slowed down.

Error Output: raw error versus conversion

Figure 5.13 and 5.15 show Error Output when using the outputted error as the uncertainty (raw), while Figure 5.14 and 5.16 show Error Output when the outputted error is first converted before being used as the uncertainty estimate (converted). This conversion is explained on Section 3.2.7.

When the error was used raw, only the information of the error outputs was used to determine the uncertainty. The conversion checked if the probabilities outputted by the classifier layer were in accordance with the output of the error layer, so both the information of the error outputs and the classifier outputs were used to determine the uncertainty.

When the output was used raw, the uncertainty never hits 0 and was often quite high, even when the probability estimate was (close to) 1, except for Figure 5.13c where the uncertainty was only close to 1 when the probability was near 0. Since Figure 5.13c represents the results when testing on an out-of-distribution dataset we would like to see a high uncertainty for all cases, not just for the cases with a low probability (the low probability itself would likely already cause the prediction to be rejected). Converting the raw error solved the first problem of the uncertainty never being close to 0 by linking the class probability and error together. This caused that an estimate class probability of for example 0.6 could still have a low uncertainty if the error agreed with this estimation. This can happen when the error output outputs an estimated error of around 0.4 for the example above. This means that if the estimated probability of 0.6 truly would point to the right class, the true error would have been 0.4. When this is more or less the same error as outputted by the error output, the error layer and classifier layer are agreeing with each other, resulting in a low uncertainty. We remain, however, with the same problem in Figure 5.14c as in Figure 5.13c.

It seems that the raw output was more able to capture the *aleatoric* uncertainty while the conversion method seems better at capturing the *epistemic* uncertainty. The capturing of epistemic uncertainty can best be seen in the 'V' shapes, where the lowest uncertainty occurred a bit below the probability that would reflect the achieved accuracy (see Table 5.1 and 5.2). This is the probability that most examples during training received and of which therefore the error was learned the best. This pattern can be useful to determine the overall range of error the uncertainty estimation of the model is exhibiting. When for example the lowest part of the "V" shape is at 0.7 probability, we can say that the uncertainty estimation of the model has an error of 0.3 or 30%. This number seems only to be around ± 0.1 or $\pm 10\%$ off of the actual accuracy, but can now be estimated without the use of the ground truth labels.

Figures (b) of Figures 5.13 till 5.14 show how the conversion method better linearly scaled with the squared error compared to the raw method.

Further observations

The Messidor-2 dataset was a difficult dataset and Ensemble, while obtaining the highest accuracy of all methods, only achieved an accuracy of 67%. Ensemble was the only method to reflect this low accuracy score by not giving high certainty to an single example in the Messidor-2 test set. On CIFAR10, where it received a higher accuracy, Ensemble was more confident in its predictions and therefore gave lower uncertainty estimations as well.

Chapter 6

Conclusion

6.1 Summary of Results

We began this thesis by asking the question which of the four methods provides the best uncertainty estimation in deep neural networks for image classification. While, in theory, a perfect Bayesian network can provide us with a tremendous amount of useful information, we found mixed evidence that MCDO and MCBN (the methods to approximate a Bayesian network) are capable of doing the same. None of the four methods was able to help distinguishing false positives from true positives and MCDO and MCBN were only partly successful at giving a high uncertainty to examples of an untrained class.

We found that the uncertainty estimation of the Ensemble method performed better than MCDO and MCBN. Ensemble was very good at giving a high uncertainty to all examples of untrained classes and better reflected overall model uncertainty on the Messidor-2 test set.

Error Output gave, without the conversion from error to uncertainty, not useful extra information. The conversion helps to capture the epistemic uncertainty and the resulting pattern, when plotted against the probability, helps to analyse this epistemic uncertainty.

MCDO and MCBN obtained the fastest training times of the four methods, but also very slow inference times (certainly on a large test set). Ensemble had the longest training time, but also received the best accuracy on the difficult Messidor-2 dataset. Error Output obtained a relatively fast training time and magnitudes faster inference time compared to the other methods. MCDOs, MCBNs and Error Outputs trained model were about equal in size, but the Ensemble was at least 40 times the size.

Overall, Ensemble gave the most reliable and accurate uncertainty estimates, but at the cost of slow training, mediocre inference time and large memory requirements. MCBN's inconsistent results make it no reliable option for most applications. MCDO seems promising, but the uncertainty estimate might not justify the slow inference time. Error Output introduces an interesting perceptive on obtaining uncertainty estimates, but again seems to not improve over the information that can already be obtained from the probability estimate.

All the above statements have to be put in perspective, however. MCDO, MCBN and Ensemble all use the mean of multiple predictions to give their final prediction. This inherently improves their accuracy as well [72] and already merges the uncertainty into the probability. Therefore, when we compare these methods to a single prediction of the complete, trained network, we can draw a different conclusion. If the variance between the multitude of predictions is high, a single prediction could just as well receive a (far) lower or higher probability then the mean now shows, all without an uncertainty estimation informing us about this problem.

Considering this, I can conclude that MCDO, MCBN and Ensemble, when increased inference time and memory requirements allow it, provide useful methods to lower the uncertainty of predictions. The addition of an uncertainty estimate proves useful to detect untrained classes and the mean prediction of MCDO, MCBN and Ensemble improve the quality of their predictions in general.

6.2 Recommendations for Future Research

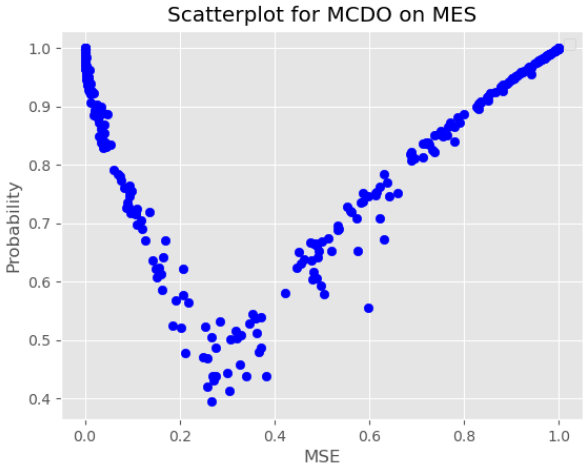
Future research could combine one or more of the above methods to try to obtain even more predictions for the calculation of the uncertainty estimate. All methods should be able to work together since none of them uses particular functions that would be incompatible with the other methods. The strengths of a particular method could be used to reduce the weakness of another method. For example, the fast training speed of MCDO could be used to train fewer members in an Ensemble, so combining the added variance of the Ensemble with also having enough separate predictions to obtain a reasonable uncertainty estimate through MCDO.

Another research topic could be expanding the Error Output method to include many more outputs that all try to capture a separate representation of the classes. This brings it closer to the ensemble method. A variation could also just include a single error output node. This single output only tries to detect hard-to-predict examples regardless of the class.

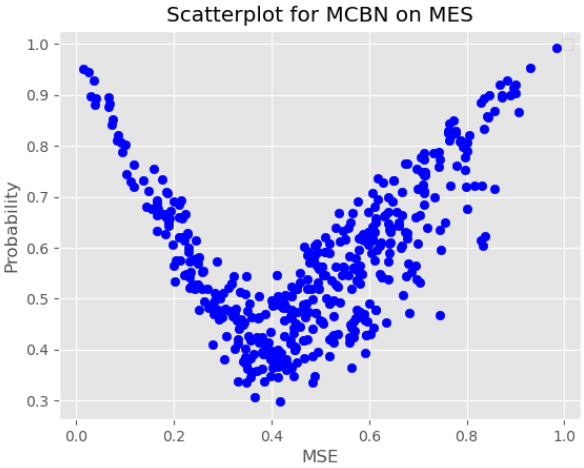
Finally, it has been shown that some of the methods are useful for detecting classes that the classifier is not trained on. This could be useful for (unsupervised) active learning where the learner can add examples with high uncertainty as new classes.

Appendix A

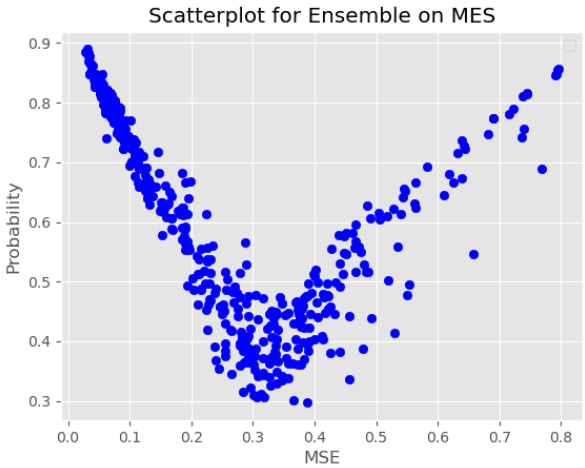
Mean Squared Error plotted against the Probability



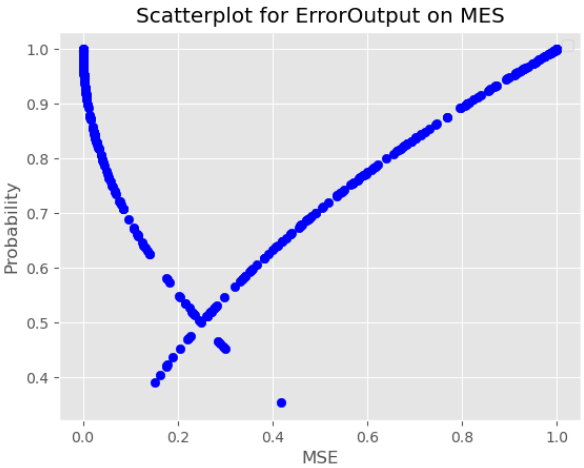
(a) MCDO



(b) MCBN

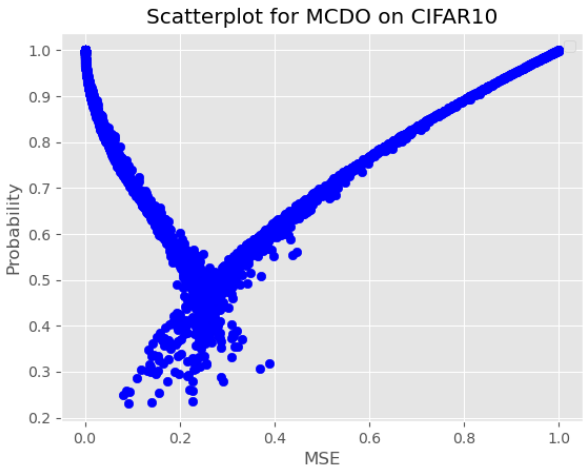


(c) Ensemble

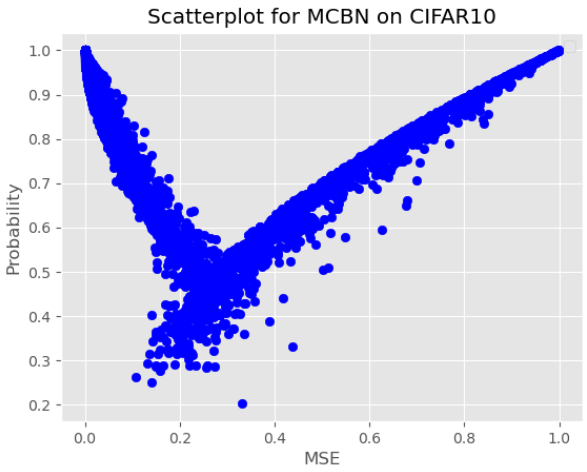


(d) Error Output

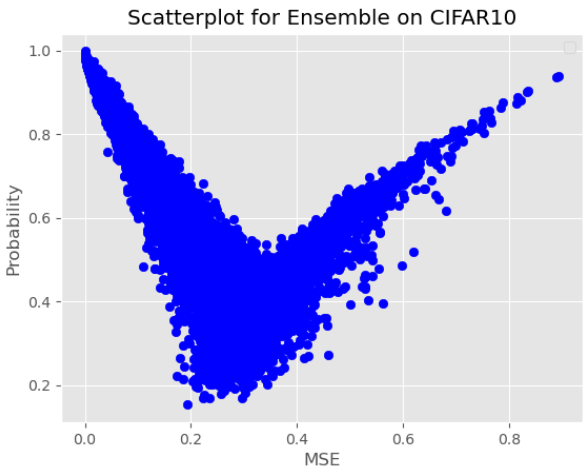
Figure A.1: Effect of mean squared error on probability on Messidor-2 test set



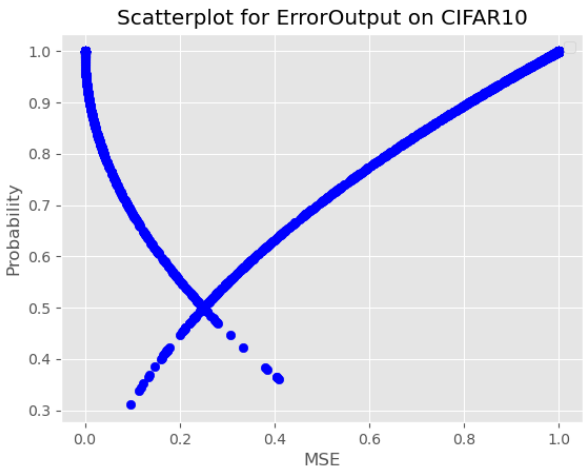
(a) MCDO



(b) MCBN



(c) Ensemble



(d) Error Output

Figure A.2: Effect of mean squared error on probability on CIFAR10 test set

Bibliography

[1] N. H. T. S. Administration, "Tesla Crash Preliminary Evaluation Report," tech. rep., NHTSA, Jan 2017.

[2] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[3] A. Blake, R. Curwen, and A. Zisserman, "A framework for spatiotemporal control in the tracking of visual contours," *International Journal of Computer Vision*, vol. 11, no. 2, pp. 127–145, 1993.

[4] X. He, R. S. Zemel, and M. Á. Carreira-Perpiñán, "Multiscale conditional random fields for image labeling," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, pp. II–II, IEEE, 2004.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[6] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5574–5584, Curran Associates, Inc., 2017.

[7] Z. Ghahramani, "Probabilistic machine learning and artificial intelligence," *Nature*, vol. 521, no. 7553, pp. 452–459, 2015.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[9] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8599–8603, IEEE, 2013.

[10] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[11] Y. Gal, *Uncertainty in deep learning*. PhD thesis, PhD thesis, University of Cambridge, 2016.

[12] S. Herzog and D. Ostwald, "Sometimes bayesian statistics are better," *Nature*, vol. 494, no. 7435, pp. 35–35, 2013.

[13] R. Van De Schoot, S. D. Winter, O. Ryan, M. Zondervan-Zwijnenburg, and S. Depaoli, "A systematic review of bayesian articles in psychology: The last 25 years.," *Psychological Methods*, vol. 22, no. 2, p. 217, 2017.

[14] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[15] B. Settles, "From theories to queries: Active learning in practice," in *Active Learning and Experimental Design workshop In conjunction with AISTATS 2010*, pp. 1–18, 2011.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[17] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," *arXiv preprint arXiv:2003.13350*, 2020.

[18] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*, pp. 63–71, Springer, 2003.

[19] R. M. Neal, *Bayesian learning for neural networks*, vol. 118. Springer Science & Business Media, 2012.

[20] H. Liu, Y.-S. Ong, X. Shen, and J. Cai, "When gaussian process meets big data: A review of scalable gps," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[21] D. J. MacKay, "A practical bayesian framework for backpropagation networks," *Neural computation*, vol. 4, no. 3, pp. 448–472, 1992.

[22] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural networks," *arXiv preprint arXiv:1505.05424*, 2015.

[23] M. Teye, H. Azizpour, and K. Smith, "Bayesian uncertainty estimation for batch normalized deep networks," *arXiv preprint arXiv:1802.06455*, 2018.

[24] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[25] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, 2009.

[26] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[27] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[28] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2625–2634, 2015.

[29] X. Sun, P. Wu, and S. C. Hoi, "Face detection using deep learning: An improved faster rcnn approach," *Neurocomputing*, vol. 299, pp. 42–50, 2018.

[30] V. N. Murthy, S. Maji, and R. Manmatha, "Automatic image annotation using deep learning representations," in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, pp. 603–606, 2015.

[31] B. Farley and W. Clark, "Simulation of self-organizing systems by digital computer," *Transactions of the IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 76–84, 1954.

[32] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[33] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[34] M. Minsky and S. A. Papert, *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[35] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[36] E. B. Baum and F. Wilczek, "Supervised learning of probability distributions by neural networks," in *Neural information processing systems*, pp. 52–61, 1988.

[37] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[38] I. J. Goodfellow, O. Vinyals, and A. M. Saxe, "Qualitatively characterizing neural network optimization problems," *arXiv preprint arXiv:1412.6544*, 2014.

[39] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[41] A. Cauchy, "M 'e general method é n 'e rale for the solution of é systems of simultaneous equations 'e es," *Comp. Give back. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.

[42] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[44] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent,"

[45] N.-N. Ji, J.-S. Zhang, and C.-X. Zhang, "A sparse-response deep belief network based on rate distortion theory," *Pattern Recognition*, vol. 47, no. 9, pp. 3179–3191, 2014.

[46] Y.-T. Zhou, R. Chellappa, A. Vaid, and B. K. Jenkins, "Image restoration using a neural network," *IEEE transactions on acoustics, speech, and signal processing*, vol. 36, no. 7, pp. 1141–1151, 1988.

[47] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in neural information processing systems*, pp. 3856–3866, 2017.

[48] J. S. Denker and Y. Lecun, "Transforming neural-net output levels to probability distributions," in *Advances in neural information processing systems*, pp. 853–859, 1991.

[49] D. J. MacKay, "A practical bayesian framework for backpropagation networks," *Neural computation*, vol. 4, no. 3, pp. 448–472, 1992.

[50] R. M. Neal, *Bayesian learning for neural networks*, vol. 118. Springer Science & Business Media, 2012.

[51] T. S. Jaakkola and M. I. Jordan, "Variational probabilistic inference and the qmr-dt network," *Journal of artificial intelligence research*, vol. 10, pp. 291–322, 1999.

[52] G. E. Box and G. C. Tiao, *Bayesian inference in statistical analysis*, vol. 40. John Wiley & Sons, 2011.

[53] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," 2012.

[54] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[55] R. Turner, P. Berkes, and M. Sahani, "Two problems with variational expectation maximisation for time-series models," *Bayesian Time Series Models*, 01 2008.

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

[57] Y. Gal, J. Hron, and A. Kendall, "Concrete dropout," in *Advances in Neural Information Processing Systems*, pp. 3581–3590, 2017.

[58] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[59] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2682–2690, 2019.

[60] D. P. Kingma, T. Salimans, and M. Welling, "Variational dropout and the local reparameterization trick," in *Advances in Neural Information Processing Systems*, pp. 2575–2583, 2015.

[61] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?," in *Advances in Neural Information Processing Systems*, pp. 2483–2493, 2018.

[62] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[63] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," in *Advances in Neural Information Processing Systems*, pp. 6402–6413, 2017.

[64] W. L. Buntine and A. S. Weigend, "Bayesian back-propagation," *Complex systems*, vol. 5, no. 6, pp. 603–643, 1991.

[65] D. A. Nix and A. S. Weigend, "Estimating the mean and variance of the target probability distribution," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 1, pp. 55–60, IEEE, 1994.

[66] E. Decencière, X. Zhang, G. Cazuguel, B. Lay, B. Cochener, C. Trone, P. Gain, R. Ordonez, P. Massin, A. Erginay, B. Charton, and J.-C. Klein, "Feedback on a publicly distributed database: the messidor database," *Image Analysis & Stereology*, vol. 33, pp. 231–234, Aug. 2014.

[67] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[68] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," 2010.

[69] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[70] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[71] C. Garbin, X. Zhu, and O. Marques, "Dropout vs. batch normalization: an empirical study of their impact to deep learning," *Multimedia Tools and Applications*, pp. 1–39, 2020.

[72] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.