An Exploratory Analysis of Multi-Class Uncertainty Approximation in Bayesian Convolutional Neural Networks

Sean Meling Murray

Department of Mathematics
University of Bergen

A thesis submitted in partial fulfillment of the requirements for the degree
Master of Science in Statistics (Data Analysis)

October 2018

Acknowledgements

I would like to thank my supervisors Professor Hans J. Skaug, Dr. Erik Hanson and Dr. Alexander Selvikvåg Lundervold for their generous support and guidance. A special thanks to Dr. Lundervold for introducing me to the topic and to Dr. Hanson for your enthusiastic interest and support. I would also like to thank you both for taking the time to discuss problems of both an academic and a personal nature. A special thank you to Professor Skaug for your interest, curiosity and steady guidance. A special thank you to Jens Christian Wahl and Tommy Odland for reading drafts and providing valuable feedback. Thank you to Jens Christian Wahl, Taran Fjell Naterstad and Yafee Ishraq for your company during the many (gruelling) hours of problem solving over the past years.

Finally, I would not have been able to pursue a master's degree in mathematics without the love, support and encouragement from my family. A very, very special thank you to Ane and my boy Johan.

Abstract

Neural networks are an important and powerful family of models, but they have lacked practical ways of estimating predictive uncertainty. Recently, researchers from the Bayesian machine learning community developed a technique called Monte Carlo (MC) dropout which provides a theoretically grounded approach to estimating predictive uncertainty in dropout neural networks [Gal and Ghahramani, 2016]. Some researchers have developed ad hoc approximations of these uncertainty estimates for use in convolutional neural networks [Feinman et al., 2017; Leibig et al., 2017]. We extend their research to a multi-class setting, and find that ad hoc approximations of predictive uncertainty in some cases provides useful information about a model's confidence in its predictions. Furthermore, we develop a novel approximation of uncertainty that in some respects performs better than those currently being used. Finally, we test these approximations in practice and compare them to other methods suggested in the literature. In our setting we find that the ad hoc approximations perform adequately, but not as well as those already suggested by experts.

Table of contents

List of figures

List of tables

Nomenclature

Roman Symbols

$\boldsymbol{A}$	Matrix
$\boldsymbol{a}$	Vector
a	Scalar
$\boldsymbol{X}$	Matrix of N inputs
$\boldsymbol{x}$	Input data point
$\boldsymbol{Y}$	Matrix of N outputs (targets/responses)
$\boldsymbol{y}$	Output data point (target/response)
$\hat{\boldsymbol{Y}}$	Matrix of N predictions
$\hat{\boldsymbol{y}}$	Prediction
$\boldsymbol{W}$	Weight matrix
$\boldsymbol{w}$	Weight vector

Greek Symbols

α	Momentum parameter
η	Learning rate
λ	Regularisation strength
$\sigma(\cdot)$	Non-linear function
$\boldsymbol{\theta}$	General notation denoting a parameter vector

$\hat{\boldsymbol{\theta}}$	General notation denoting an estimated parameter vector

Superscripts

0	Used to denote statistic associated with incorrectly classified data (e.g. $\hat{\sigma}^0_{pred}$)
1	Used to denote statistic associated with correctly classified data (e.g. $\hat{\sigma}^1_{pred}$)
$*$	Denotes previously unseen data (e.g. $\boldsymbol{x}^*$)

Subscripts

i	Row/column of matrix or specific element of vector (e.g. $\boldsymbol{x}_i$ or y_i)
ij	Specific element of matrix (e.g. w_{ij})

Other Symbols

$\mathbb{R}$	The real numbers
$\mathbb{E}$	Expected value
Var	Variance
$\mathcal{N}$	Gaussian distribution
$\mathcal{T}$	Training set of N input-output pairs
$\mathcal{L}(\boldsymbol{\theta})$	Loss function
$\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$	Alternative formulation of loss function
$R(\theta)$	Regularisation term, where θ denotes all model parameters
$C(\boldsymbol{\theta})$	Cost function (regularised loss)

Acronyms / Abbreviations

BCNN	Bayesian convolutional neural network
BNN	Bayesian neural network
CNN	Convolutional neural network
GP	Gaussian process
KL	Kullback-Leibler

MC	Monte Carlo
MI	Mutual information
ML	Maximum likelihood
MLE	Maximum likelihood estimate
MLP	Multi-layer perceptron, also known as a feedforward neural network
PE	Predictive entropy
VR	Variation ratio
i.i.d.	independent and identically distributed

Chapter 1

Introduction

Interest in machine learning has exploded in recent years thanks to an ever-growing treasure trove of available data and advances in computing hardware. This has led to a resurgence of interest in a family of models known as neural networks, which in turn has spawned a fast-paced field of research called deep learning. In the last few years, specialised networks have made significant breakthroughs in areas as diverse as computer vision, natural language processing and reinforcement learning. As a result of their tremendous success, neural networks are quickly becoming ubiquitous in problems where there are large amounts of data.

Although they are very powerful, neural networks can sometimes make inexplicably bad predictions with high perceived confidence. This is a problem in applications where human safety is a concern. In these settings, being able to express uncertainty about the predictions is important. Uncertainty estimation has typically been associated with a branch of statistics known as Bayesian modelling, but until recently there haven't been any practical ways of obtaining Bayesian uncertainty estimates from neural networks.

This changed when recent work established a connection between dropout training in a neural network and approximate inference in a Bayesian model known as the Gaussian process [Gal and Ghahramani, 2016]. Interpreting dropout training as approximate Bayesian inference allows us to obtain theoretically grounded estimates of predictive uncertainty in dropout neural networks. In practice, this is done by leaving dropout on at test time. An input is passed through the network many times, and each time a slightly different network is making predictions because dropout randomly switches off parameters. In a regression setting this allows us to obtain uncertainty estimates by calculating the empirical variance of the different outputs associated with a single input. This method has been dubbed Monte Carlo (MC) dropout for reasons that will become clear in chapter 4. Furthermore, MC dropout is extended beyond dropout networks to more complex architectures such as convolutional neural networks. In this setting the Gaussian process approximation interpretation is lost.

However, researchers have formulated ad hoc approximations of the predictive uncertainty obtained in a regression setting and applied them to binary image classification tasks [Feinman et al., 2017; Leibig et al., 2017] using so-called Bayesian convolutional neural networks. These methods rely on computing the empirical variance of the predicted probabilities of belonging to a certain class, and seem to work well in practice. In applications, the idea is that uncertainty information can be used to flag predictions in which the model has low confidence, allowing the associated images to be examined further downstream. This is known as uncertainty-informed referral.

In this thesis we extend these ad hoc uncertainty approximations to a multi-class image classification setting. Our goal is to explore the empirical properties of these ad hoc methods. Furthermore, we introduce a novel approximation of uncertainty and show that it compares favourably to existing methods in some important respects. We also compare these ad hoc methods to other uncertainty quantifications suggested in the literature which are better suited to classification tasks. The main contributions of this thesis are:

- As far as we are aware, this is the first work that extends existing ad hoc uncertainty approximations to a multi-class classification setting. To the best of our knowledge, we are also the first to compare the empirical properties of these ad hoc approximations to other uncertainty quantifications which are rooted in information theory.

- As far as we are aware, we are the first to introduce the novel uncertainty approximation $\hat{\sigma}_{\text{model}}$ given in section 5.5. In brief, we estimate the mean of the empirical standard deviations of the different class probabilities associated with an input. We observe that our measure of uncertainty is sensitive to inputs where predicted class probabilities are low. Furthermore, compared to other ad hoc methods, our quantification of uncertainty performs better in an uncertainty-informed referral experiment.

- Finally, an important motivation for this thesis was to provide an overview of machine learning, neural networks, Bayesian statistics and recent research into practical applications of MC dropout in image classification tasks. This is useful for other students interested in the intersection of Bayesian modelling and convolutional neural networks. To the best of our knowledge, there does not exist an introductory level text describing practical uncertainty estimation in Bayesian deep learning at the time of writing.

This thesis will be split into two parts. Part 1 is an overview of the theory underlying MC dropout. In chapter 2 we introduce the theoretical framework of learning from data. In chapter 3 we introduce the general theory of neural networks and convolutional neural networks. Chapter 4 gives an overview of the Bayesian approach to modelling, and we briefly review some of the methods used to develop MC dropout. We also sketch the main results of [Gal and Ghahramani, 2016] and its extension to convolutional neural networks [Gal and Ghahramani, 2015]. Chapter 4 concludes with a brief review of recent research into practical applications of MC dropout and an introduction to the ad hoc methods we are interested in.

In Part 2 we explore the empirical properties of these ad hoc methods, with particular focus on the work presented in [Leibig et al., 2017]. We briefly compare this to a different ad hoc method presented in [Feinman et al., 2017], which can be interpreted as a measure of multi-class uncertainty. We go on to introduce a novel approximation of uncertainty and briefly review how it compares to our extensions of existing methods. We conclude chapter 5 with a simple experiment where all three ad hoc uncertainty approximations are put to the test and we compare them to other uncertainty quantifications proposed by the developers of MC dropout. In chapter 6 we provide details on the data set, the implementation and training of our models and our implementation of MC dropout.

Part 2 is followed by a summary of our results. We also point to interesting areas of future research. All figures in this thesis have been made using Sketch[1], draw.io[2], the Python[3] plotting library Matplotlib and the ggplot2 package for R[4]. The LaTeX template we use is available at `https://github.com/kks32/phd-thesis-template`.

[1]Sketch: `https://www.sketchapp.com/`
[2]Draw.io: `https://www.draw.io/`
[3]Python: `https://www.python.org/`
[4]R: `https://www.r-project.org/`

Part I

Background

Chapter 2

Learning From Data

Machine learning is a sub-field of mathematics, statistics and computer science concerned with designing algorithms that can learn from data. It is closely related to the field of statistical learning, and conventional wisdom will have it that these two fields represent different approaches to modelling data. The distinction, often taught in practically oriented courses and textbooks, is that machine learning is freed from modelling constraints and assumptions; the ultimate goal is predictive accuracy. Statistical learning, on the other hand, concerns itself with model validity, accurate estimations and inference. Leo Breiman argues that as data evolves and becomes more complex, so to must the statistician [Breiman et al., 2001]. By embracing both paradigms, the statistician gains access to a more diverse set of tools. As Breiman himself writes: "Framing the question as a choice between interpretability and accuracy is an incorrect interpretation of what the goal of statistical analysis is. The point of a model is to get useful information about the relation between the response and predictor variables."

This will be the guiding principle of this chapter, and consequently we will use terms and ideas from both fields interchangeably as we give a brief introduction to the theoretical framework of learning from data. We primarily rely on the following sources (and the references therein):

- Chapters 3, 5 and 7 of [Goodfellow et al., 2016] (machine learning basics, regularisation and information theory).
- Chapters 2, 7, and 8 of [Hastie et al., 2001] (supervised learning, model assessment and model averaging).

- Chapters 2, 5 and 8 of [James et al., 2014] (bias-variance trade-off, cross-validation and model averaging).
- Chapter 1 and 2 of [Mitchell, 1997] (basic machine learning terminology).
- Chapter 7 of [Casella and Berger, 2002] (point estimation).
- Chapter 1 of [Bishop, 2006] (information theory).
- Chapter 2 of [Pawitan, 2013] (likelihood estimation).

We follow [Goodfellow et al., 2016] closely in notation. Generally, a bold-faced $\boldsymbol{x}$ denotes a (column) vector, $\boldsymbol{X}$ denotes a matrix with rows $\boldsymbol{x}^\intercal$, and x denotes a scalar. In conventional statistical notation uppercase letters denote random variables and lowercase letters denote realisations of random variables. We will not follow this notational convention.

2.1 Basic terminology

A useful and commonly cited definition of machine learning is provided in [Mitchell, 1997]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Mitchell states that a well-posed machine learning problem needs to identify the class of tasks, the measure of performance and the source of experience. For example, we may be interested in recognising and classifying cats and dogs (T) based on a collection of images of cats (E). A relevant performance measure could be the percentage of correctly classified images (P).

The experience E is a data set, which is a collection of inputs, assumed to be independent and identically distributed (i.i.d.). The inputs are typically organised in a matrix $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, where the i'th row corresponds to a D-dimensional vector of features $\boldsymbol{x}_i^\intercal = (x_{i1}, ..., x_{iD})$. Each element corresponds to a feature of the input. In statistical literature features are referred to as covariates. For example, if we are predicting the height of an adolescent, we may use a feature vector $\boldsymbol{x}_i^\intercal = (x_{i1}, x_{i2})$ where x_{i1} is the height of the father and x_{i2} is the height of the mother.

Sometimes we know the desired output of the model. The data set $\boldsymbol{X}$ is then paired with a vector $\boldsymbol{y} \in \mathbb{R}^N$ (or a matrix $\boldsymbol{Y} \in \mathbb{R}^{N \times K}$) containing the associated target values (response variables) for each input. This is called a training set, and the target values and observations are assumed to share a joint distribution $p(\boldsymbol{X}, \boldsymbol{y})$. Using known target values to make a model perform better is called supervised learning. In unsupervised learning we are typically interested in detecting patterns or meaningful structure in the data. A third option is semi-supervised learning, where the model is built using both labelled and unlabelled data. In this thesis we focus on supervised learning.

There are many different tasks T to be learned, but in most cases we want to model some unknown function or distribution. Regression and classification are among the most common examples. In regression, the task is to model an unknown function $f : \mathbb{R}^D \to \mathbb{R}$. Predicting height based on a set of features is a regression problem. Classification tasks are concerned with assigning observations to one of K different classes through some function $f : \mathbb{R}^D \to \{1, ..., K\}$, like predicting a person's gender or recognising one of K objects in an image. In some cases we want f to output the probability of belonging to a class rather than the class value itself. If this is the case, the model is denoted $p : \mathbb{R}^D \to [0, 1]$.

Example 2.1. Linear regression. Suppose that we assume a linear relationship between a target variable $y_i \in \mathbb{R}$ and some input $\boldsymbol{x}_i \in \mathbb{R}^D$:

$$y_i = f(\boldsymbol{x}_i) + \epsilon = \boldsymbol{x}_i^\mathsf{T}\boldsymbol{\theta} + \theta_0 + \epsilon_i. \tag{2.1}$$

We want to find the linear function f that captures the systematic relationship between $\boldsymbol{x}_i$ and y_i. The error term ϵ_i is assumed to be $\mathcal{N}(0, \sigma_\epsilon^2)$ and independent of $\boldsymbol{x}_i$. It captures all deviations from the deterministic relationship between y_i and $f(\boldsymbol{x}_i)$. The vector $\boldsymbol{\theta} \in \mathbb{R}^D$ describes how each individual feature affects the response. θ_0 is called the bias or intercept, and indicates the value of y_i if $\boldsymbol{x}_i = \boldsymbol{0}$. Typically we concatenate a 1 to $\boldsymbol{x}_i$ and incorporate θ_0 into the parameter vector so that we can express eq. 2.1 more compactly. Since we don't know the true $\boldsymbol{\theta}$, we must estimate it. The process of estimation is called model fitting. The estimated parameters are denoted $\hat{\boldsymbol{\theta}}$, and the estimated function is denoted $\hat{f}$ (using the more compact notation):

$$\hat{y}_i = \hat{f}(\boldsymbol{x}_i) = \boldsymbol{x}_i^\mathsf{T}\hat{\boldsymbol{\theta}}.$$

⌟

Example 2.2. Logistic regression. Suppose we want to model the probability of a binary outcome $y_i \in \{0, 1\}$. One way to do this is to define a function that maps the output of $\boldsymbol{x}_i^\intercal \boldsymbol{\theta}$ to an interval $[0, 1]$. The logistic function accomplishes this:

$$p(y_i = 1 | \boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{1}{1 + \exp(-\boldsymbol{x}_i^\intercal \boldsymbol{\theta})}.$$

⌟

Approximating a function f is in other words the same as estimating the parameters $\boldsymbol{\theta}$. One view of machine learning is that we use the available data to search the parameter space for a setting of $\hat{\boldsymbol{\theta}}$ that gives us the best approximation of f. An equivalent view is that we are searching a space of functions for the best approximation.

2.2 Performance Measures

How well a model performs with respect to a given task T is usually determined by some quantitative performance measure P. The typical example of a performance measure for linear regression is the mean squared error (MSE). Given a data set where $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{y} \in \mathbb{R}^N$, we can define

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \boldsymbol{x}_i^\intercal \boldsymbol{\theta})^2.$$

Predictions that are far from the target value y_i cause the MSE to increase. By adjusting $\boldsymbol{\theta}$ we can decrease the MSE and get a model that is consistently closer to the target values. Generally, a function that quantifies prediction errors is referred to as a cost, objective or loss function, depending on your choice of statistical or machine learning literature. In this thesis we will use the term loss to denote the function measuring the prediction error, and we will reserve the term cost for regularised loss functions (section 2.4). The loss is a function of $\boldsymbol{\theta}$, $\boldsymbol{X}$ and $\boldsymbol{y}$, but for notational convenience we denote it by $\mathcal{L}(\boldsymbol{\theta})$. Equivalently, we can denote the loss as a function of the target values and the predictions $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$. We will use these two notational conventions interchangeably.

The model's overall usefulness is determined by its ability to generalise to new data. Let $\mathcal{T} = (\boldsymbol{X}, \boldsymbol{y})$ denote a fixed training set of N observations and let $(\boldsymbol{x}^*, \boldsymbol{y}^*)$ denote an independent observation drawn randomly from the joint distribution of the data. We want to minimise the prediction error of the model trained on the specific training set $\mathcal{T}$, given by

$$\mathbb{E}[\mathcal{L}(\boldsymbol{y}^*, \hat{\boldsymbol{y}}^*)|\mathcal{T}], \tag{2.2}$$

where $\hat{\boldsymbol{y}}^*$ denotes the prediction for $\boldsymbol{x}^*$. The expression in eq. 2.2 is also called the test error. It is clear that the test error depends on the sampling of training data, so what we really want is to minimise the expected test error over all possible samplings of $\mathcal{T}$,

$$\mathbb{E}_{\mathcal{T}}[\mathbb{E}[\mathcal{L}(\boldsymbol{y}^*, \hat{\boldsymbol{y}}^*)|\mathcal{T}]] = \mathbb{E}_{\mathcal{T}}[\mathcal{L}(\boldsymbol{y}^*, \hat{\boldsymbol{y}}^*)].$$

This is generally not possible because we only have the information provided by the training data at hand. Consequently, the expected test error must be estimated. The average training error

$$\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i)$$

could serve as a proxy for the expected test error, but this approach has a serious drawback: The training error can be made arbitrarily small if we choose a sufficiently flexible model. This is called overfitting, which will be the subject of the next section. The standard approach is to partition the data into a training set used to fit the model and an independent test set used to estimate the expected test error. Sometimes we also want a validation set to help us make modelling choices before we estimate the expected test error. This will be discussed in section 2.5.

2.2.1 Bias and Variance

In this section we will discuss how different types of errors can affect predictions. Before we move on, it may be helpful to review the basic properties of an estimator. In the frequentist view of statistics, a point estimate $\hat{\boldsymbol{\theta}}$ is the best possible prediction of some fixed but unknown quantity of interest $\boldsymbol{\theta}$, such as the parameters in example 2.1. Let $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ be a sample of i.i.d. data points. In general, a point estimator is defined as a function of the data:

$$\hat{\boldsymbol{\theta}} = g(\boldsymbol{x}_1, ..., \boldsymbol{x}_N).$$

Since $\hat{\boldsymbol{\theta}}$ is a function of a random sample of variables, $\hat{\boldsymbol{\theta}}$ itself is a random variable. In many applications we are interested in the bias and the variance of $\hat{\boldsymbol{\theta}}$. The bias is the distance from the expected value of $\hat{\boldsymbol{\theta}}$ and the true value of $\boldsymbol{\theta}$:

$$\text{Bias}(\hat{\boldsymbol{\theta}}) = \mathbb{E}(\hat{\boldsymbol{\theta}}) - \boldsymbol{\theta}.$$

An estimator is unbiased if $\text{Bias}(\hat{\boldsymbol{\theta}}) = \mathbf{0}$. The variance of an estimator is denoted $\text{Var}(\hat{\boldsymbol{\theta}})$ and describes how much we expect $\hat{\boldsymbol{\theta}}$ to vary when calculated over a different random sampling of the data.

In machine learning applications bias and variance correspond to the prediction errors introduced by overly simple or overly complicated models, respectively. The more complex a model is, the more likely it is to adapt to random noise in the training data instead of the true, underlying function f. As a result, complex models may perform very well on the training data but fail miserably on the test set, and we say that the model has overfitted. Overfit models are prone to high variance, meaning $\hat{f}$ changes substantially if fit on a different sampling of data. It is also possible to underfit a model. In this situation $\hat{f}$ fails to capture the complexity in the data, and performs poorly on both the training and the test set. Underfitting introduces bias. Fig. 2.1 shows an example of under- and overfitting.

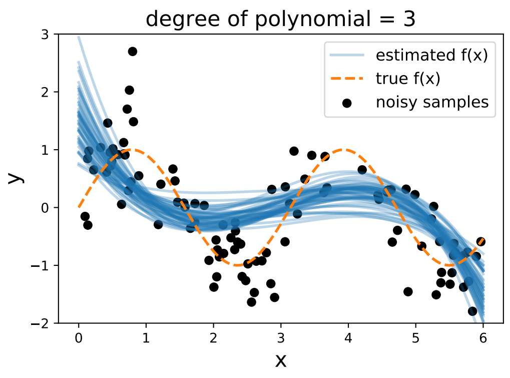
Bias and variance are collectively referred to as reducible errors. Increasing the complexity of $\hat{f}$ will tend to increase variance and decrease bias. Conversely, decreasing capacity introduces bias but reduces variance. One of the main challenges of building a model is controlling this so-called bias-variance trade-off. See fig. 2.1d for an example of this. Irreducible errors are introduced by inherent uncertainty in our measurements, and cannot be controlled. The bias-variance trade-off is related to the no free lunch theorem [Wolpert and Macready, 1997], which states that there does not exist one superior universal model for all tasks. The bias-variance trade-off is typically illustrated using the following example:

Example 2.3. Bias-variance decomposition. Assume that $y = f(\boldsymbol{x}) + \epsilon$, where $\mathbb{E}(\epsilon) = 0$ and $\text{Var}(\epsilon) = \sigma_\epsilon^2$. Let $\mathbb{E}[(y^* - \hat{f}(\boldsymbol{x}^*))^2]$ denote the expected test error, which is the mean squared error resulting from training a large number of models on different samplings of data and testing them all on an independent sample $(\boldsymbol{x}^*, y^*)$. By expanding the squared term and using the definitions of bias and variance, it can be shown[1] that the expected squared error at a new data point $\boldsymbol{x}^*$ can expressed as

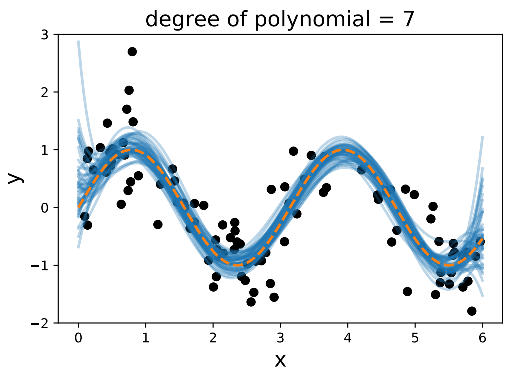
$$\begin{aligned}\mathbb{E}[(y^* - \hat{f}(\boldsymbol{x}^*))^2] &= \sigma_\epsilon^2 + \text{Var}[\hat{f}(\boldsymbol{x}^*)] + \text{Bias}^2[\hat{f}(\boldsymbol{x}^*)] \\ &= \text{Irreducible error} + \text{Var}[\hat{f}(\boldsymbol{x}^*)] + \text{Bias}^2[\hat{f}(\boldsymbol{x}^*)].\end{aligned} \tag{2.3}$$

Eq. 2.3 shows that the expected test error can be decomposed into a sum of the variance of the error term ϵ, the squared bias of $\hat{f}(\boldsymbol{x}^*)$ and the variance of $\hat{f}(\boldsymbol{x}^*)$. ⌟

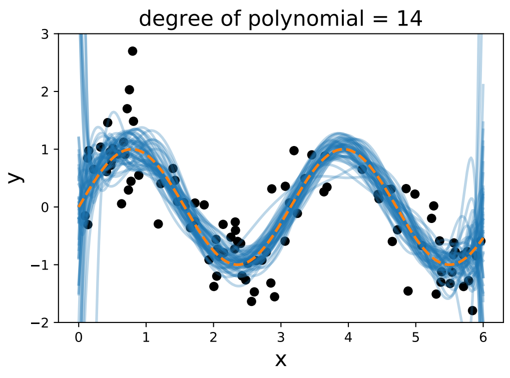
[1]See appendix A, section A.1.



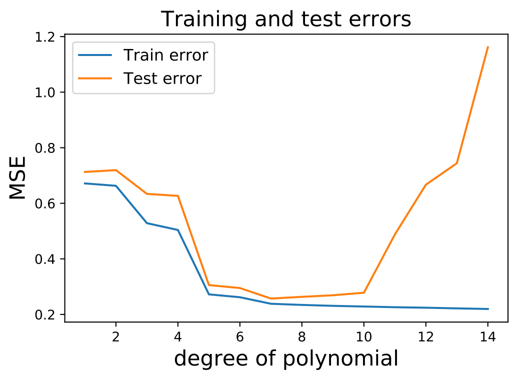
(a)



(b)



(c)



(d)

Fig. 2.1: The plots show three different polynomial regression models of degrees $d_a = 3$, $d_b = 7$ and $d_c = 14$ (see chapter 5 of [Hastie et al., 2001] for details). The models have been fit on 50 data sets, each containing 100 random samples generated from $y_i = \sin(2x_i) + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, 0.1^2)$. Fig. 2.1a shows an example of underfitting, i.e. $\hat{f}(x)$ fails to capture the complexity in the data. Fig. 2.1b shows an example of a good fit. Fig. 2.1c shows a model that is overfit, i.e. $\hat{f}(x)$ changes substantially with different samples of data. Overfitting can be diagnosed by plotting the training and test errors for the various polynomial degrees. The characteristic U-shape of the test terror (in orange) in fig. 2.1d indicates for which polynomial degrees the model fails to generalise. Note that the training error steadily decreases as the $\hat{f}(x)$ gets more complex.

Note that the decomposition in example 2.3 does not apply in a classification setting (see chapter 7 in [Hastie et al., 2001]). To control the bias-variance trade-off we typically rely on techniques such as cross-validation (see section 2.5.1).

2.3 Fitting a Model

To reiterate, the goal of model fitting is to find the parameters that best describe the observed data. In practice we minimise the average training error with respect to the parameters $\boldsymbol{\theta}$, which makes learning an optimisation problem:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\theta}).$$

For some performance measures, such as the MSE, the minimisation problem has a convenient closed form solution, as illustrated in the following example:

Example 2.4. Least squares estimation. Consider a simple linear regression model where $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{y} \in \mathbb{R}^N$. The model, parameterised by $\boldsymbol{\theta} \in \mathbb{R}^D$, is then given by

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\theta}.$$

Minimising the MSE is equivalent[2] to minimising the sum of squared errors (SSE), and the best estimates are given by

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^{\mathsf{T}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}).$$

Expanding the terms, taking the partial derivatives with respect to $\boldsymbol{\theta}$ and solving for zero gives us the least square estimates $\hat{\boldsymbol{\theta}} \in \mathbb{R}^D$:

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathsf{T}}\boldsymbol{y}.$$

⌟

2.3.1 Maximum Likelihood Estimation

The MSE is a natural choice when dealing with models that assume additive errors, such as in example 2.1 and 2.3. In classification tasks we typically model the conditional probability of belonging to a given class, denoted $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})$. Estimating $\boldsymbol{\theta}$ often amounts to finding the parameters that make $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})$ as close as possible to the true but unknown data-generating distribution $p(\boldsymbol{y}|\boldsymbol{X})$. One way to accomplish this is by

[2]The optimal parameter values aren't affected by scaling.

maximum likelihood estimation (MLE). We follow [Goodfellow et al., 2016] closely in this section.

The maximum likelihood (ML) estimator of $\boldsymbol{\theta}$ is given by:

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{\text{ML}} &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}).\end{aligned} \tag{2.4}$$

The function $L(\boldsymbol{\theta})$ is called the likelihood. It expresses a measure of relative preference for various parameter values. The ML estimates thus provide the parameter values that maximises the probability of observing the sampled data. Noting that the log-transformation of eq. 2.4 is maximised by the same value[3] of $\boldsymbol{\theta}$ yields

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{\text{ML}} &= \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} l(\boldsymbol{\theta}).\end{aligned}$$

The logarithm transforms the product into a sum, which in many cases simplifies mathematical analysis. The function $l(\boldsymbol{\theta})$ is called the log-likelihood. Maximising the log-likelihood is equivalent to minimising the negative log-likelihood, so that

$$\hat{\boldsymbol{\theta}}_{\text{ML}} = \arg\min_{\boldsymbol{\theta}} -\sum_{i=1}^{N} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}). \tag{2.5}$$

The estimator $\hat{\boldsymbol{\theta}}_{\text{ML}}$ has several desirable properties. Under certain conditions, $\hat{\boldsymbol{\theta}}_{\text{ML}}$ is a consistent estimator. This means that $\hat{\boldsymbol{\theta}}_{\text{ML}}$ converges to the true value of the parameter as the number of training observations approaches infinity. Consequently, MLE is a useful approach when lots of data is available.

We can obtain an optimisation objective from eq. 2.5 by observing that the ML estimates are invariant to scaling. This means that we can express $\hat{\boldsymbol{\theta}}_{\text{ML}}$ as an expectation over the empirical distribution of the observed data. Reframing the MLE

[3]The logarithm is a monotonically increasing function.

as a loss function results in the negative log-likelihood loss or cross-entropy loss (see section 2.6)

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) &= \arg\min_{\boldsymbol{\theta}} -\mathbb{E}\Big[\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})\Big] \\ &= \arg\min_{\boldsymbol{\theta}} -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}), \end{aligned} \tag{2.6}$$

which is very common in many machine learning applications. In the following example we show a simple case where $\hat{\boldsymbol{\theta}}_{\mathrm{ML}}$ can be derived analytically:

Example 2.5. Maximum likelihood estimation. Consider the same setup as in example 2.4. Let $y_i|\boldsymbol{x}_i, \boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{x}_i^\mathsf{T}\boldsymbol{\theta}, \sigma_\epsilon^2)$. The conditional distribution has the functional form

$$p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{1}{\sqrt{2\pi}\sigma_\epsilon} \exp\left(-\frac{1}{2\sigma_\epsilon^2}(y_i - \boldsymbol{x}_i^\mathsf{T}\boldsymbol{\theta})^2\right).$$

The MLE is given by

$$\begin{aligned} \hat{\boldsymbol{\theta}}_{\mathrm{ML}} &= \arg\min_{\boldsymbol{\theta}} -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta}} \left(\frac{1}{2}\log(2\pi\sigma_\epsilon^2) + \frac{1}{2N\sigma_\epsilon^2}\sum_{i=1}^{N}(y_i - \boldsymbol{x}_i^\mathsf{T}\boldsymbol{\theta})^2\right). \end{aligned}$$

It follows that maximum likelihood estimates $\hat{\boldsymbol{\theta}}_{\mathrm{ML}}$ can be obtained by minimising the SSE term, as in example 2.4, resulting in the same optimal parameters for this particular example. ⌟

2.3.2 Gradient Descent

In most machine learning problems a closed form solution does not exist, and we must resort to iterative methods. One of the most widely used is gradient descent. Recall that the gradient of $\mathcal{L}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta} \in \mathbb{R}^D$ is the vector of partial derivatives

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) = \left(\frac{\partial}{\partial\theta_1}\mathcal{L}(\boldsymbol{\theta}), ..., \frac{\partial}{\partial\theta_D}\mathcal{L}(\boldsymbol{\theta})\right)^\mathsf{T}.$$

The gradient points in the direction where the loss function increases the most. This means that the negative gradient $-\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ points in the direction where the loss decreases most rapidly. The idea behind gradient descent is to start with a set of

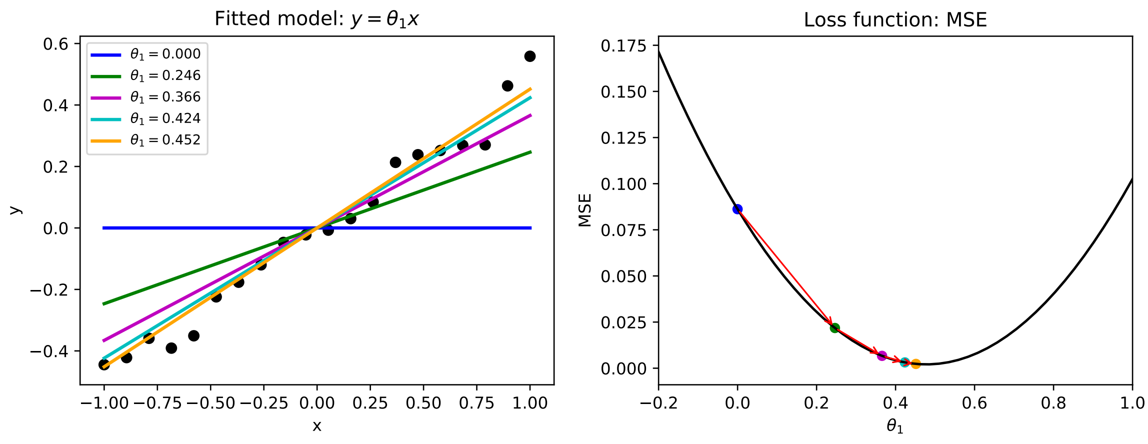


Fig. 2.2: The plot shows an example of gradient descent in linear regression with one parameter $\theta_1 \in \mathbb{R}$. The data is generated from the model $y_i = 0.5x_i + \epsilon_i$, where $i = 1, ..., 20$ and $\epsilon_i \sim \mathcal{N}(0, 0.05^2)$. The left hand plot shows the model at each iteration of gradient descent, starting with the blue line and ending at the yellow line. The plot to the right shows the corresponding value of the MSE at each iteration. The best line (in yellow) on the left corresponds to the lowest MSE on the right. Code for generating the plots can be found at `https://scipython.com/blog/visualizing-the-gradient-descent-method/`.

randomly initialised parameters $\boldsymbol{\theta}$ and iteratively adjust them by taking small steps in the direction where $\mathcal{L}(\boldsymbol{\theta})$ decreases the most:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1}).$$

This is repeated until convergence, which typically depends on some predetermined stopping criteria. The symbol η is called the learning rate. It determines the step size in the direction of the negative gradient and must be controlled carefully. If η is too large, gradient descent may overshoot the minimum and $\boldsymbol{\theta}$ could fail to converge. On the other hand, if η is too small, convergence may take too long. Parameters like η that control the behaviour of learning algorithms are called hyperparameters. There are several techniques for determining the best hyperparameters. One way is to use cross-validation, described in 2.5.1.

As stated in section 2.2, the average training error is typically a sum of losses associated with individual observations, and the gradient over the training data is then given by

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}). \tag{2.7}$$

Calculating the gradient update requires summing over the entire training set, which becomes computationally expensive when N is large. However, if we leverage the fact that the gradient in eq. 2.7 is in fact an expectation, then we can estimate $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ by randomly sampling a subset of m examples from the original training data:

$$\boldsymbol{g} = \frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta}).$$

Estimating the gradient based on a subset of training examples is called mini-batch stochastic gradient descent (SGD). The SGD update rule is given by the following algorithm:

Algorithm 1: Stochastic gradient descent

```
Require: Learning rate $\eta$
Require: Initial values for $\boldsymbol{\theta}$
while stopping criterion not met do
    Randomly sample mini-batch of $m$ input-output pairs $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$
    Compute gradient estimate: $\boldsymbol{g} = \frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta}_{t-1})$
    Update parameters: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta\boldsymbol{g}$
end
```

One nice property of SGD is that it may in fact converge before the entire training set has been processed, provided that the training set is large enough. Figure 2.2 shows an example of gradient descent applied for four iterations to a linear regression problem.

2.4 Reducing the Generalisation Error

The risk of overfitting dramatically increases as models get very complex, which can often happen in practical applications. In machine learning and statistics, regularisation commonly refers to strategies meant to prevent generalisation errors as a result of overfitting. In the following we will introduce two such strategies which will be relevant for later discussions: L_2-regularisation and bagging. The interested reader is referred to [Goodfellow et al., 2016] and [Hastie et al., 2001] for an excellent overview of other methods for reducing generalisation error.

2.4.1 Weight decay

One way to deal with an overly complex model is to constrain the parameter estimates, resulting in a less expressive model (see fig. 2.3). In practice this can done by adding a L_2 norm penalty term $\|\boldsymbol{\theta}\|^2$ to the loss function. This is often referred to as ridge regression (in statistics) or weight decay (in machine learning) and results in the cost function

$$C(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2 \tag{2.8}$$

where λ is a non-negative hyperparameter that controls the relative impact of the penalty term $\|\boldsymbol{\theta}\|^2$ on $C(\boldsymbol{\theta})$.

Example 2.6 (Ridge regression). Consider the same setup as in example 2.1, but this time with an added penalty term $\lambda \|\boldsymbol{\theta}\|^2$. The ridge estimate is given by

$$\hat{\boldsymbol{\theta}}_{\text{ridge}} = \arg\min_{\boldsymbol{\theta}} \Big((\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^\mathsf{T}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\theta} \Big).$$

Following the same steps as in example 2.1 results in

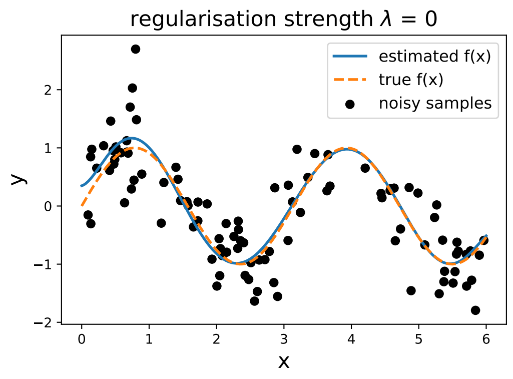
$$\hat{\boldsymbol{\theta}}_{\text{ridge}} = (\boldsymbol{X}^\mathsf{T}\boldsymbol{X} + \lambda \mathbf{I})^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y}.$$

⌟

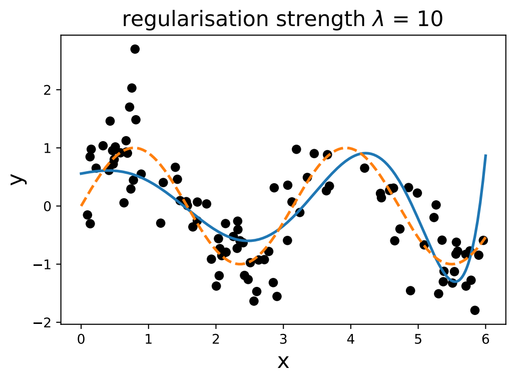
Larger values of λ shrink the parameters towards zero when minimising $C(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. Applying regularisation controls overfitting by limiting the model's ability to adapt to random noise in the data. The appropriate value of λ can be found using cross-validation (see section 2.5.1).

2.4.2 Bagging

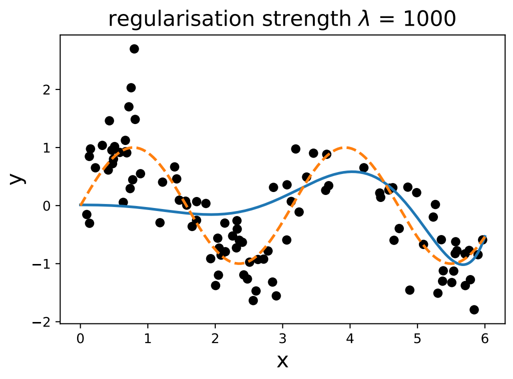
Another way to reduce the generalisation error is to average, or ensemble, the predictions of many different models. One such approach is called bagging (short for bootstrap aggregation). The idea is to construct B training sets, $\{(\boldsymbol{x}_1^b, y_1^b), \ldots, (\boldsymbol{x}_N^b, y_N^b)\}_{b=1}^B$ by sampling with replacement from our original training data $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$. Each training set is used to train a separate model $\hat{f}_b(\boldsymbol{x})$. Finally, all B models are



(a)



(b)



(c)

Fig. 2.3: As the regularisation strength λ increases, the penalty on the weights get more severe, causing them to shrink. Regularisation is useful when we want to limit the model's ability to adapt to random noise in the data.

averaged to get the bagged estimate, given by

$$\hat{f}_{bag}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(\boldsymbol{x}).$$

The variance of the bagged estimate will at worst be as bad as any of the individual models in the ensemble, but potentially much lower. To see this, consider the following example (taken from p. 249 in [Goodfellow et al., 2016]):

Example 2.7. Bagging. Suppose we have k regression models, and that each model makes an error $\epsilon_i \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ on each test observation, where $\mathbf{\Sigma}$ denotes the covariance matrix of the error terms. Suppose $\Sigma_{ii} = v$ and $\Sigma_{ij} = c$. The average prediction error of all k models is given by

$$\bar{\epsilon} = \frac{1}{k}\sum_i \epsilon_i,$$

and the variance of $\bar{\epsilon}$ is

$$\begin{aligned}\mathrm{Var}[\bar{\epsilon}] &= \mathbb{E}[\bar{\epsilon}^2] + \mathbb{E}[\bar{\epsilon}]^2 \\ &= \mathbb{E}\Big[\Big(\frac{1}{k}\sum_i \epsilon_i\Big)^2\Big] \\ &= \frac{1}{k^2}\mathbb{E}\Big[\sum_i\Big(\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j\Big)\Big] \\ &= \frac{1}{k}v + \frac{k-1}{k}c.\end{aligned}$$

If the errors are perfectly correlated, i.e. $c = v$, $\mathrm{Var}[\bar{\epsilon}] = v$ and model averaging does nothing. If the errors are independent, i.e. $c = 0$, $\mathrm{Var}[\bar{\epsilon}] = \frac{1}{k}v$ is inversely proportional to the number of models in the ensemble. ⌟

In a classification setting the bagged estimate may be expressed as a vector where each element denotes the proportion of models that assigned the input to class k. The function $\hat{f}_{bag}(x)$ can be used to select the class with the largest proportion of "votes" and classify it accordingly. Alternatively, we can average all the estimated class probabilities to obtain a vector of mean probabilities. We then base our prediction on whichever class is associated with the largest mean prediction, rather than use a voting scheme. See chapter 8.7 in [Hastie et al., 2001] for details.

2.5 Model Validation

In section 2.3.2 we briefly mentioned that hyperparameters control the behaviour of machine learning algorithms. The degree of a polynomial in ordinary regression is an example of a hyperparameter. The degree controls the capacity of the fitted model, and as we saw in fig. 2.1 it can be fit arbitrarily well by choosing a high enough degree.

We have also stated the importance of holding out a test set so that we can estimate how well our model performs on unseen data. It is important that the test set is not used to make choices about the learning algorithm, such as hyperparameter values. To preserve the independence of the test set, we hold out an additional set of observations from the training set. This is called a validation set. The validation set approach (see fig. 2.4) lets us estimate the training error as a function of our hyperparameter values, and we can adjust them accordingly.

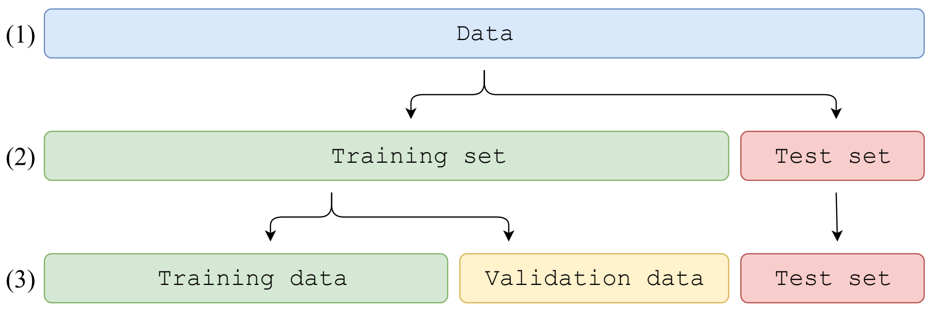


Fig. 2.4: We partition the available data into training, validation and test sets. We start at step (1) with the full data set. In step (2) we divide the data into a training set and a test set. The test set is ignored until it is time to evaluate the model. In step (3) we partition the training data into a training set and a validation set. The validation set is used for hyperparameter tuning and model assessment.

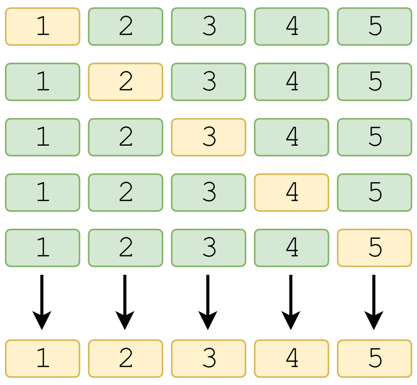


Fig. 2.5: *K*-fold cross-validation. The data is randomly split into $k = 5$ folds. In each iteration the model is trained on four folds (in green) and an estimate of the test error is calculated on the left-out fold (in yellow). The errors (bottom row) are averaged to obtain the final test error estimate.

2.5.1 Cross-validation

Sometimes we have a small amount of data at our disposal, which means that partitioning the data into training and validation sets can be problematic. An alternative procedure is to randomly partition the data into k groups, called folds, of approximately equal size. The model is fit on $k-1$ folds and the left out fold is treated as a validation set. The procedure is repeated k times, resulting in k estimates of the test error, denoted $\mathrm{CV}_1, ..., \mathrm{CV}_k$. The k-fold cross-validation estimate of the test error is given by

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} CV_i.$$

		Prediction		
		Positive	Negative	Total
Ground truth	Positive	TP	FN	TP + FN
	Negative	FP	TN	FP + TN
	Total	TP + FP	FN + TN	N

Table 2.1: Each cell in the confusion matrix gives the absolute counts of instances classified by the model (column) and arranges them by their true label (row). Moving clockwise from the top left, we have true positives (TP), false negatives (FN), true negatives (TN) and false positives (FP). The accuracy is given by $(\mathrm{TP} + \mathrm{TN})/N$.

Figure 2.5 shows the case where $k = 5$. Note that cross-validation techniques can be used for both regression and classification problems. In addition to model assessment, k-fold cross-validation is typically used to select appropriate values of hyperparameters. The cross-validation error is calculated for each value of one or more hyperparameters. The best hyperparameters are the ones associated with the lowest cross-validation error. Searching for the optimal value of a hyperparameter is also known as tuning.

2.5.2 Confusion matrix

Another useful tool for model assessment in classification problems is the confusion matrix. Generally, a confusion matrix is a frequency table that displays which kinds of errors a model is making. In the binary case where $y \in \{0, 1\}$ there are four possible outcomes:

- True positive (TP) $\Rightarrow \hat{y} = 1,\ y = 1$.
- True negative (TN) $\Rightarrow \hat{y} = 0,\ y = 0$.
- False positive (FP) $\Rightarrow \hat{y} = 1,\ y = 0$ (type I error).
- False negative (FN) $\Rightarrow \hat{y} = 0,\ y = 1$ (type II error).

The confusion matrix gives a quick overview of how the model performs on the test data. In a general classification setting where $y \in \{1, ..., K\}$ the accuracy of the classifier is given by the sum of the diagonals over the total amount of predictions made. The confusion matrix is the basis for a range of different performance measures, such as the area under the receiving operator characteristic curve (ROC AUC). For details, see [Hanley and McNeil, 1982].

2.6 Basics of Information Theory

Information theory [Shannon, 1948] is the study of how to quantify the amount of information present in a signal transmitted over a noisy channel. Ideas from information theory are used in a machine learning context to characterise probability distributions, or to describe similarities between different probability distributions. In this section we follow [Goodfellow et al., 2016] closely.

A fundamental quantity in information theory is self-information, defined as

$$I(x) = -\log p(x).$$

Informally, self-information conveys the intuition that an unlikely event contains more information than a very likely event. Entropy is the expected value of the self-information, and can be viewed as a measure of the uncertainty associated with an observation:

$$H[p(x)] = -\mathbb{E}[\log p(x)].$$

A closely related quantity is the Kullback-Leibler (KL) divergence. If we have two distinct probability distributions p and q over the same random variable x, then the KL divergence is given by

$$\begin{aligned}\mathrm{KL}[p(x)||q(x)] &= \mathbb{E}_{p(x)}\left[\log\frac{p(x)}{q(x)}\right] \\ &= \mathbb{E}_{p(x)}[\log p(x)] - \mathbb{E}_{p(x)}[\log q(x)].\end{aligned} \tag{2.9}$$

In machine learning applications, the KL divergence is often used as a measure of similarity between distributions. It can be shown that $\mathrm{KL}[p(x)||q(x)] \geq 0$. See section 1.6.1 of [Bishop, 2006] for a proof of this. Minimising the KL divergence is therefore the same as maximising $\mathbb{E}_{p(x)}[\log q(x)]$, which is known as the cross-entropy in information theoretic terms. The cross-entropy loss shows up in many machine learning software libraries, and as stated in section 2.3.1 minimising the cross-entropy corresponds to minimising the negative log-likelihood.

Finally, the mutual information is a measure of dependence between two random variables x and y drawn from a joint probability distribution $p(x, y)$. It is defined as

$$\begin{aligned}\mathrm{MI}[p(x), p(y)] &= H[p(x)] - H[p(x)|p(y)] \\ &= H[p(y)] - H[p(y)|p(x)],\end{aligned}$$

where the second term on the right-hand side is the conditional entropy (see e.g. [Bishop, 2006] for details). The mutual information can be interpreted as the amount of information gained about one variable by knowing the value of another variable.

Chapter 3

Deep Learning

Neural networks are a family of models with a long and rich history (see chapter 1.2 in [Goodfellow et al., 2016] for an overview). They have periodically fallen in and out of favour with researchers, but thanks to advances in computing hardware the popularity of so-called deep neural networks has exploded. This has resulted in a relatively new field of machine learning research called deep learning, which will be the focus of this chapter.

In section 3.1 we start by introducing the basic idea of a neural network with the help of a simple example. In sections 3.2 and 3.3 we introduce the fundamentals of a deep neural network. Section 3.4 gives an overview of the optimisation and backpropagation algorithms used to fit neural networks. In sections 3.5 and 3.6 we introduce some strategies to prevent overfitting. Of particular importance is the stochastic regularisation technique known as dropout training, which we will revisit in chapter 4 in the context of so-called Bayesian neural networks. Finally, in section 3.7 we introduce a specialised neural network that is typically used for computer vision problems, called a convolutional neural network. We primarily rely on the following sources and the references therein:

- Chapters 6, 7, 8 and 9 in [Goodfellow et al., 2016] (optimisation, neural networks, convolutional neural networks, activation functions, gradient descent, dropout, regularisation).
- Chapters 38 and 39 in [MacKay, 2002] (single-neuron classifiers, neural networks).
- Chapter 12 in [Gonzalez and Woods, 2017] (backpropagation, convolutional neural networks).

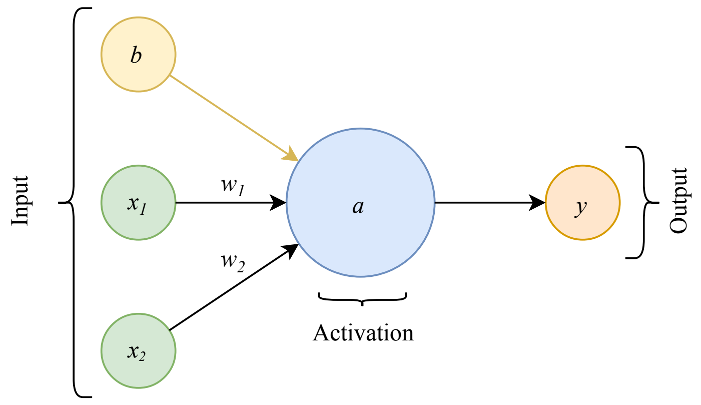


Fig. 3.1: A common graphical depiction of a single-neuron classifier. The arrows represent the weights of the network and the circles, also called neurons, represent the inputs to the next layer. The figure is a graphical representation of the model $y = \sigma(w_1x_1 + w_2x_2 + b)$.

Most books that deal with neural networks drop the generic parameter notation $\boldsymbol{\theta}$ and instead denote the weights by $\boldsymbol{w}$. We follow this convention for most of this chapter, with the exception of section 3.4.1 where we revert to the parameter notation $\boldsymbol{\theta}$ for brevity.

3.1 A Very Small Neural Network

We will introduce the concept of a neural network through the simplest possible case: The single-neuron classifier. Many statistical models can be viewed as single-neuron classifiers, and it is a useful representation when introducing the basic terminology of neural networks. In this and the next section we follow [MacKay, 2002] and [Gonzalez and Woods, 2017] closely.

Example 3.1 (Single-neuron classifier.). The single-neuron classifier consists of a D-dimensional input vector $\boldsymbol{x}^\intercal = (x_1, ..., x_D)$, where each element is associated with a corresponding weight from the parameter vector $\boldsymbol{w}^\intercal = (w_1, ..., w_D)$. As mentioned in section 2.1, in most cases we also have an additional parameter w_0 associated with an input $x_0 := 1$ which is called the bias[1]. Most machine learning books denote the bias term in neural networks by b and keep it separate from the parameter vector $\boldsymbol{w}$. This is the convention we will follow.

[1]Not to be confused with the definition of bias introduced in chapter 2.

The single-neuron classifier is commonly called a feedforward model. This means that information flows from the input via some intermediate neuron to the output (see figure 3.1 for an example where $D = 2$). The intermediate neuron computes a weighted sum, referred to as the net input:

$$z = \boldsymbol{x}^\mathsf{T}\boldsymbol{w} + b.$$

The output of this neuron is generally not seen, and is therefore referred to as a hidden neuron. The output is fed forward to the final neuron y, which applies a non-linear function $\sigma(z)$ to the input. For this example we will use the logistic function:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

The full, single-neuron classifier is then given by the composition of the weighted sum and the non-linear function $\sigma(\cdot)$:

$$\begin{aligned} y &= \sigma(\boldsymbol{x}^\mathsf{T}\boldsymbol{w} + b) \\ &= \frac{1}{1 + \exp(-(\boldsymbol{x}^\mathsf{T}\boldsymbol{w} + b))}. \end{aligned}$$

This may look familiar to the reader: It is the logistic regression model from example 2.2. ⌟

3.2 Multi-Layer Perceptrons

In the previous section we introduced neurons and the non-linear transformation $\sigma(\cdot)$. In neural network terminology, these are collectively referred to as components of the model's architecture. The remarkable success of neural networks in recent years is a result of increasingly complicated architectures and novel non-linear functions. The most basic neural network is the multi-layer perceptron[2] (MLP). The MLP is typically composed of input neurons, hidden neurons and output neurons which are grouped together and arranged in grid-like structures called layers. Each hidden layer is a

[2]The term perceptron refers to the non-linear step function in the original development of the single-neuron classifier [Rosenblatt, 1958].

function of the preceding layer, and the term hidden refers to the fact that we do not directly observe the output of these layers[3].

Hidden layers consist of neurons that first compute a weighted sum of all the neurons in the previous layer, referred to as the net input. The net input is fed through a scalar-valued, differentiable, non-linear function called the activation function[4]. The activations indicate how "excited" the hidden neurons are by the input, which is a neuroscientifically inspired way of saying that the neurons have detected some meaningful pattern in the input. The activations are subsequently weighted and either (a) fed to the next hidden layer where the same steps are repeated, or (b) fed to the output layer where a final activation function is applied and predictions are made.

Since every neuron in a layer is a function of all neurons in the previous layer, MLPs are often referred to as fully connected neural networks. An MLP is fully specified by its weights, biases and activation functions, and can be viewed as a composition of non-linear functions. Figure 3.2 shows a fully connected two-layer[5] neural network with one hidden layer.

In the following we will give a mathematical description of a neural network. For simplicity we assume a single input vector $\boldsymbol{x} \in \mathbb{R}^D$. It is straightforward to extend the following to multi-dimensional arrays, also known as tensors. See chapter 12 in [Gonzalez and Woods, 2017] for details.

Let $l = 2, ..., L$ denote the layers in a neural network. Let $a_j(l)$ denote the activation of neuron j in layer l. If $l = 1$, the activation is simply the input

$$a_j(1) = x_j, \tag{3.1}$$

where $j = 1, ..., D$ (i.e. the dimensionality of $\boldsymbol{x}$). Let the net input to neuron i in layer l be denoted by

$$z_i(l) = \sum_{j=1}^{n_{l-1}} w_{ij}(l)a_j(l-1) + b_j(l), \quad i = 1, ..., n_l, \tag{3.2}$$

[3]Strictly speaking, we do not observe the output of the input layer either, but it is not considered a hidden layer.

[4]The single-neuron classifier in section 3.1 applies an identity transformation in the hidden neuron and a logistic activation function in the output neuron. Alternatively, we could apply the logistic activation in the hidden neuron and the identity transformation in the output neuron. Both yield the same result.

[5]The literature is somewhat inconsistent on this point, but generally the input layer is not counted when specifying the number of layers in a network.

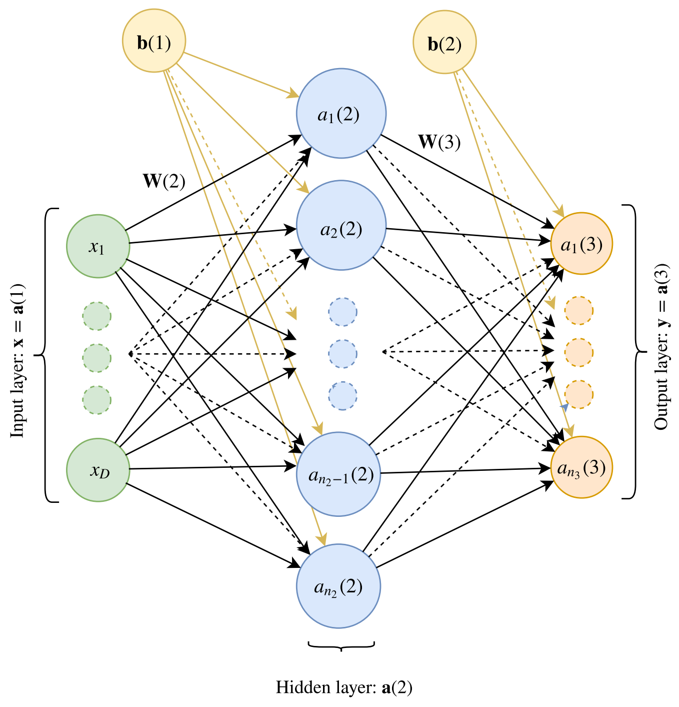


Fig. 3.2: A two-layer neural network.

where n_l denotes the number of neurons in layer l. The weight $w_{ij}(l)$ describes the connection between the j'th neuron in layer $l-1$ and the i'th neuron in layer l. The ordering of the parameter subscripts may seem counterintuitive, but are defined that way in order to avoid matrix transposition in the equations that describe how an input is fed forward through the network (eq. 3.5).

The activation of the neuron i in layer l is given by

$$a_i(l) = \sigma(z_i(l)), \quad i = 1, ..., n_l, \tag{3.3}$$

where $\sigma(\cdot)$ denotes the activation function. Typically, the same activation function will be used throughout the network, except in the final output layer. The output of i'th neuron in the final layer of the network is given by

$$a_i(L) = \sigma(z_i(L)). \tag{3.4}$$

We will denote the output by $\hat{y}_i = a_i(L)$. In matrix form, equations 3.1–3.4 are given by

$$\begin{aligned} \boldsymbol{a}(1) &= \boldsymbol{x}, \\ \boldsymbol{z}(l) &= \boldsymbol{W}(l)\boldsymbol{a}(l-1) + \boldsymbol{b}(l), \quad l = 2, ..., L, \end{aligned} \tag{3.5}$$

where $\boldsymbol{z}(l)$ is a $n_l \times 1$ column vector of net inputs to layer l. The $n_l \times n_{l-1}$ weight matrix $\boldsymbol{W}(l)$ is given by

$$\boldsymbol{W}(l) = \begin{bmatrix} w_{11}(l) & w_{12}(l) & \dots & w_{1n_{l-1}}(l) \\ w_{21}(l) & w_{22}(l) & \dots & w_{2n_{l-1}}(l) \\ \vdots & \vdots & \vdots & \vdots \\ w_{n_l1}(l) & w_{n_l2}(l) & \dots & w_{n_ln_{l-1}}(l) \end{bmatrix}.$$

The $n_{l-1} \times 1$ column vector $\boldsymbol{a}(l-1)$ containing the activations of layer $l-1$ is given by

$$\boldsymbol{a}(l-1) = \begin{bmatrix} \sigma(z_1(l-1)) \\ \sigma(z_2(l-1)) \\ \vdots \\ \sigma(z_{n_{l-1}}(l-1)) \end{bmatrix},$$

where the non-linear transformation $\sigma(\cdot)$ is applied elementwise to the net input $\boldsymbol{z}(l-1)$.

Equations 3.1–3.4 and their equivalent matrix formulations in eq. 3.5 are collectively referred to as forward propagation or a forward pass. The following algorithm shows the forward pass of a single training example where $\hat{\boldsymbol{y}}$ is the output of the network:

Algorithm 2: Forward pass

```
Require: Network depth, $L$
Require: $\boldsymbol{W}(l), l = 2, ..., L$
Require: $\boldsymbol{b}(l), l = 2, ..., L$
Require: Activation function $\sigma(\cdot)$
Require: Input $\boldsymbol{x}$
$\boldsymbol{a}(1) = \boldsymbol{x}$
for $l = 2, ..., L$ do
    $\boldsymbol{z}(l) = \boldsymbol{W}(l)\boldsymbol{a}(l-1) + \boldsymbol{b}(l)$
    $\boldsymbol{a}(l) = \sigma(\boldsymbol{z}(l))$
end
$\hat{\boldsymbol{y}} = \boldsymbol{a}(L)$
```

A two-layered MLP can approximate any function f arbitrarily well provided there are enough neurons in the hidden layers. This result is known as the universal approximation theorem [Cybenko, 1989; Hornik et al., 1989] and holds for a wide range of activation functions. The universal approximation theorem guarantees that there exists an MLP that can represent a given function f, but it does not guarantee that we will be able to learn f. Furthermore, the size of the network required to learn f may be infeasibly large. Empirically, using deeper models (i.e. more hidden layers) can reduce the number of neurons in each hidden layer that are required to learn f. In recent years the most successful networks have grown deeper and deeper, particularly in the field of image recognition (see for example [He et al., 2016] and [Huang et al., 2017]).

3.3 Activation functions

The flexibility of neural networks stems from the non-linearities introduced by the activation functions[6]. As stated in the previous section, the activation function is a differentiable function which indicates how a neuron responds to the input. Differentiability is an important property because deep learning models are trained using gradient descent (see section 2.3.2). In the following we present some of the most common activation functions. For ease of exposition we will assume scalar-valued inputs (with the exception of the softmax activation).

Sigmoid activations

A sigmoid function is a bounded, differentiable real function that is defined for all real inputs and has a non-zero derivative everywhere [Han and Moraga, 1995]. It maps any real input into a bounded interval, such as $[0, 1]$ in the case of the logistic activation function and $[-1, 1]$ in the case of the hyperbolic tangent activation function.

Sigmoid functions tend to saturate across most of their domain, which leads to the so-called vanishing gradient problem. For example, as the input to the logistic activation function becomes arbitrarily large (negative or positive), the output will saturate at 0 or 1, causing the derivative to be practically 0 (see e.g. fig. 3.3b). This makes gradient-based learning difficult, and consequently sigmoid activation functions are discouraged in hidden layers (for details, see [Glorot and Bengio, 2010]).

[6]Many textbooks take the basis function view when explaining neural networks. A basis function $\boldsymbol{\phi}(\boldsymbol{x})$ is a fixed, non-linear function of the input, which allows us to estimate non-linear functions using linear models of the form $y = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{w}$. By allowing the basis functions to be adaptive rather than fixed, we can view a neural network as a composition of parameterised basis functions where the parameters are learned during training. See e.g. chapter 5 in [Bishop, 2006] for details.

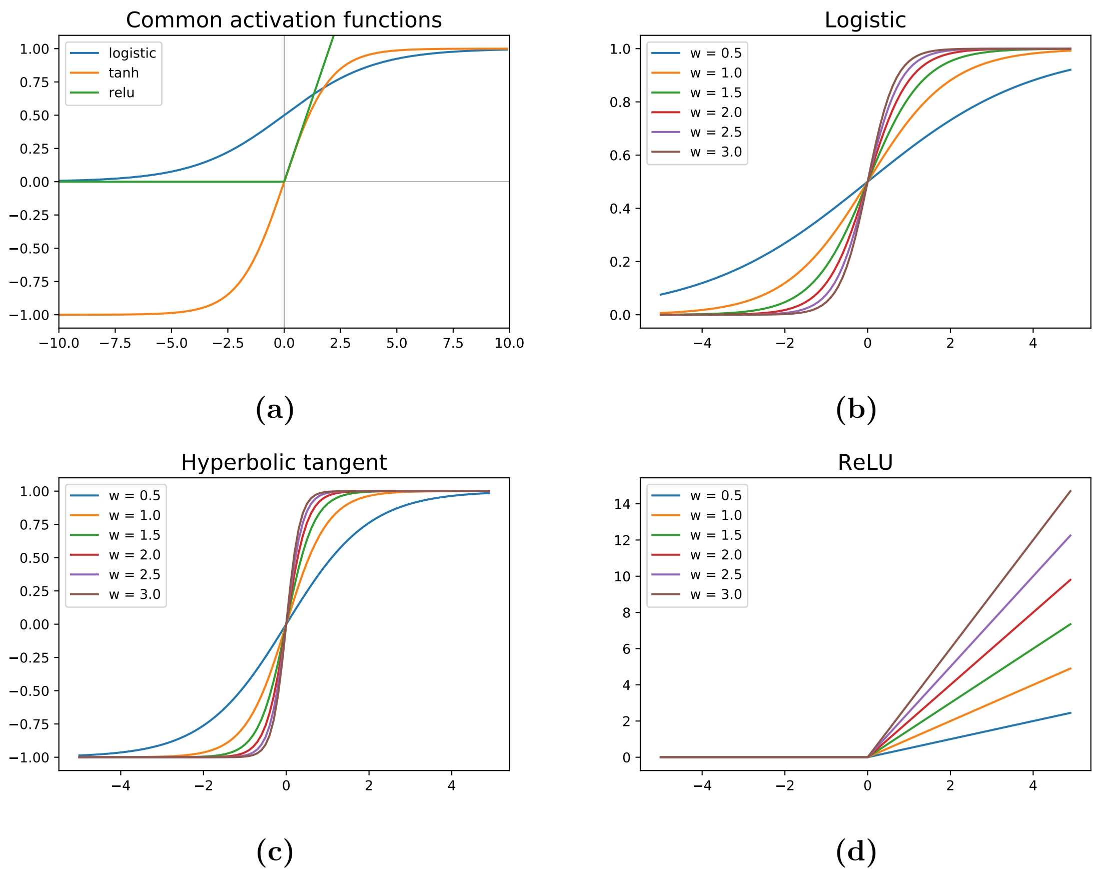


Fig. 3.3: Fig. 3.3a shows how the sigmoid activation functions compare to each other and to the ReLU. Given input $z = wx + b$, figures 3.3b–3.3d show how the individual activation functions change with respect to w (assuming $b = 0$).

The logistic function is typically used in settings where we want to model the outcome of a binary variable $y \in \{0, 1\}$ by assigning the input to 0 or 1 based on some predetermined threshold. It is given by

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

If a sigmoid activation must be used internally in a network, then the hyperbolic tangent given by

$$\sigma(z) = \frac{1 - \exp(-2z)}{1 + \exp(-2z)}$$

is generally preferred because it behaves better in optimisation. The reader is referred to [LeCun et al., 1998] for details.

Rectified Linear Unit

The rectified linear unit (ReLU) [Nair and Hinton, 2010], or variants of it, is considered the default activation function for hidden layers in modern deep learning models [Goodfellow et al., 2016]. The ReLU is given by

$$\sigma(z) = \max(0, z).$$

It is a piecewise linear function, which makes optimisation with gradient-based learning efficient. It does not saturate for positive values, and consequently it alleviates the vanishing gradient problem. However, zero-valued ReLU-outputs will erradicate the gradient, which inhibits learning. Variants such as the leaky ReLU [Maas et al., 2013] deal with this problem. Also note that the ReLU is not differentiable at $z = 0$. In practice this is not a problem, as it is extremely unlikely that the we reach a point in training where the loss is exactly 0.

Softmax

The softmax activation function is most commonly found in the output layer of a network whenever we want to model a probability distribution over K different outcomes, typically in a classification setting. The softmax takes a vector $\boldsymbol{z} \in \mathbb{R}^K$ as input, and outputs a vector where the k'th element is interpreted as the probability of belonging to class k, given by

$$\sigma_k(\boldsymbol{z}) = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}, \quad k = 1, ..., K.$$

Furthermore, the elements of the softmax are subject to the constraints $\sigma_k(\boldsymbol{z}) \geq 0$ and $\sum_{k=1}^{K} \sigma_k(\boldsymbol{z}) = 1$. The softmax is a generalisation of the logistic function to a situation with multiple outcomes. In practical applications, the predicted class k is given by $\arg\max_k \sigma_k(\boldsymbol{z})$.

3.4 Training Deep Neural Networks

In section 2.3.2 we introduced stochastic gradient descent (often referred to as "vanilla" SGD in deep learning), an optimisation algorithm that samples a mini-batch of m

examples and calculates an unbiased estimate of the gradient of the loss function. SGD is the most common optimisation algorithm in deep learning. However, when applied to complicated non-convex loss functions such as the cross-entropy loss (eq. 2.6), "vanilla" SGD tends to perform sub-optimally. As a consequence, several variants of SGD have been introduced. In this section we will briefly introduce the momentum algorithm. The reader is referred to [Ruder, 2016] for an excellent summary of other variants. We temporarily revert to the parameter notation $\boldsymbol{\theta}$ introduced in chapter 2 for brevity.

3.4.1 Gradient Descent in Deep Learning

One of the most widely used variants of SGD is the momentum algorithm [Qian, 1999], named after the momentum term $\boldsymbol{v}$ that accumulates past gradients in order to speed up the learning process. The momentum is an exponentially weighted moving average of the gradient, given by

$$\boldsymbol{v}_t \leftarrow \alpha \boldsymbol{v}_{t-1} + \eta \boldsymbol{g},$$
$$\boldsymbol{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}_{t-1}).$$

The hyperparameter $\alpha \in (0, 1)$ determines how fast the accumulated gradients decay and η denotes the learning rate as before. This gives the update rule

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t.$$

Informally, the momentum term increases when the gradients align, resulting in faster convergence. SGD with momentum is summarised in the following algorithm:

Algorithm 3: Stochastic gradient descent with momentum

```
Require: Learning rate $\eta$, momentum parameter $\alpha$
Require: Initial values for $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
while stopping criterion not met do
    Randomly sample mini-batch of $m$ input-output pairs $\{\boldsymbol{x}_i, y_i\}_{i=1}^{m}$
    Compute gradient estimate: $\boldsymbol{g} = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}_{t-1})$
    Update momentum: $\boldsymbol{v}_t \leftarrow \alpha \boldsymbol{v}_{t-1} + \eta \boldsymbol{g}$
    Update parameters: $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t$
end
```

One of the challenges when using gradient descent is setting the appropriate learning rate η. Typically the deep learning practitioner uses a combination of experience and brute-force methods like grid-search combined with cross-validation to find the best learning rates. See chapter 11 in [Goodfellow et al., 2016] for details. Alternatively, more automated approaches such as Bayesian optimisation [Snoek et al., 2012] can be used.

It is also common to decay the learning rate during training according to some predetermined schedule. The idea is that a high initial setting of η will move the parameters towards rough but reasonable estimates relatively quickly. Gradually decreasing η will then trigger a more refined search for the best parameter values. This is known as learning rate annealing. Recently, cyclical learning rate annealing [Smith, 2015, 2017] has gained popularity. As is implied by the name, the learning rate is reset at a given or random interval, causing the gradient to potentially "jump" out of unproductive areas of the loss surface. Furthermore, there are variants of SGD which adaptively change the learning rate for each individual weight. The most widely used are RMSProp, AdaDelta and Adam (see [Ruder, 2016] or chapter 8 in [Goodfellow et al., 2016] for an overview).

It is also important to be aware of different parameter initialisation strategies. Since the weights of the network typically start out as random values, researchers have developed different schemes for determining the best initial values of the parameters. Some widely used initialisation strategies are based on [Glorot and Bengio, 2010] and [He et al., 2015]. Furthermore, networks can be initialised with weights that are pre-trained on a different data set. For example, in image classification tasks many networks are initialised using weights that are pre-trained on the ImageNet database [Deng et al., 2009], then fine-tuned for the task at hand.

3.4.2 Backpropagation: Estimating the Gradient of the Loss

The backpropagation algorithm [Rumelhart et al., 1986], often referred to as backprop, is the bread and butter of neural network training. In algorithm 2 we described how information flows forward through a network, ultimately resulting in a prediction $\hat{\boldsymbol{y}}$. We then calculate a measure of prediction error represented by a regularised cost function C. Backprop allows information from the cost function to flow backwards through the network, yielding gradients on the net inputs to each layer. This step is interpreted as an indication of how each layer should adjust its output in order to reduce the prediction error. From these gradients we can obtain the gradients on the weights and biases for use in a parameter update. This highlights an important aspect of backprop: It only estimates the gradients of the network parameters. An optimisation algorithm, such as those outlined in the previous section, performs the actual parameter updates.

Backprop is essentially a highly efficient implementation of the chain rule of calculus. The chain rule offers a way of computing the derivative of a function which is composed of two or more functions. As an example, let $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. Suppose $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y}) = f(g(\boldsymbol{x}))$. Then the chain rule states that

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

In matrix-vector notation, the gradient of z with respect to $\boldsymbol{x}$ can be expressed as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^{\mathsf{T}} \nabla_{\boldsymbol{y}} z,$$

where $\partial \boldsymbol{y} / \partial \boldsymbol{x}$ is the $n \times m$ Jacobian matrix of g.

For simplicity we omit the regularisation term and develop the backprop equations for a single training example $(\boldsymbol{x}, \boldsymbol{y})$. In practice backprop uses a mini-batch of m inputs. It is fairly straightforward to extend the following equations from vectors to tensors. The reader is referred to chapter 12 in [Gonzalez and Woods, 2017] or chapters 6.5.2 and 6.5.6 in [Goodfellow et al., 2016] for details. We follow [Gonzalez and Woods, 2017] closely in notation.

As stated earlier, the aim of backprop is to estimate the gradients of the weights of the network so that we can adjust them accordingly in an effort to reduce the prediction error. The weights of the network are contained in the net inputs to each neuron (eq. 3.5), and consequently we would like to express how the cost function C changes with respect to the net input of each neuron. In other words, we want to find $\delta_j(l) = \partial C / \partial z_j(l)$, where $z_j(l)$ is the net input to neuron j in layer l. Intuitively, this can be thought of as a measure of error which indicates how sensitive our cost function is to the parameters in layer l. Starting at the final layer L, we can define

$$\delta_j(L) = \frac{\partial C}{\partial z_j(L)}. \tag{3.6}$$

By applying the chain rule, we can express eq. 3.6 in terms of the j'th activation $a_j(L)$ of layer L:

$$\delta_j(L) = \frac{\partial C}{\partial a_j(L)} \frac{\partial a_j(L)}{\partial z_j(L)} = \frac{\partial C}{\partial a_j(L)} \sigma'(z_j(L)). \tag{3.7}$$

Recall that $a_j(L) = \sigma(z_j(L))$, so $\partial a_j(L)/\partial z_j(L) = \sigma'(z_j(L))$. The relationship between the net input of any neuron and its activation is the same for all layers (with the exception of the input layer $l = 1$), hence we can define

$$\delta_j(l) = \frac{\partial C}{\partial z_j(l)} \tag{3.8}$$

for $l = L-1, L-2, ..., 2$. If we can express $\delta_j(l)$ in terms of $\delta_j(l+1)$, then we can start with $\delta_j(L)$ and work our way backwards to obtain $\delta_j(L-1)$, $\delta_j(L-1)$, ..., $\delta_j(2)$. Noting that the error in neuron j of layer l will influence all neurons in the subsequent layer $l+1$, we apply the chain rule to obtain

$$\begin{aligned} \delta_j(l) &= \sum_i \frac{\partial C}{\partial z_i(l+1)} \frac{\partial z_i(l+1)}{\partial a_j(l)} \frac{\partial a_j(l)}{\partial z_j(l)} \\ &= \sum_i \delta_i(l+1) \frac{\partial z_i(l+1)}{\partial a_j(l)} \sigma'(z_j(l)), \end{aligned} \tag{3.9}$$

where i sums over the number of neurons in layer $l+1$. Now, recall that $z_i(l) = \sum_{j=1}^{n_{l-1}} w_{ij}(l) a_j(l-1) + b_j(l)$ (eq. 3.2). Therefore

$$\frac{\partial z_i(l+1)}{\partial a_j(l)} = w_{ij}(l+1).$$

Substituting this into eq. 3.9 and rearranging the terms gives

$$\delta_j(l) = \sum_i w_{ij}(l+1)\delta_i(l+1)\sigma'(z_j(l)), \quad L = L-1, ..., 2. \tag{3.10}$$

So far we have established a framework for expressing how the output error changes with respect to the net input to every neuron in the network. Eqs. 3.6–3.10 are thus intermediary steps towards obtaining final expressions for $\partial C/\partial w_{ij}(l)$ and $\partial C/\partial b_i(l)$ in terms of $\delta_i(l)$. We apply the chain rule again, obtaining

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}(l)} &= \frac{\partial C}{\partial z_i(l)} \frac{\partial z_i(l)}{\partial w_{ij}(l)} \\ &= \delta_i(l) a_j(l-1). \end{aligned} \tag{3.11}$$

The second equality follows from

$$\frac{\partial z_i(l)}{\partial w_{ij}(l)} = a_j(l-1).$$

Similarly,

$$\begin{aligned} \frac{\partial C}{\partial b_i(l)} &= \frac{\partial C}{\partial z_i(l)} \frac{\partial z_i(l)}{\partial b_i(l)} \\ &= \delta_i(l), \end{aligned} \tag{3.12}$$

where we have used

$$\frac{\partial z_i(l)}{\partial b_i(l)} = 1.$$

Eqs. 3.11 and 3.12 provide the rate of change of the cost function with respect to every parameter in the network. The above equations can be formulated conveniently in matrix notation:

$$\boldsymbol{\delta}(L) = \begin{bmatrix} \delta_1(L) \\ \delta_2(L) \\ \vdots \\ \delta_{n_L}(L) \end{bmatrix} = \begin{bmatrix} \frac{\partial C}{\partial a_1(L)} \sigma'(z_1(L)) \\ \frac{\partial C}{\partial a_2(L)} \sigma'(z_2(L)) \\ \vdots \\ \frac{\partial C}{\partial a_{n_L}(L)} \sigma'(z_{n_L}(L)) \end{bmatrix} = \nabla_{\boldsymbol{a}(L)} C \odot \sigma'(\boldsymbol{z}(L)).$$

Here $\odot$ denotes the elementwise product, also known as the Hadamard product. Eq. 3.10 can also be expressed in matrix notation, resulting in

$$\begin{aligned} \boldsymbol{\delta}(l) = \begin{bmatrix} \delta_1(l) \\ \delta_2(l) \\ \vdots \\ \delta_{n_l}(l) \end{bmatrix} &= \begin{bmatrix} \sum_i w_{i1}(l+1)\delta_i(l+1)\sigma'(z_1(l)) \\ \sum_i w_{i2}(l+1)\delta_i(l+1)\sigma'(z_2(l)) \\ \vdots \\ \sum_i w_{in_l}(l+1)\delta_i(l+1)\sigma'(z_{n_l}(l)) \end{bmatrix} \\ &= \boldsymbol{W}^\mathsf{T}(l+1)\boldsymbol{\delta}(l+1) \odot \sigma'(\boldsymbol{z}(l)), \end{aligned}$$

where $\boldsymbol{W}^\mathsf{T}(l+1)$ is a $n_l \times n_{l+1}$ matrix and $\boldsymbol{\delta}(l+1)$ is an $n_{l+1} \times 1$ vector. Finally, eqs. 3.11 and 3.12 can be expressed as

$$\nabla_{\boldsymbol{W}(l)} C = \boldsymbol{\delta}(l)\boldsymbol{a}^\mathsf{T}(l-1) \tag{3.13}$$

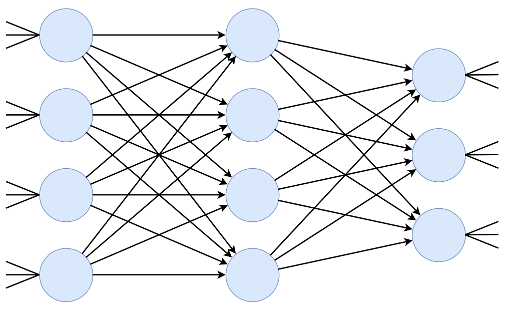
$$\nabla_{\boldsymbol{b}(l)} C = \boldsymbol{\delta}(l). \tag{3.14}$$

Algorithm 4 summarises training on a single example $(\boldsymbol{x}, \boldsymbol{y})$ using backpropagation. In the update step, $R(\boldsymbol{\theta})$ is shorthand notation for the regularisation term where $\boldsymbol{\theta}$ contains all the network parameters.

Algorithm 4: Training a deep neural network

```
Require: Network depth, $L$
Require: $\boldsymbol{W}(l), \boldsymbol{b}(l), l = 2, ..., L$, activations $\sigma(\cdot)$
Require: Input $\boldsymbol{x}$, target output $\boldsymbol{y}$
Require: SGD hyperparameters
while stopping criterion not met do
    Compute forward pass as described in algorithm 2.
    Calculate regularised cost and initial error term:
    $C = \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) + R(\boldsymbol{\theta})$
    $\boldsymbol{\delta}(L) = \nabla_{\boldsymbol{a}(L)} C \odot \sigma'(\boldsymbol{z}(L))$
    Backpropagate output error:
    for $l = L-1, L-2, ..., 2$ do
        $\boldsymbol{\delta}(l) = \boldsymbol{W}^{\mathsf{T}}(l+1)\boldsymbol{\delta}(l+1) \odot \sigma'(\boldsymbol{z}(l))$
    end
    Update parameters:
    for $l = 2, ..., L$ do
        $\nabla_{\boldsymbol{W}(l)} C = \boldsymbol{\delta}(l)\boldsymbol{a}^{\mathsf{T}}(l-1) + \nabla_{\boldsymbol{W}(l)} R(\boldsymbol{\theta})$
        $\nabla_{\boldsymbol{b}(l)} C = \boldsymbol{\delta}(l) + \nabla_{\boldsymbol{b}(l)} R(\boldsymbol{\theta})$
        $\boldsymbol{W}(l) \leftarrow \boldsymbol{W}(l) - \eta \nabla_{\boldsymbol{W}(l)} C$
        $\boldsymbol{b}(l) \leftarrow \boldsymbol{b}(l) - \eta \nabla_{\boldsymbol{b}(l)} C$
    end
end
```

As mentioned at the start of this section, practical implementations of backprop apply the algorithm to a mini-batch of m inputs. The loss function is then a measure of the average prediction error of the m inputs and we would use a stochastic gradient descent method for optimisation (as outlined in the previous section). In practice the while-condition of algorithm 4 depends on the number of epochs. An epoch is the number of iterations over all training examples. An iteration is a completed forward and backward pass of one mini-batch. For example, if we have $N = 10.000$ training examples and we define the mini-batch size $m = 100$, each epoch contains $k = 100$ iterations (i.e. updates to the network parameters).


(a) Standard neural network.

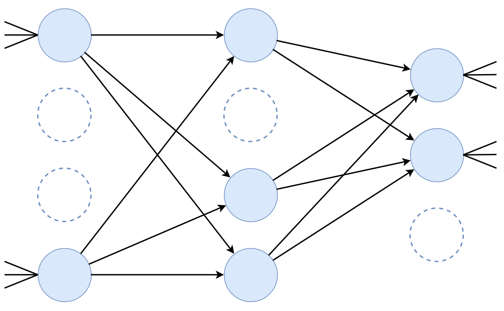

(b) Dropout neural network.

Fig. 3.4: In a dropout neural network neurons are switched off with probability $1 - p_{l-1}$, essentially isolating them from the rest of the network (fig. 3.4b).

3.5 Preventing Overfitting: Dropout

Overfitting is a major challenge when training neural networks, due to their high flexibility and large amount of parameters. One way to reduce the risk of overfitting is by averaging many models (see section 2.4.2). Dropout [Hinton et al., 2012; Srivastava et al., 2014] is a widely used stochastic regularisation technique that approximates model averaging, and has proven to be very effective. Dropout randomly removes non-output neurons along with all incoming and outgoing connections during training (see fig. 3.4). This is motivated by the observation that hidden neurons tend to co-adapt, which means that they learn to correct each others mistakes. The intuition behind dropout is that if the presence of neighbouring neurons is unreliable, then co-adaptation is prevented and each neuron is forced to learn how to detect meaningful features on its own.

In a dropout neural network the activations are multiplied elementwise with a random vector consisting of zeroes and ones, effectively turning some neurons off and keeping others active. Let $\boldsymbol{r}(l-1) \in \mathbb{R}^{n_{l-1}}$ denote a random vector of independent Bernoulli variables where $p(r_j(l-1) = 1) = p_{l-1}$ for $j = 1, \ldots, n_{l-1}$. Forward propagation through the l'th layer of a dropout neural network is given by

$$\begin{aligned}
&r_j(l-1) \sim \text{Bernoulli}(p_{l-1}), \\
&\boldsymbol{a}_{\text{drop}}(l-1) = \boldsymbol{a}(l-1) \odot \boldsymbol{r}(l-1), \\
&\boldsymbol{z}(l) = \boldsymbol{W}(l)\boldsymbol{a}_{\text{drop}}(l-1) + \boldsymbol{b}(l), \\
&\boldsymbol{a}(l) = \sigma(\boldsymbol{z}(l)).
\end{aligned} \tag{3.15}$$

Dropout is done independently for each neuron and each training example. As described in [Srivastava et al., 2014], dropout can be viewed as sub-sampling one of 2^n possible "thinned" networks from the parent network consisting of n neurons, where there is extensive weight-sharing between the sub-networks. Since the weights are shared by every conceivable sub-network, the vast majority that go unsampled will still arrive at reasonable parameter values without being explicitly trained. Consequently, it is possible to approximate an average of all 2^n sub-networks by scaling the weights by their associated dropout rates and turning off dropout at prediction time. This is called the weight scaling inference rule. It is applied so that the expected output at prediction time matches the expected output during training. Furthermore, Srivastava suggests that weight scaling is a reasonable approximation of Monte Carlo estimation (see section 4.3). As we shall see in chapter 4, recent research in fact casts dropout neural networks as Bayesian neural networks by leaving dropout on at test time and averaging the predictions. Another view of dropout is that we are regularising the weights by adding noise to the hidden neurons, which may help the optimisation process explore otherwise inaccessible parts of the parameter space.

Finally, it is interesting to note that dropout training is not restricted to neural networks. For example, in [Rashmi and Gilad-Bachrach, 2015] dropout is used to increase the performance of an ensemble of boosted regression trees. For details on boosting and regression trees, refer to e.g. chapters 8 and 9 in [Hastie et al., 2001].

3.6 Preventing Overfitting: Other Methods

In section 2.4 we introduced weight decay, which is a parameter norm penalty that we add to the loss function in order to reduce model complexity. Weight decay is frequently applied in a neural network setting, but there are also other important methods which we will outline in the following.

3.6.1 Early Stopping

During training, the training and validation set errors will typically decrease given an appropriate model. However, for sufficiently complex models, it is common to observe that the validation error at some point starts to increase even though the training error continues to decrease. This is a sign of overfitting. Obviously, we would like to retrieve the weights from before the validation error started to increase. This is what early stopping allows us to do. Typical implementations of early stopping save the model parameters every time they improve (as measured by the validation error), and the learning algorithm continues until some predefined stopping criteria is met. Usually this is when the parameters have stopped improving beyond a specified threshold. The weights associated with the best validation loss are then retrieved and used in the final model.

3.6.2 Data Augmentation

Finally, one of the best ways to prevent generalisation error is to train on more data. If more data is not available, we can artificially create some by transforming the limited data set we already have. Data augmentation is not easily applicable to every task, but a very typical application can be found in image classification. For example, given an input image $\boldsymbol{x}$ and an associated label k, we essentially add a "new" example of class k to the training data by randomly transforming $\boldsymbol{x}$ through shifting, scaling, rotation or flipping (or other transformations that make sense for our particular data set). Data augmentation can be thought of as synthesising new data and thereby increasing the size of the training set. This helps reduce the generalisation error.

3.7 Convolutional Neural Networks

There exists a wide array of specialised neural networks. One of the most well known is the convolutional neural network (CNN) [LeCun et al., 1990], which has been hugely successful in computer vision problems such as object recognition, detection and segmentation. A CNN makes clever use of spatial relationships in the grid-like topology of image data (which can be thought of as multidimensional array of pixels) to automatically learn how to detect the distinguishing features of an image.

In an MLP, the activation of every neuron of layer $l-1$ is fed as input to every neuron of layer l. This results in massive weight matrices for high-dimensional data such as images. CNNs rely on the insight that if an image feature (such as an edge or a corner) is useful in one spatial region of the input, it is probably useful in another. In practice, the spatial information contained in a neighbourhood of pixels is obtained through the convolution of the input and a small, learnable weight matrix called a kernel or filter. The result of the convolution is a feature map of neurons, where the activation of each neuron indicates the presence of the feature that a kernel has learned to detect.

Consequently, the neurons in an activation layer (feature map) share the same parameters, drastically reducing the number of weights for high-dimensional inputs such as images. The power of CNNs comes from the fact that the convolution operation also operates along the depth of the network. This enables the a CNN to form rich, high-level abstractions of low-level features in earlier layers by convolving a hierarchy of feature maps. Thus, the network is able to identify complex shapes, such as a dog or a cat, through complicated interactions between simple shapes, such as edges and blobs.

In the following we will take a closer look at the different components of a conventional CNN under the assumption that we are applying it to image data. We will start by introducing the convolution operation, before moving on to kernels, feature maps and pooling functions. Finally we will put all these pieces together to form the classic LeNet architecture developed in [LeCun et al., 1990]. In this section we follow [Gonzalez and Woods, 2017] and [Goodfellow et al., 2016] closely.

3.7.1 Convolution

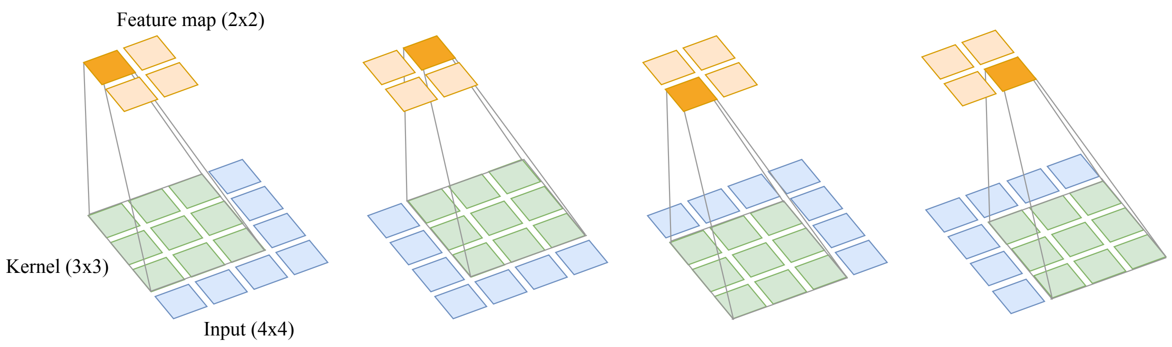
Since we are working with discretised data in the form of grids of pixels, we will focus on the discrete convolution and rely on (and extend) the notation provided in [Gonzalez and Woods, 2017]. The convolution of a two-dimensional input $a_{i,j}$ around the point (i, j) is given by

$$\begin{aligned} z_{i,j} &= \sum_m \sum_n w_{m,n} a_{i-m,j-n} \\ &= w * a_{i,j}, \end{aligned}$$

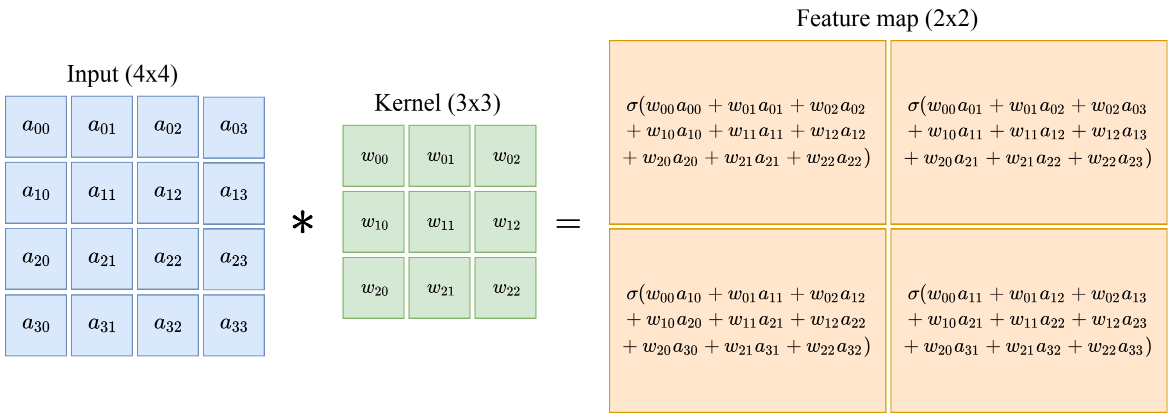
where $*$ denotes the convolution operation and $m \times n$ are the dimensions of the kernel. However, most machine learning libraries actually implement the cross-correlation[7], given by

$$\begin{aligned} z_{i,j} &= \sum_m \sum_n w_{m,n} a_{i+m,j+n} \\ &= w * a_{i,j}. \end{aligned}$$

[7]The terms cross-correlation and convolution are used interchangeably in the literature [Goodfellow et al., 2016]. We will follow this convention.



(a) Sliding filter view of convolution. The kernel (in green) is shifted over the input (in blue) and at each position the overlapping cells are multiplied and summed. Each cell in the feature map (in orange) is the result of a convolution of a corresponding spatial neighbourhood, called the receptive field (see fig 3.5b).


Input (4x4)
a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
$*$
Kernel (3x3)
w_{00} w_{01} w_{02}
w_{10} w_{11} w_{12}
w_{20} w_{21} w_{22}
$=$
Feature map (2x2)
$\sigma(w_{00}a_{00} + w_{01}a_{01} + w_{02}a_{02} + w_{10}a_{10} + w_{11}a_{11} + w_{12}a_{12} + w_{20}a_{20} + w_{21}a_{21} + w_{22}a_{22})$
$\sigma(w_{00}a_{01} + w_{01}a_{02} + w_{02}a_{03} + w_{10}a_{11} + w_{11}a_{12} + w_{12}a_{13} + w_{20}a_{21} + w_{21}a_{22} + w_{22}a_{23})$
$\sigma(w_{00}a_{10} + w_{01}a_{11} + w_{02}a_{12} + w_{10}a_{20} + w_{11}a_{21} + w_{12}a_{22} + w_{20}a_{30} + w_{21}a_{31} + w_{22}a_{32})$
$\sigma(w_{00}a_{11} + w_{01}a_{12} + w_{02}a_{13} + w_{10}a_{21} + w_{11}a_{22} + w_{12}a_{23} + w_{20}a_{31} + w_{21}a_{32} + w_{22}a_{33})$

(b) Simple example of convolution arithmetic. We have excluded the bias term for simplicity.

Fig. 3.5: Convolution operation in neural networks.

This is often visualised as sliding a filter over an input, multiplying the overlapping cells elementwise and summing the result. Note that we start the indices from 0 in this notation. As before, we add a bias term and perform a non-linear transformation of $z_{i,j}$ to obtain $a_{i,j}$ for the subsequent layer. Figure 3.5 shows a simplified visual representation of the convolution operation. To keep track of layers we introduce the layer index l as before:

$$\begin{aligned} z_{i,j}(l) &= \sum_m \sum_n w_{m,n}(l) a_{i+m,j+n}(l-1) + b(l) \\ &= w(l) * a_{i,j}(l-1) + b(l), \end{aligned}$$

where

$$a_{i,j}(l-1) = \sigma(z_{i,j}(l-1)).$$

As stated earlier, the input and outputs of convolutions are usually tensors in practical applications. This means we need additional indices to keep track of the weight connecting a neuron in the input channel g to the neuron of the output channel k:

$$\begin{aligned} z_{k,i,j}(l) &= \sum_g \sum_m \sum_n w_{k,g,m,n}(l) a_{g,i+m,j+n}(l-1) + b_k(l) \\ &= w_k(l) * a_{i,j}(l-1) + b_k(l). \end{aligned} \tag{3.16}$$

Eq. 3.16 concisely describes the local connectivity to a given region of the input (called the receptive field, given by m and n) and the full connectivity along the depth dimension (given by g). Note that the kernel $w_k(l)$ will always have the same depth dimension as the input. In practice there are several hyperparameters associated with the filters which also need to be taken into account.

3.7.2 Filter Hyperparameters

The output volume of a convolution is controlled by four hyperparameters[8]: the number of filters (K), the filter size[9] (F), the stride (S) and the amount of zero-padding (P).

We have already mentioned the receptive field, which correspond to the spatial dimensions of the filter. The number of filters will then correspond to the depth of the output volume. The stride determines how we shift the filter over the image, and S will then dictate how many pixels we skip when computing the net inputs to the feature map. Sometimes we may want to pad the input volume with zeroes in order to manipulate the dimensionality of the output volume. This is called zero-padding. Zero-padding decouples the filter size and the output volume size, allowing us to control one independently of the other. There are three main types of zero-padding:

- valid (the output size shrinks at each layer, i.e. downsampling),
- same (the output size is the same as the input size),
- full (the output size is larger than the input, i.e. upsampling).

[8]The notation we follow is provided in [Karpathy, 2016].

[9]The filter is typically square, which is why we denote the size F by a single value.

The reader is referred to chapter 9.5 in [Goodfellow et al., 2016] for details. Suppose we have an input with dimensions $W_{\text{in}} \times H_{\text{in}} \times D_{\text{in}}$, where W denotes the width, H denotes the height and D denotes the depth. Then the dimensions of the output volume $W_{\text{out}} \times H_{\text{out}} \times D_{\text{out}}$ are given by

$$\begin{aligned} W_{\text{out}} &= \frac{W_{\text{in}} - F + 2P}{S} + 1, \\ H_{\text{out}} &= \frac{H_{\text{in}} - F + 2P}{S} + 1, \\ D_{\text{out}} &= K. \end{aligned}$$

See [Dumoulin and Visin, 2016] for an exhaustive overview of the arithmetics of convolutional layers. Note that we are not free to set the hyperparameters arbitrarily. The combinations of values must be constrained to produce integer-valued dimensions. Most deep learning libraries will warn the user of invalid hyperparameter choices.

Example 3.2. Volume of output after convolution. Assume the input is a colour image of size $(32 \times 32 \times 3)$. Convolving the image with $K = 12$ filters of size $F = 5$ where the stride $S = 1$ and zero-padding $P = 0$ will result in a output volume of $(27 \times 27 \times 12)$. Assuming that our kernels are trained to detect some meaningful features, we can think of the output volume as twelve 2-D feature maps. Each feature map consists of 729 neurons, arranged in a (27×27) grid indicating the presence of a feature learned by it's corresponding filter. ⌟

3.7.3 Pooling

A typical convolutional layer will first apply the convolution operation to an input followed by a non-linear activation function, resulting in a volume of K 2-D feature maps. The final step of a convolutional layer is the pooling operation. Pooling layers operate on each feature map separately by replacing multiple values in non-overlapping spatial neighbourhoods with a single value, thus reducing the resolution. This brings down the number of network parameters and helps to control overfitting. Furthermore, pooling provides some robustness to translations in the input, which is useful if the primary goal is to detect a feature rather than pinpointing its exact position.

Typically, the max-pooling function is used. A window of size F is shifted over the image with a stride S (much like a filter during convolution), returning the maximum activation in the overlapping region of the feature map and the pooling window. The

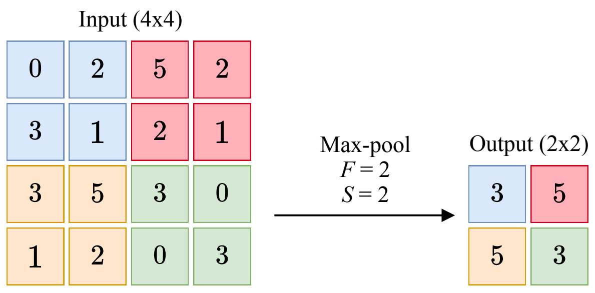


Fig. 3.6: Max-pooling downsamples a feature map by reducing non-overlapping spatial neighbourhood to the maximum activation in that neighbourhood. The illustration above shows an example of a pooling window of size $F = 2$ with stride $S = 2$.

output dimensions of the pooling layer are controlled by two hyperparameters: the window size F and the stride S. Typically $F = 2$ and $S = 2$. See fig. 3.6 for a simple example. An example of an alternative to max-pooling is global average pooling [Lin et al., 2013].

Some researchers argue that the pooling operation throws away an unnecessarily large amount of useful information, and suggest downsampling by convolution with larger strides instead [Springenberg et al., 2014].

3.7.4 LeNet: A Simple Convolutional Neural Network

In summary, the main building block of a CNN is the convolutional layer. First, the convolution operation computes a weighted sum of inputs which are passed through a non-linear activation function. This results in a 2-D feature map of activations which are pooled to produce a lower resolution feature map.

One of the earliest and most famous CNNs is the LeNet family of models [LeCun et al., 1990], developed by Yann LeCun and colleagues to identify handwritten digits. We will examine the LeNet architecture in detail for two reasons: (1) It is simple to understand and conveys the idea behind CNNs without the complex architectural quirks of contemporary networks, and (2) we will be using a variant of LeNet in the empirical analysis in chapter 5.

Fig. 3.7 shows the general architecture of a LeNet model applied to a colour image input. Two convolutional layers are stacked on top of each other. The number of feature maps depends on the number of filters F specified in each layer. The final volume of feature maps is unrolled into a long feature vector which is fed into a fully connected network, which we introduced in section 3.2. The output layer typically

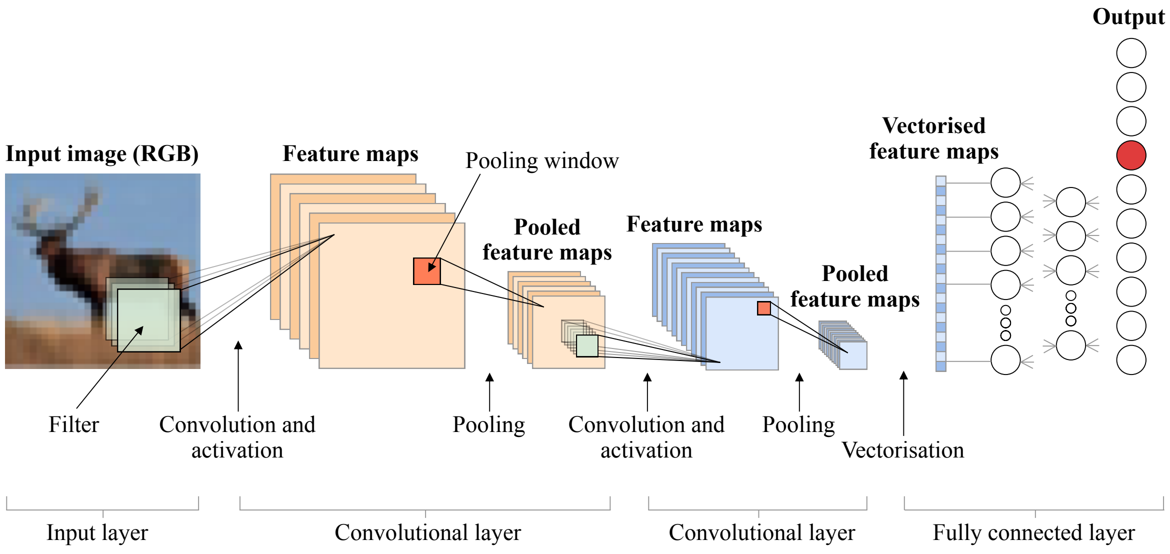


Fig. 3.7: An example of the LeNet architecture. The input is followed by two convolutional layers. The output of the last convolutional layer is unrolled into a vector of activations and fed into a fully connected network. Finally the neuron with the largest activation in the output layer will correspond to the network's prediction. The figure above is an adaptation of the illustration on p. 965 in [Gonzalez and Woods, 2017].

applies a softmax activation function (see section 3.3), and the output neuron with the largest activation determines the class prediction. In fig. 3.7 the input layer is an example from the CIFAR-10 data set [Krizhevsky and Hinton, 2009], which consists of 60.000 images belonging to 10 different classes (see chapter 6 for details).

Chapter 4

Modelling Uncertainty

In the frequentist approach to statistics, the true, unobservable value of some parameter vector $\boldsymbol{w}$ is assumed fixed and unknown and the estimate $\hat{\boldsymbol{w}}$ is based on a random sample of data points. This means that the values of $\hat{\boldsymbol{w}}$ will vary from sample to sample, making $\hat{\boldsymbol{w}}$ itself a random variable. Uncertainty about the point estimate of $\boldsymbol{w}$ is reflected in the variance of $\hat{\boldsymbol{w}}$. The variance indicates how much $\hat{\boldsymbol{w}}$ is expected to change with respect to a different sampling of data.

Bayesian statistics takes another approach: Statements about $\boldsymbol{w}$ are made in terms of probability distributions that are conditioned on observed data, denoted by $p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})$. Uncertainty about $\boldsymbol{w}$ is thus expressed in terms of probability, which is why the Bayesian framework is often referred to as *probabilistic* modelling. In this chapter we will explore probabilistic modelling and how it can be applied to deep learning.

In section 4.1 we give a brief introduction to the basics of Bayesian statistics. In section 4.2 we focus on approximate Bayesian inference (4.2.1) and Gaussian process models (4.2.2), which will be relevant for later discussions. In section 4.3 we briefly review Monte Carlo estimation. In section 4.4 we introduce Bayesian neural networks and summarise recent research into dropout training as an approximation to Bayesian inference.

In this chapter we primarily rely on the following sources and the references therein:

- Chapters 1 and 21 in [Gelman et al., 2013] (Bayesian inference and Gaussian processes).
- Chapter 2 in [Rasmussen and Williams, 2005] (Gaussian processes for regression).

- Chapter 15 in [Murphy, 2012] (Gaussian processes).
- Chapters 3, 5 and 17 in [Goodfellow et al., 2016] (probability, Bayesian linear regression, variational inference).
- Chapters 2 and 3 in [Gal, 2016] (Bayesian inference, Bayesian deep learning, variational inference).
- Chapter 6 in [Bishop, 2006] (Gaussian processes).

4.1 A Brief Introduction to Bayesian Statistics

Bayes' rule states that the conditional probability of an event A given event B can be expressed by

$$p(A|B) = \frac{p(A)p(B|A)}{p(B)}.$$

More formally, let S be the union of k mutually exclusive events $A_1, A_2, ..., A_k$. Then, if $B \in S$ and $p(A_j) > 0$ and $p(B) > 0$, Bayes' theorem states that

$$p(A_j|B) = \frac{p(A_j)p(B|A_j)}{\sum_{i=1}^{k} p(A_i)p(B|A_i)},$$

where the sum is replaced by an integral in the continuous case. Bayes' theorem can also be applied to parameter estimation. Information or assumptions about $\boldsymbol{w}$ are incorporated into a prior distribution over $\boldsymbol{w}$, denoted $p(\boldsymbol{w})$. Strong priors are more informative, meaning $p(\boldsymbol{w})$ is more concentrated around some values of $\boldsymbol{w}$. On the other hand, having no particular beliefs about the parameters should result in a more uniform, or uninformative, prior. After observing $(\boldsymbol{X}, \boldsymbol{y})$, we update the prior to reflect which values of $\boldsymbol{w}$ are most likely. This is called the posterior distribution of $\boldsymbol{w}$, and is given by

$$p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{w})p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})}{p(\boldsymbol{y}|\boldsymbol{X})}. \tag{4.1}$$

Here $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$ is the model, represented by the likelihood function (see section 2.3.1). The normalising constant $p(\boldsymbol{y}|\boldsymbol{X})$ is called the marginal likelihood. Some textbooks refer to this as the model evidence. The marginal likelihood is obtained by integrating out $\boldsymbol{w}$:

$$p(\boldsymbol{y}|\boldsymbol{X}) = \int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w})\, d\boldsymbol{w}.$$

Note the slight abuse of notation in eq. 4.1: We use $p(\cdot)$ to denote the distributions of the posterior, prior, likelihood and marginal likelihood, but this does not mean that they share the same distribution.

Let $\boldsymbol{x}^*$ denote a new observation with unknown target value y^*. The Bayesian approach makes inferences based on the full distribution of y^*. This is called the posterior predictive distribution, given by

$$p(y^*|\boldsymbol{x}^*, \boldsymbol{X}, \boldsymbol{y}) = \int p(y^*|\boldsymbol{x}^*, \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) \, d\boldsymbol{w}. \tag{4.2}$$

Eq. 4.2 can be viewed as an ensemble of models with different settings of $\boldsymbol{w}$, weighted by $p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})$. Thus the most likely parameter values will contribute the most to the probability of y^* taking on a particular value. Making statements about parameter values and predictions in terms of probability highlights one of the key features of the Bayesian approach: It gives an intuitive and principled way of expressing uncertainty.

4.2 Bayesian modelling

To see how Bayes' theorem can be applied in a machine learning context, consider the following example:

Example 4.1 (Bayesian linear regression). As before, assume $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and $\boldsymbol{y} \in \mathbb{R}^N$. Let $y_i|\boldsymbol{x}_i, \boldsymbol{w} \sim \mathcal{N}(\boldsymbol{x}_i^\top \boldsymbol{w}, \sigma_\epsilon^2)$ and suppose we place a Gaussian prior over the weights, such that $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \tau^2 \mathbf{I})$. Assume that σ_ϵ^2 and τ^2 are known. By applying Bayes' theorem we can figure out which setting of $\boldsymbol{w}$ best describes the observed data:

$$\begin{aligned} p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) &= \frac{p(\boldsymbol{w}) p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})}{p(\boldsymbol{y}|\boldsymbol{X})} \\ &\propto p(\boldsymbol{w}) p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) \\ &= p(\boldsymbol{w}) \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}). \end{aligned}$$

The symbol $\propto$ means that the expression on the left-hand side is proportional to the expression on the right-hand side. Computing the log-likelihood gives

$$\begin{aligned}
\log p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y}) &\propto \log p(\boldsymbol{w}) + \sum_{i=1}^{N} \log p(y_i|\boldsymbol{x}_i,\boldsymbol{w}) \\
&= \log \mathcal{N}(\mathbf{0}, \tau^2\mathbf{I}) + \sum_{i=1}^{N} \log \mathcal{N}(\boldsymbol{x}_i^\intercal \boldsymbol{w}, \sigma_\epsilon^2) \\
&\propto -\frac{\boldsymbol{w}^\intercal\boldsymbol{w}}{2\tau^2} - \frac{(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^\intercal(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})}{2\sigma_\epsilon^2} \\
&= -\frac{1}{2\sigma_\epsilon^2}\left((\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^\intercal(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}) + \frac{\sigma_\epsilon^2}{\tau^2}\boldsymbol{w}^\intercal\boldsymbol{w}\right).
\end{aligned}$$

We recognise the parenthesised terms as the regularised cost of the ridge estimate (see example 2.6) with $\lambda = \sigma_\epsilon^2/\tau^2$. In other words, minimising the negative log-likelihood is equivalent to ridge regression where the regularisation strength λ depends on the variance of y_i and the precision[1] of $\boldsymbol{w}$. The regularising effect of the prior distribution is considered one of the core features of Bayesian inference. The resulting estimates of $\boldsymbol{w}$ are called the maximum a posteriori (MAP) estimates. ⌟

In the above example we ignored the marginal likelihood $p(\boldsymbol{y}|\boldsymbol{X})$ because the MAP estimate of $\boldsymbol{w}$ is unaffected by it. In this sense we are still providing a point estimate of $\boldsymbol{w}$. A fully Bayesian treatment requires that we integrate over the posterior distribution of $\boldsymbol{w}$, which requires that we evaluate $p(\boldsymbol{y}|\boldsymbol{X})$. Unfortunately, in many practical applications $p(\boldsymbol{y}|\boldsymbol{X})$ is intractable, making $p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y})$ impossible to evaluate analytically. In such cases approximations must be made.

4.2.1 Variational Inference

We can approximate $p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y})$ using variational inference. The main idea is that we can define a probability distribution $q_{\boldsymbol{\phi}}(\boldsymbol{w})$ that is easy to evaluate, and use $q_{\boldsymbol{\phi}}(\boldsymbol{w})$ to approximate $p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y})$. $q_{\boldsymbol{\phi}}(\boldsymbol{w})$ is parameterised by $\boldsymbol{\phi}$, which are called the variational parameters. Finding the distribution $q_{\boldsymbol{\phi}}(\boldsymbol{w})$ that best approximates $p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y})$ amounts to finding the best setting of variational parameters, turning the marginalisation problem in eq. 4.2 into an optimisation problem.

[1]Model precision is a term that is frequently used in Bayesian inference when referring to the inverse of the variance of a normal distribution.

Specifically, let $\mathcal{Q}$ denote all possible approximating distributions $q_\phi(\boldsymbol{w})$. The best approximating density $q^*_\phi(\boldsymbol{w}) \in \mathcal{Q}$ is the one that minimises the KL divergence:

$$q^*_\phi(\boldsymbol{w}) = \arg\min_{\boldsymbol{\phi}} \mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})],$$

where

$$\mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})] = \int q_\phi(\boldsymbol{w}) \log \frac{q_\phi(\boldsymbol{w})}{p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})} \, d\boldsymbol{w}. \tag{4.3}$$

As mentioned in section 2.6, the KL divergence is a measure of similarity between distributions. By substituting eq. 4.1 into eq. 4.3, it can be shown[2] that

$$\begin{aligned}\mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})] = {} & \mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w})] \\ & - \mathbb{E}_{q_\phi(\boldsymbol{w})}[\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})] + \log p(\boldsymbol{y}|\boldsymbol{X}).\end{aligned} \tag{4.4}$$

The KL divergence still depends on the intractable marginal likelihood $p(\boldsymbol{y}|\boldsymbol{X})$. We must therefore minimise an alternative objective. Since $\mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})] \geq 0$ (see section 2.6), we can rearrange the terms in eq. 4.4 to obtain the inequality

$$\mathbb{E}_{q_\phi(\boldsymbol{w})}[\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})] - \mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w})] \leq \log p(\boldsymbol{y}|\boldsymbol{X}).$$

The expression on the left-hand side of the inequality is called the evidence lower bound (ELBO). Maximising the ELBO with respect to $q_\phi(\boldsymbol{w})$ is equivalent to minimising eq. 4.4, resulting in the variational objective

$$C_{\mathrm{VI}} = \mathbb{E}_{q_\phi(\boldsymbol{w})}[\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})] - \mathrm{KL}[q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w})], \tag{4.5}$$

such that

$$q^*_\phi(\boldsymbol{w}) = \arg\max_{\boldsymbol{\phi}} C_{\mathrm{VI}}. \tag{4.6}$$

Eq. 4.6 encourages the variational parameters $\boldsymbol{\phi}$ to explain the data well (by maximising the likelihood) while simultaneously ensuring that $q^*_\phi(\boldsymbol{w})$ is as close to the prior $p(\boldsymbol{w})$ as possible (by minimising the KL term), thus reflecting the balance between the observed data and prior distribution. This allows us to define an approximate

[2]See section A.2 in appendix A

posterior predictive distribution for y^*:

$$q(y^*|\boldsymbol{x}^*) \approx \int p(y^*|\boldsymbol{x}^*, \boldsymbol{w}) q^*_\phi(\boldsymbol{w}) d\boldsymbol{w}.$$

The reader is referred to [Blei et al., 2017] for an introduction to variational inference.

4.2.2 Gaussian Processes

In the previous sections we inferred a distribution over model parameters by placing a prior over the weight space. An alternative (but equivalent) view is that we are placing a prior distribution over the space of possible functions that have generated the data (see chapter 2.2 in [Rasmussen and Williams, 2005] for details).

The Gaussian process (GP) is a non-parametric method for performing Bayesian inference over functions. In the GP view we place a prior distribution over the function space, and the posterior

$$p(f|\boldsymbol{X}, \boldsymbol{y}) \propto p(f)p(\boldsymbol{y}|\boldsymbol{X}, f)$$

reflects the functions most likely to have generated the data. The posterior predictive distribution is then given by

$$p(y^*|\boldsymbol{x}^*, \boldsymbol{X}, f) = \int p(y^*|\boldsymbol{x}^*, f)p(f|\boldsymbol{X}, \boldsymbol{y})\, df.$$

[Rasmussen and Williams, 2005] is considered the definitive reference for GPs in machine learning, but we follow [Murphy, 2012] closely in this section due to notational convenience.

It is very hard to imagine a distribution over all possible functions that could have generated the data. The GP approach defines a distribution over the function values. Specifically, given a finite, arbitrary set of points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N\}$, the function values $\mathbf{f} = \{f(\boldsymbol{x}_1), f(\boldsymbol{x}_2), ..., f(\boldsymbol{x}_N)\}$ are assumed to be jointly Gaussian. We can state this assumption more compactly:

$$f(\boldsymbol{x}) \sim \mathcal{GP}(m(\boldsymbol{x}), \kappa(\boldsymbol{x}, \boldsymbol{x'})),$$

where

$$m(\boldsymbol{x}) = \mathbb{E}[f(\boldsymbol{x})],$$

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \mathbb{E}[(f(\boldsymbol{x}) - m(\boldsymbol{x}))(f(\boldsymbol{x}') - m(\boldsymbol{x}'))^{\mathsf{T}}].$$

The function $m(\boldsymbol{x})$ is called the mean function and $\kappa(\boldsymbol{x}, \boldsymbol{x}')$ is a positive semi-definite covariance function, called a kernel[3]. The kernel specifies the covariance of function values. Intuitively, if $\boldsymbol{x}$ and $\boldsymbol{x}'$ are similar, then the kernel should output similar values. Thus, for any finite set of points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N\}$, the GP defines

$$\begin{aligned}
&p(\mathbf{f}|\boldsymbol{X}) \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{K}),\\
&\boldsymbol{\mu} = (m(\boldsymbol{x}_1), m(\boldsymbol{x}_2), ..., m(\boldsymbol{x}_N))^{\mathsf{T}},\\
&\mathbf{K}_{ij} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j), \quad i = 1, \dots, N, j = 1, \dots, N.
\end{aligned}$$

Example 4.2 (GPs for regression). The following example is taken from chapter 15 in [Murphy, 2012]. Suppose we want to estimate $y = f(\boldsymbol{x}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. Given $\boldsymbol{X} \in \mathbb{R}^{N\times D}$ and $\boldsymbol{y} \in \mathbb{R}^N$, we want to predict $\mathbf{f}^*$ for some new observations $\boldsymbol{X}^* \in \mathbb{R}^{K\times D}$. For simplicity we assume a mean of $\mathbf{0}$. By definition, the joint distribution of $\boldsymbol{y}$ and $\mathbf{f}^*$ is given by

$$\begin{bmatrix} \boldsymbol{y} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K}_{\boldsymbol{y}} & \mathbf{K}^* \\ \mathbf{K}^{*\mathsf{T}} & \mathbf{K}^{**} \end{bmatrix}\right),$$

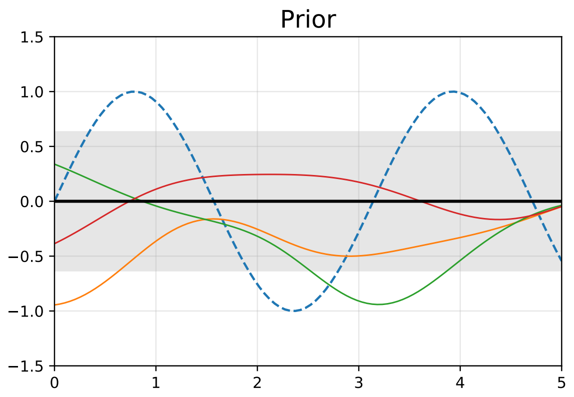
where

$$\begin{aligned}
&\mathbf{K}_{\boldsymbol{y}} = \kappa(\boldsymbol{X}, \boldsymbol{X}) + \sigma_\epsilon^2 \mathbf{I} \text{ is } N \times N,\\
&\mathbf{K}^* = \kappa(\boldsymbol{X}, \boldsymbol{X}^*) \text{ is } N \times K,\\
&\mathbf{K}^{**} = \kappa(\boldsymbol{X}^*, \boldsymbol{X}^*) \text{ is } K \times K.
\end{aligned}$$

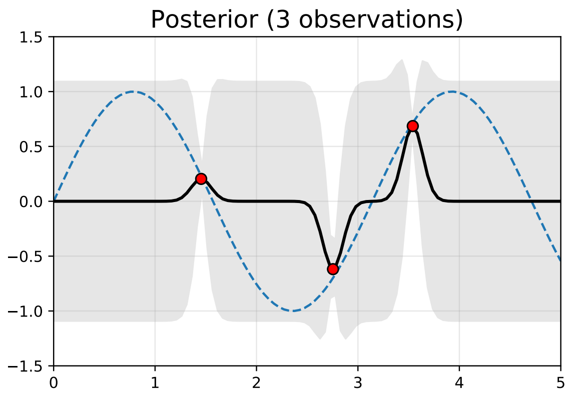
The mean $\boldsymbol{\mu}^*$ and variance $\boldsymbol{\Sigma}^*$ of the posterior predictive distribution are obtained by applying the rules of conditioning[4] for the multivariate Gaussian distribution:

$$\begin{aligned}
&p(\mathbf{f}^*|\boldsymbol{X}^*, \boldsymbol{X}, \boldsymbol{y}) \sim \mathcal{N}(\boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*),\\
&\boldsymbol{\mu}^* = \mathbf{K}^{*\mathsf{T}} \mathbf{K}_{\boldsymbol{y}}^{-1} \boldsymbol{y},\\
&\boldsymbol{\Sigma}^* = \mathbf{K}^{**} - \mathbf{K}^{*\mathsf{T}} \mathbf{K}_{\boldsymbol{y}}^{-1} \mathbf{K}^*.
\end{aligned}$$

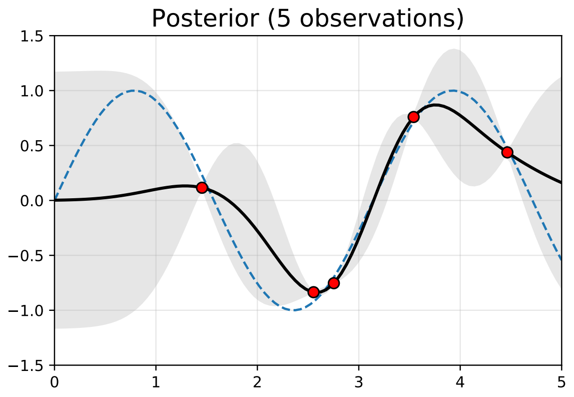
[3]Not to be confused with the kernel in convolutional neural networks.
[4]See appendix A, section A.3.



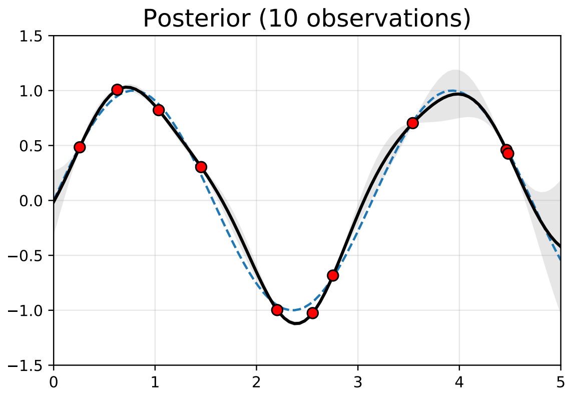
(a)



(b)



(c)



(d)

Fig. 4.1: An example of a Gaussian process (GP) defined by a radial basis function kernel and a mean of zero (refer to [Rasmussen and Williams, 2005] or chapter 15 in [Murphy, 2012] for details on different kernels and their effects on function estimation). The true function $f(x) = \sin(2x)$ is given by the dotted blue line. The mean of the GP is given by the solid black line. Fig. 4.1a shows three functions sampled from the prior. Figs. 4.1b–4.1d show the predictive mean $\boldsymbol{\mu}^*$ (in black) for varying numbers of observations $y_i = f(x_i) + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, 0.05)$. The grey band shows the uncertainty associated with $\boldsymbol{\mu}^*$. We see that as the number of observations increase, the uncertainty associated with points that are far from the observed data decreases.

Due to the matrix inversions required to compute $\boldsymbol{\Sigma}^*$, GPs often fail to scale to very large data sets. In these cases approximations such as variational inference must be used. Figure 4.1 shows an example of a Gaussian process. ⌟

It is possible to stack GPs in a hierarchical manner such that the input to one GP is itself determined by another GP. These are called deep GPs [Damianou and Lawrence, 2013]. Furthermore, there exists a connection between GPs and neural networks: By choosing appropriate priors over the network weights, an MLP with a single hidden layer will converge to a GP with a specific covariance function in the limit of infinitely many hidden neurons [Neal, 1996]. GPs are also used for Bayesian optimisation [Snoek et al., 2012], which is the process of automatically tuning hyperparameters of machine learning algorithms.

4.3 Basics of Monte Carlo Sampling

Monte Carlo (MC) estimation is a random sampling method that allows us to compute estimates of sums and integrals. MC estimation is particularly helpful in problems where the desired sum or integral cannot be computed analytically. In these cases, we view the sum or integral as an expectation under some probability distribution $p(\boldsymbol{x})$

$$\mathbb{E}[f(\boldsymbol{x})] = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x},$$

which can be approximated by evaluating f on samples from $p(\boldsymbol{x})$ and computing the empirical average

$$\mathbb{E}[f(\boldsymbol{x})] \approx \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i), \quad \boldsymbol{x}_i \sim p(\boldsymbol{x}). \tag{4.7}$$

It can be shown that the MC estimate in eq. 4.7 is an unbiased estimator of $\mathbb{E}[f(\boldsymbol{x})]$ and that the MC estimate converges to the true expectation as N gets large, provided that the samples are i.i.d. and $\mathrm{Var}[f(\boldsymbol{x}_i)] < \infty$. The reader is referred to [Goodfellow et al., 2016] and [Gelman et al., 2013] for details. MC estimates allow us to obtain uncertainty estimates by computing the empirical variance of $f(\boldsymbol{x}_i)$ and dividing by the number of samples.

4.4 Bayesian Deep Learning

The neural networks introduced in chapter 3 are limited by their inability to provide principled uncertainty estimates associated with their predictions. However, it is possible to view neural networks in a probabilistic framework by placing a prior distribution over the network parameters. The result is a so-called Bayesian neural network (BNN) [MacKay, 1992; Neal, 1996], from which we can obtain theoretically grounded quantifications of uncertainty. Unfortunately, many BNNs have turned out to be impractical to work with due to their inability to scale to large data sets. Refer to section 2.2 in [Gal, 2016] for a historical overview of BNNs. In recent work, Gal and colleagues try to alleviate this impracticality by suggesting a novel approach to Bayesian inference in neural networks [Gal and Ghahramani, 2016].

4.4.1 Dropout as Variational Inference

The full derivation of this result is very technical and well beyond the scope of this thesis, but we will briefly sketch the general idea in this section. We follow the notation in [Gal and Ghahramani, 2016]. Note that in this notation the layers are indexed by i and the layer numbering starts at $i = 1$ as opposed to $i = 2$. For a complete and detailed exposition of the derivation, the reader is referred to [Gal, 2016].

The regularised cost function (eq. 2.8) of a dropout neural network (eq. 3.15) is given by

$$C_{\text{dropout}} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) + \lambda\sum_{i=1}^{L}(\|\boldsymbol{W}_i\|^2 + \|\boldsymbol{b}_i\|^2). \tag{4.8}$$

Starting in a regression setting, Gal and colleagues show that minimising C_{dropout} is equivalent to performing variational inference between an approximating distribution $q_{\boldsymbol{M}}(\boldsymbol{\omega})$ and the posterior predictive distribution of a deep GP. Here $\boldsymbol{\omega} = \{\boldsymbol{W}_i\}_{i=1}^{L}$ denotes the collection of weight matrices associated with the covariance function at different layers of the deep GP, and the subscript $\boldsymbol{M}$ denotes the variational parameters.

After a series of reparameterisations and factorisations, the posterior predictive distribution is integrated over the covariance parameters $\boldsymbol{\omega}$. The approximating distribution $q_{\boldsymbol{M}_i}(\boldsymbol{W}_i)$ of the covariance parameters in the i'th layer of the deep GP is defined by

$$\begin{aligned}
&\boldsymbol{W}_i = \boldsymbol{M}_i \times \text{diag}\big(\boldsymbol{r}_i\big),\\
&\boldsymbol{r}_i = [r_{ij}]_{j=1}^{K_{i-1}},\\
&r_{ij} \sim \text{Bernoulli}(p_i),\\
&i = 1, ..., L, \quad j = 1, ..., K_{i-1}.
\end{aligned} \tag{4.9}$$

The authors call eq. 4.9 a Bernoulli approximating variational distribution. By evaluating the ELBO (eq. 4.5) the authors are able to retrieve a scaled version of the cost function in eq. 4.8. Gal concludes that approximating a deep GP with a Bernoulli approximating variational distribution results in the same optimal parameters as dropout training.

Gal goes on to argue that sampling $\boldsymbol{W}_i$ from $q_{\boldsymbol{M}_i}(\boldsymbol{W}_i)$ is identical to performing dropout in layer i of a neural network. Suppose we now construct a BNN, and approximate the posterior predictive distribution using a Bernoulli approximating variational distribution over the network parameters. Then, Gal argues, the optimal parameters of the BNN will be the same as those in a dropout network with the same structure. According to Gal, this means that a dropout network can be viewed as approximate inference in a BNN, thus granting access to the principled uncertainty estimates associated with Bayesian modelling.

In practice, sampling from the posterior of a BNN is equivalent to leaving dropout *on* when making predictions. The input $\boldsymbol{x}^*$ is propagated through the network T times. Each $t \in T$ is called a stochastic forward pass, and each t results in a new sampling of weights (due to dropout). The MC estimate (eq. 4.7) of the prediction $\boldsymbol{y}^*$ is given by

$$\mathbb{E}(\boldsymbol{y}^*) \approx \frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t), \quad \hat{\boldsymbol{\omega}}_t \sim q_{\boldsymbol{M}}(\boldsymbol{\omega}), \tag{4.10}$$

where $\hat{\boldsymbol{\omega}}_t$ is shorthand notation for the collection of sampled weights from the Bernoulli approximating variational distribution, denoted by $q_{\boldsymbol{M}}(\boldsymbol{\omega})$. The model output for the t'th stochastic forward pass is denoted $\hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)$. The authors refer to the aggregated estimate in 4.10 as MC dropout.

By establishing a connection to GPs, the authors are also able to derive the predictive uncertainty associated with the MC dropout estimate, given by

$$\begin{aligned}\mathrm{Var}(\boldsymbol{y}^*) \approx \tau^{-1}\mathbf{I} + \frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)^{\mathsf{T}} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) \\ - \left(\frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)\right)^{\mathsf{T}} \left(\frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)\right),\end{aligned} \tag{4.11}$$

where τ is a precision term.

These results are applicable to both regression and classification settings. For classification, we can use the softmax function (see section 3.3) to output a vector of class probabilities

$$p(\hat{\boldsymbol{y}}|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) = (p(\hat{y} = 1|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t), \ldots, p(\hat{y} = K|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)),$$

where the dimensionality K corresponds to the number of classes. The k'th element corresponds to the probability that the input belongs to class k, given the weights sampled in the t'th stochastic forward pass.

The MC dropout estimate of the vector of softmax probabilities is denoted

$$\begin{aligned}\hat{\boldsymbol{\mu}} &= (\hat{\mu}_1, \ldots, \hat{\mu}_K) \\ &= \left(\frac{1}{T} \sum_{t=1}^{T} p(\hat{y} = 1 | \boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t), \ldots, \frac{1}{T} \sum_{t=1}^{T} p(\hat{y} = K | \boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) \right) \\ &= \frac{1}{T} \sum_{t=1}^{T} p(\hat{\boldsymbol{y}} | \boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t).\end{aligned}$$

Finally, the predicted class corresponds to the largest element in $\hat{\boldsymbol{\mu}}$, such that

$$\hat{\mu}_{\text{pred}} = \max_{\hat{\mu}_k} \hat{\boldsymbol{\mu}}, \quad k = 1, \ldots, K.$$

In classification tasks the predictive uncertainty given by eq. 4.11 does not hold. Gal argues that other methods are necessary, suggesting predictive entropy (PE), mutual information (MI) and variation-ratios (VR) as alternatives (see section 3.3.1 in [Gal, 2016] for details). PE and MI are rooted in information theory (see section 2.6). Assuming a softmax output, the PE measures the uncertainty associated with the vector of class probabilities, and can be approximated by

$$\text{PE}(\hat{\boldsymbol{\mu}}) = -\sum_{k} \mu_k \log \mu_k. \tag{4.12}$$

As stated in section 2.6, the MI is a measure of information gain. In this context, the MI measures the information gained about the model parameters if we were to receive a label y for some input $\boldsymbol{x}^*$. In a classification setting the MI can be approximated by

$$\text{MI}(\hat{\boldsymbol{\mu}}) = \text{PE}(\hat{\boldsymbol{\mu}}) + \frac{1}{T} \sum_{t} \sum_{k} p(\hat{y} = k | \boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) \log p(\hat{y} = k | \boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t). \tag{4.13}$$

The VR measures the spread of the distribution around the mode, and can be approximated by

$$\begin{aligned} \mathrm{VR}(\hat{\boldsymbol{\mu}}) &= 1 - \frac{f_{\boldsymbol{x}^*}}{T}, \\ f_{\boldsymbol{x}^*} &= \sum_t \mathbf{1}(\hat{y}_t = k^*), \\ k^* &= \underset{k=1,\dots,K}{\arg\max} \sum_t \mathbf{1}(\hat{y}_t = k), \end{aligned} \tag{4.14}$$

where $\hat{y}_t$ denotes the predicted class label for the t'th stochastic forward pass.

4.4.2 Bayesian Convolutional Neural Networks

The results in the previous section can be applied to CNNs also. However, many CNNs typically apply dropout only in the last, fully connected layer of the network. This is because dropping the weights in the convolutional kernels has in many cases been shown to negatively impact the predictive performance of a standard dropout CNN [Gal and Ghahramani, 2015]. In Gal's initial development of approximate inference in Bayesian deep learning, the convolutional kernels are viewed as deterministic transformations, and predictions and uncertainty estimates can still be obtained following the results of section 4.4.1.

By casting dropout training as Bernoulli approximate variational inference for Bayesian NNs directly [Gal and Ghahramani, 2015], the authors develop a framework for approximate inference over the convolutional kernels in a CNN. This is accomplished by adding a dropout layer after every convolutional layer. As before, dropout remains on during prediction and the results are averaged to give an MC dropout estimate. These so-called fully Bayesian CNNs (BCNNs) are shown empirically to outperform standard dropout CNNs in terms of predictive accuracy [Gal and Ghahramani, 2015]. However, this extension comes at a cost: The GP interpretation is now lost, and Gal does not discuss how to obtain uncertainty estimates from BCNNs.

4.4.3 Practical Applications of MC Dropout

Several researchers in the field of computer vision have extrapolated on the results in section 4.4.1 to obtain uncertainty estimates in image classification tasks [Feinman et al., 2017; Leibig et al., 2017]. These researchers use BCNNs and ad hoc approximations of

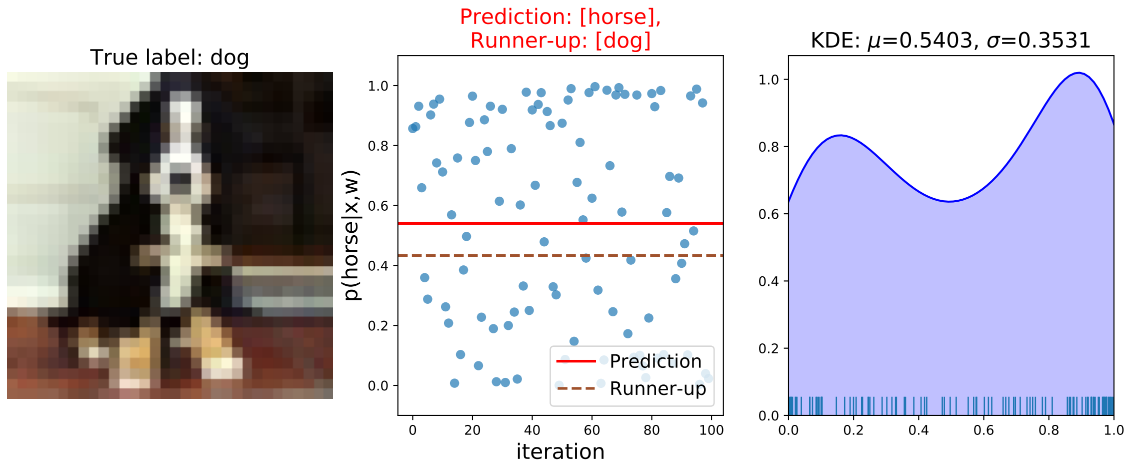


Fig. 4.2: The above plot shows a practical example of MC dropout, taken from chapter 5. The leftmost plot shows the image with the corresponding ground truth label. The middle plot shows the softmax output of the predicted class for each stochastic forward pass. The MC dropout estimate is given by the solid red line. The rightmost plot shows a kernel density estimate (KDE) of the distribution of the approximated uncertainty associated with the MC dropout estimate.

the predictive uncertainty given by eq. 4.11 with successful results, even though Gal warns that the GP interpretation is lost for BCNNs and that eq. 4.11 is inappropriate in a classification setting. We will briefly summarise this research.

In a recently published paper, Leibig and colleagues use a BCNN to capture uncertainty estimates associated with classifying diabetic retinopathy[5] in fundus[6] images [Leibig et al., 2017]. The researchers show that automatic disease detection can be improved by making uncertainty-informed referrals to human experts, thus obtaining state-of-the-art results. The key idea is that this human-machine interaction will lead to better diagnostic accuracy than either could produce individually.

The problem of disease detection is reduced to a binary classification task, where Leibig et. al. use a softmax activation to output the probabilities of the two classes (0: Healthy, 1: Diseased). Instead of predicting the class associated with the largest element of $\hat{\boldsymbol{\mu}}$, the authors define

$$\hat{\mu}_{\text{diseased}} = \frac{1}{T}\sum_{t=1}^{T} p(\hat{y} = 1|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t),$$

[5]Diabetic retinopathy is an eye disease resulting from complications following diabetes which can lead to blindness.

[6]The fundus is the interior surface of the eye opposite to the lens which includes, among other things, the retina.

and classify a patient as healthy if $\mu_{\text{diseased}} < 0.5$. The authors use the empirical standard deviation as a proxy for predictive uncertainty[7]:

$$\hat{\sigma}_{\text{pred}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left[p(\hat{y}=1|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_{\text{diseased}}\right]^2}.$$

Leibig et. al. argue that the width of the empirical distribution of T samples of $p(\hat{y}=1|\boldsymbol{x}, \hat{\boldsymbol{\omega}}_t)$ should indicate the network's confidence in its prediction.

In another recent paper, Feinman and colleagues explore how ad hoc uncertainty estimates obtained from BCNNs can be used to detect adversarial examples in image classification problems [Feinman et al., 2017]. An adversarial example is an image where the pixel values are perturbed in a way that is imperceptible to humans, but causes great confusion in a neural network. To this end Feinman et. al. develop the following approximation of predictive uncertainty:

$$\hat{\sigma}^2 = \frac{1}{K}\sum_{k=1}^{K}\hat{\sigma}_k^2,$$

where K is the number of classes and

$$\hat{\sigma}_k^2 = \frac{1}{T}\sum_{t=1}^{T}[p(\hat{y}=k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_k]^2, \quad k = 1, ..., K.$$

In other words, $\hat{\sigma}^2$ returns a single, scalar-valued measure of uncertainty for each observation by averaging the mean squared prediction error of each class. The authors found that their ad hoc uncertainty estimate can be used to detect adversarial examples.

Recent work by Smith and Gal suggests a connection between $\hat{\sigma}^2$ and the MI of the predictions [Smith and Gal, 2018]. In brief, the authors show that $\hat{\sigma}^2$ is identical (up

[7] As of 10.10.18, the published version of [Leibig et al., 2017] (available at `https://www.nature.com/articles/s41598-017-17876-z`) states that the empirical standard deviation is given by $\hat{\sigma}_{\text{pred}} = \frac{1}{T-1}\sqrt{\sum_{t=1}^{T}\left[p(\hat{y}=1|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_{\text{diseased}}\right]^2}$. We assume that this is a typographical error, since the $\frac{1}{T-1}$ term should be included under the square root. A review of the associated GitHub repository (`https://github.com/chleibig/disease-detection`) also seems to indicate that their definition is a misprint. Furthermore, it is not clear from the code that the authors actually use the unbiased estimator of the empirical standard deviation. As far as we can tell, the authors use the NumPy implementation (see the documentation at `https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html` for details) of the empirical standard deviation, such that $\hat{\sigma}_{\text{pred}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left[p(\hat{y}=1|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_{\text{diseased}}\right]^2}$.

to a multiplicative constant) to the leading term of a Taylor expansion of the MI (refer to [Smith and Gal, 2018] for details on the derivation), thus offering an explanation to why $\hat{\sigma}^2$ seems to work reasonably well in practice. The empirical properties of these ad hoc uncertainty approximations is the focus of part 2.

Part II

Experiments

Chapter 5

Approximated Predictive Uncertainty in a Multi-Class Setting

In part 1 we reviewed the basic theory and ideas that form the basis for uncertainty estimation in dropout neural networks. To reiterate, Gal and Ghahramani establish a connection between dropout neural networks and a well-known Bayesian model called the Gaussian process (GP). They show that dropout training can be viewed as Bernoulli approximate variational inference over the weights of a Bayesian neural network (BNN) [Gal and Ghahramani, 2016], which allows researchers to obtain principled uncertainty estimates from dropout networks by leaving dropout on at prediction time. The predictive uncertainty is given by

$$\begin{aligned}\mathrm{Var}(\boldsymbol{y}^*) \approx \tau^{-1}\mathbf{I} &+ \frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)^\intercal \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) \\ &- \left(\frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)\right)^\intercal \left(\frac{1}{T}\sum_{t=1}^{T} \hat{\boldsymbol{y}}(\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)\right).\end{aligned} \tag{5.1}$$

The authors extend this idea to Bayesian convolutional neural networks (BCNNs) [Gal and Ghahramani, 2015], stating that the GP interpretation is lost but observing that MC dropout estimates tend to increase accuracy. Furthermore, predictive uncertainty estimates for BCNNs are not discussed, and it is made clear that eq. 5.1 is inappropriate for use in a classification setting. However, as is often the case among many practitioners in the field of deep learning, theory plays second fiddle to what works well in practice.

Despite the apparent lack of theoretically grounded uncertainty estimates for BCNNs, several researchers [Feinman et al., 2017; Leibig et al., 2017] have developed ad hoc approximations of eq. 5.1 for use in image classification problems, which seem to work well in practical applications.

In this chapter we continue in the experimental tradition of the deep learning community by exploring the empirical properties of these ad hoc uncertainty estimates. As far as we are aware, there has not been much work directed towards determining the viability of ad hoc uncertainty approximation in multi-class classification problems. The first and main focus of this chapter will therefore be the extension of $\hat{\sigma}_{\text{pred}}$ [Leibig et al., 2017] to the multi-class image classification problem CIFAR-10 (see chapter 6 for details on the data set). We start by examining the distributions of $\hat{\sigma}_{\text{pred}}$ for correctly and incorrectly classified images. Next we establish a connection between the results of the confusion matrix and class-specific uncertainty estimates. We follow this with a brief exploration of the relationship between approximated predictive uncertainty and MC dropout estimates. Next, we explore how $\hat{\sigma}_{\text{pred}}$ responds to feature degradation by adding random noise to the images. Finally, we test the effect of varying dropout rates on approximated uncertainty and predictive accuracy.

Uncertainty estimation in deep neural networks is an extremely active and fast-moving field of study. While working on the problem stated above, we became aware of Feinman and colleagues' work (section 4.4.3) on detecting adversarial image inputs using $\hat{\sigma}^2$ [Feinman et al., 2017]. Feinman's approximation is developed for a multi-class setting, so we will present a brief analysis of the empirical properties of $\hat{\sigma}^2$ and how they compare to $\hat{\sigma}_{\text{pred}}$, although this exploration will not be as extensive as our examination of $\hat{\sigma}_{\text{pred}}$.

Finally, we introduce $\hat{\sigma}_{\text{model}}$, a novel predictive uncertainty approximation that is based on calculating the mean of the standard deviations of each element in $\hat{\boldsymbol{\mu}}$. We briefly explore $\hat{\sigma}_{\text{model}}$ and show that it compares favourably to both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$ in some important respects. At the end of this chapter we put all three approximated predictive uncertainties to the test in a simple uncertainty-informed referral experiment, again finding that our approximation $\hat{\sigma}_{\text{model}}$ outperforms both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$. Furthermore, we estimate values for some established uncertainty measures for classification suggested by Gal (see end of section 4.4.1) and see how they compare to $\hat{\sigma}_{\text{model}}$, $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$.

We will explore the different approximations of predictive uncertainty using the model **lenet-allMC**, which is a variant of the LeNet architecture [LeCun et al., 1998] described in section 3.7.4. See chapter 6 for implementation details. The choice of the LeNet is mainly motivated by the following:

- LeNet is the same model used to introduce the idea of MC dropout for classification [Gal and Ghahramani, 2016]. Moreover, LeNet is typically used as a baseline or proof-of-concept model in the literature.
- Compared to state-of-the art architectures[1], LeNet is simple to implement and trains quickly. This facilitates fast prototyping and efficient experimentation.
- Recent research [Guo et al., 2017] suggests that the softmax output of simple models such as LeNet can be interpreted as model confidence (see section 5.3.3). This gives us an additional perspective when analysing approximated predictive uncertainty.

We start section 5.1 by restating and introducing the notation needed for analysis. In section 5.2 we highlight the differences in predictive accuracy between standard dropout CNNs and BCNNs. From section 5.3 and onwards we will explore the empirical properties of the different approximations of predictive uncertainty.

5.1 Notation

Since we are in a multi-class setting we use the softmax activation to output a vector of K class probabilities. As stated earlier, the vector of softmax probabilities obtained after the t'th stochastic forward pass is denoted $p(\hat{\boldsymbol{y}}|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t)$, where $\hat{\boldsymbol{\omega}}_t$ denotes the sampled parameters resulting from dropout.

The CIFAR-10 data set consists of $K = 10$ classes. Conventionally, the classes are indexed from 0 to 9. Thus, the vector of MC dropout estimates is given by

$$\begin{aligned}
\hat{\boldsymbol{\mu}} &= (\hat{\mu}_0, \hat{\mu}_1, \ldots, \hat{\mu}_9), \\
\hat{\mu}_k &= \frac{1}{T}\sum_{t=1}^{T} p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t), \quad k \in \{0, \ldots, 9\}.
\end{aligned}$$

[1]See for example Deep Residual Networks [He et al., 2016] and Densely Connected Convolutional Networks [Huang et al., 2017]

Let the predicted class k correspond to the largest element of $\hat{\boldsymbol{\mu}}$, given by

$$\hat{\mu}_{\text{pred}} = \max_{\hat{\mu}_k} \hat{\boldsymbol{\mu}}.$$

and let the runner-up prediction be given by

$$\hat{\mu}_{\text{run}} = \max_{\hat{\mu}_j} \hat{\boldsymbol{\mu}}, \quad \hat{\mu}_j \neq \hat{\mu}_{\text{pred}},$$

i.e. $\hat{\mu}_{\text{run}}$ is the second largest element in $\hat{\boldsymbol{\mu}}$.

In our analysis we are often interested in the softmax probabilities of the correctly and incorrectly classified images. Let the superscripts 0 and 1 correspond to the statistics associated with incorrect and correct classifications, respectively. For example, $\hat{\mu}^0_{\text{pred}}$ and $\hat{\mu}^0_{\text{run}}$ denote the prediction and runner-up for an incorrectly classified image.

The definitions of $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$ were given in chapter 4.4.3, but we will restate them here and at the start of the relevant sections. We will also define $\hat{\sigma}_{\text{model}}$ here, and restate it in the relevant section.

$$\hat{\sigma}_{\text{pred}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}[p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_{\text{pred}}]^2}.$$

$$\hat{\sigma}^2 = \frac{1}{K}\sum_{k=1}^{K}\hat{\sigma}^2_k, \quad \hat{\sigma}^2_k = \frac{1}{T}\sum_{t=1}^{T}[p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_k]^2, \quad k = 1, ..., K.$$

$$\hat{\sigma}_{\text{model}} = \frac{1}{K}\sum_{k=1}^{K}\sqrt{\frac{1}{T}\sum_{t=1}^{T}[p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_k]^2}.$$

We set $T = 100$ following [Gal and Ghahramani, 2015]. A superscript of 0 or 1 corresponds to the approximated uncertainty associated with an incorrect or correct classification, respectively.

For most of the analysis we will be interested in the aggregated values of the predictions, runner-up values and approximated uncertainty. A statistic enveloped by avg$(\cdot)$ denotes the mean value of that statistic. For example, avg$(\hat{\mu}^1_{\text{pred}})$ denotes the average value of $\hat{\mu}_{\text{pred}}$ for correctly classified images. Similarly, avg$(\hat{\sigma}^0_{\text{pred}})$ denotes the average approximated predictive uncertainty for incorrectly classified images.

Model	Accuracy	MC Accuracy
lenet-all	34.86 %	–
lenet	78.64 %	–
lenet-allMC	–	**83.63 %**

Table 5.1: The *Accuracy* column indicates the percentage of correct predictions using standard dropout implementations of LeNet at test time. The column *MC Accuracy* shows the overall accuracy of our BCNN implementation of LeNet. The deterministic implementation **lenet-all** with dropout after every convolution is the worst performing model. Note that MC dropout increases accuracy by almost 5 % compared to the deterministic baseline model **lenet**.

5.2 Comparing Accuracy

The purpose of this analysis is not to achieve state-of-the-art classification results[2]. However, it is interesting to see how MC dropout boosts predictive accuracy (see section 4.4.2). We show this by comparing the test time accuracy on $N = 10.000$ images using the following models (see chapter 6 for details on implementation and training):

- **lenet**: Deterministic baseline model, with dropout ($p = 0.5$)[3] only in fully connected layer.
- **lenet-all**: Deterministic variant of baseline model, with dropout ($p = 0.5$) after every convolutional layer.
- **lenet-allMC**: BCNN with dropout ($p = 0.5$) after every convolutional layer.

The results are summarised in table 5.1. The baseline model **lenet** achieves an accuracy of 78.64 % at test time. In comparison, **lenet-allMC** significantly boosts the predictive performance. By averaging $T = 100$ stochastic forward passes for each image in the test set, the model achieves an overall accuracy of 83.63 %, reducing the error rate from 21.36 % to 16.37 %. **lenet-all** is by far the worst performing model, with an accuracy of 34.86 %. This is to be expected, as standard dropout CNNs fail to generalise when every convolutional layer is followed by a dropout layer [Gal and Ghahramani, 2015]. This problem is clearly alleviated by MC dropout.

[2]As of 10.10.2018, the current state-of-the-art techniques report an error rate of 1.48 % on CIFAR-10. See [Cubuk et al., 2018] for details.

[3]The dropout rate $p = 0.5$ is based on the original implementation in [Gal and Ghahramani, 2016], see chapter 6 for details.

Label	n	avg($\hat{\mu}_{\text{pred}}$)	avg($\hat{\mu}_{\text{run}}$)	avg($\hat{\sigma}_{\text{pred}}$)	Median $\hat{\sigma}_{\text{pred}}$	IQR
0: Incorrect	1637	0.5388	0.2544	0.2082	0.2075	0.0663
1: Correct	8363	0.8407	0.0967	0.1128	0.1043	0.1661

Table 5.2: Average predictions, runner-up predictions and uncertainty for the correctly and incorrectly classified images. In addition, the median uncertainty and interquartile range (IQR) are provided. Note that on average $\hat{\sigma}_{\text{pred}}$ is higher for incorrect classifications.

5.3 Leibig's Predictive Uncertainty

In [Leibig et al., 2017] the approximated predictive uncertainty is given by the empirical standard deviation of the positive class. Extending this to a multi-class setting gives the uncertainty estimate

$$\hat{\sigma}_{\text{pred}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}[p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_{\text{pred}}]^2},$$

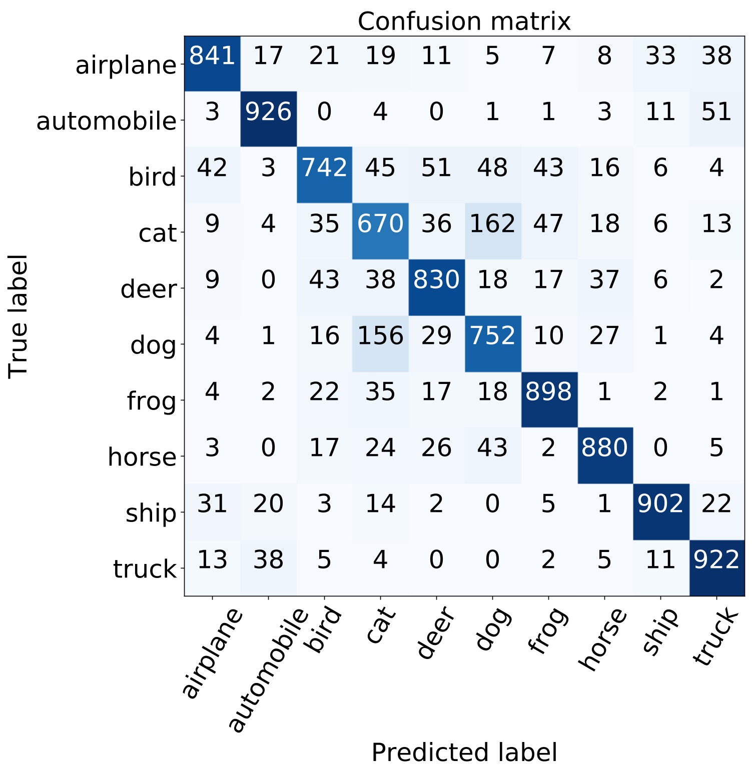
i.e. the standard deviation of the softmax values associated with the predicted class k.

Table 5.2 shows the summary statistics for **lenet-allMC** on our test set. The most important quantities are summarised in the following:

- The average approximated uncertainty of mislabelled images is greater than for the correctly labelled images (avg($\hat{\sigma}^0_{\text{pred}}$) = 0.2082 versus avg($\hat{\sigma}^1_{\text{pred}}$) = 0.1128).
- On average, incorrectly classified images output lower predictive means than correctly classified images (avg($\hat{\mu}^0_{\text{pred}}$) = 0.5388 versus avg($\hat{\mu}^1_{\text{pred}}$) = 0.8407).
- The average runner-up prediction for incorrectly classified images is higher than it is for correctly classified images (avg($\hat{\mu}^0_{\text{run}}$) = 0.2544 versus avg($\hat{\mu}^1_{\text{run}}$) = 0.0967).

We see that incorrectly classified images tend to be associated with larger uncertainty on average. Furthermore, the summary statistics indicate that softmax outputs of the predicted and runner-up classes seem to be closer to each other when the model misclassifies. This seems reasonable, as one would expect uncertainty to increase in the situation where there are one or more candidate classes for $\hat{\mu}_{\text{pred}}$, which would result in more probability being assigned to the runner-up class.

The confusion matrix in fig. 5.1 provides details on how the model performs. **lenet-allMC** is most successful when classifying images of automobiles, where 926



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	841	17	21	19	11	5	7	8	33	38
automobile	3	926	0	4	0	1	1	3	11	51
bird	42	3	742	45	51	48	43	16	6	4
cat	9	4	35	670	36	162	47	18	6	13
deer	9	0	43	38	830	18	17	37	6	2
dog	4	1	16	156	29	752	10	27	1	4
frog	4	2	22	35	17	18	898	1	2	1
horse	3	0	17	24	26	43	2	880	0	5
ship	31	20	3	14	2	0	5	1	902	22
truck	13	38	5	4	0	0	2	5	11	922

Fig. 5.1: Our model is good at labelling pictures of automobiles, where 926 images are classified correctly. The most frequently mislabelled class is cat, with only 670 correct predictions.

images are predicted correctly. The most frequently mislabelled class is cat (670 correct predictions). In images of cats the most frequent incorrect label is dog (162). For dogs the most frequent incorrect label is cat (156). In general it seems that the misclassifications belong to the same domain as the correct label (i.e. one vehicle type is mistaken for another and one species of animal is misclassified as another). This suggests that our model is unable to learn distinguishing features which adequately separate similar classes. However, one should be cautious about reading too much into the specifics of misclassifications, as the results may very well be an artefact of the model, the data set or both. For example, it is conceivable that birds are misclassified as planes due to the presence of large patches of blue sky in the image. The four most certain and uncertain images for each class are displayed[4] in fig. 5.2 (certain/correct), 5.3 (uncertain/correct), 5.4 (certain/incorrect) and 5.5 (uncertain/incorrect).

[4]We use a kernel density estimate (KDE) with a Gaussian kernel to give an idea of the distribution of $\hat{\sigma}_{\text{pred}}$ in figs. 5.2–5.5. Note that some of the boundary cut-offs are quite abrupt at 0 and 1. There exists an entire literature devoted to boundary correction in KDEs (see e.g. [Silverman, 2018]), which is beyond the scope of our use. We mainly use KDEs for illustrative purposes.

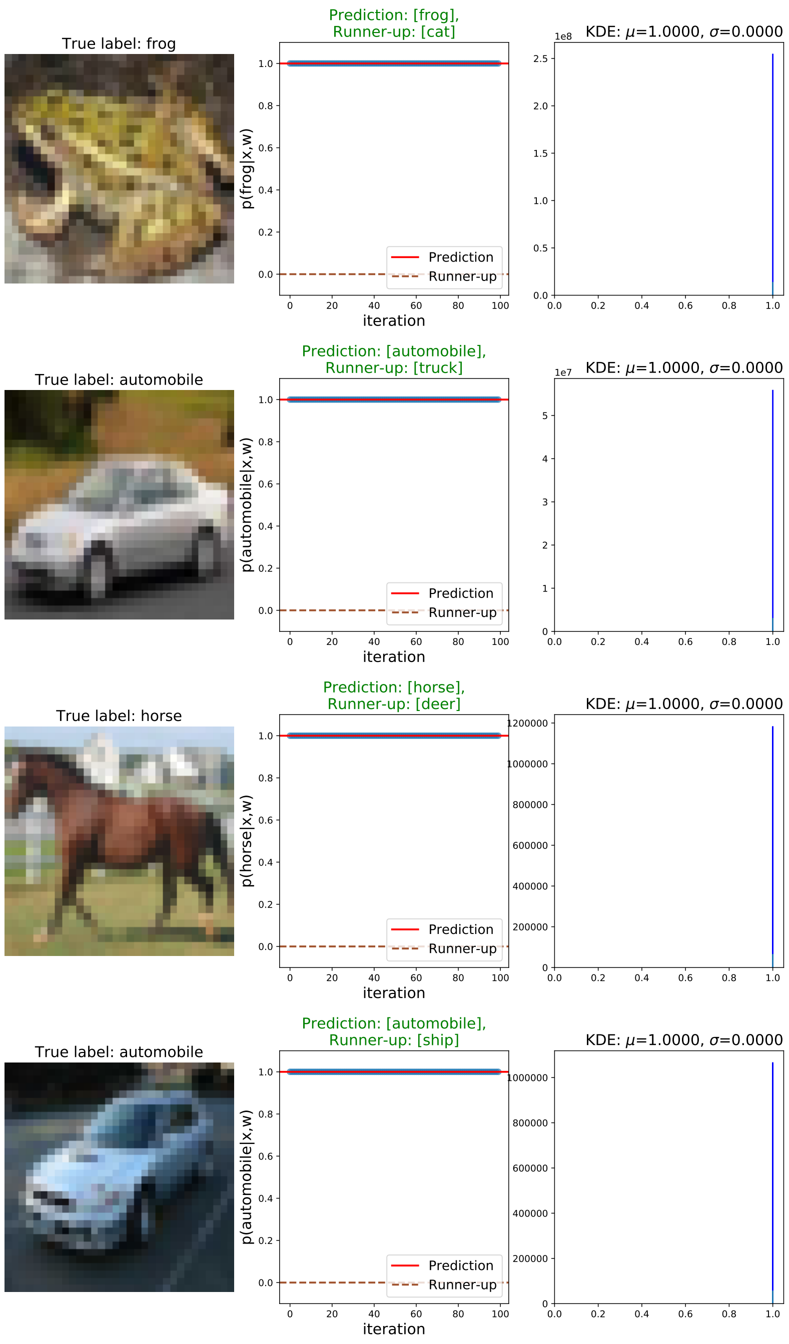


Fig. 5.2: Correctly classified images associated with low uncertainty. The leftmost plot shows the image with the corresponding ground truth label. The middle plot shows the softmax output of the predicted class for each of the $T = 100$ stochastic forward passes. $\hat{\mu}_{\text{pred}}$ is given by the solid red line, $\hat{\mu}_{\text{run}}$ is given by the dashed brown line. The title of the middle plot gives both the predicted class and the runner-up class. The rightmost plot shows a kernel density estimate (KDE) of the distribution of $\hat{\mu}_{\text{pred}}$, with a rug indicating the spread of the softmax predictions. Note that there is practically complete agreement between the softmax outputs of every stochastic forward pass.

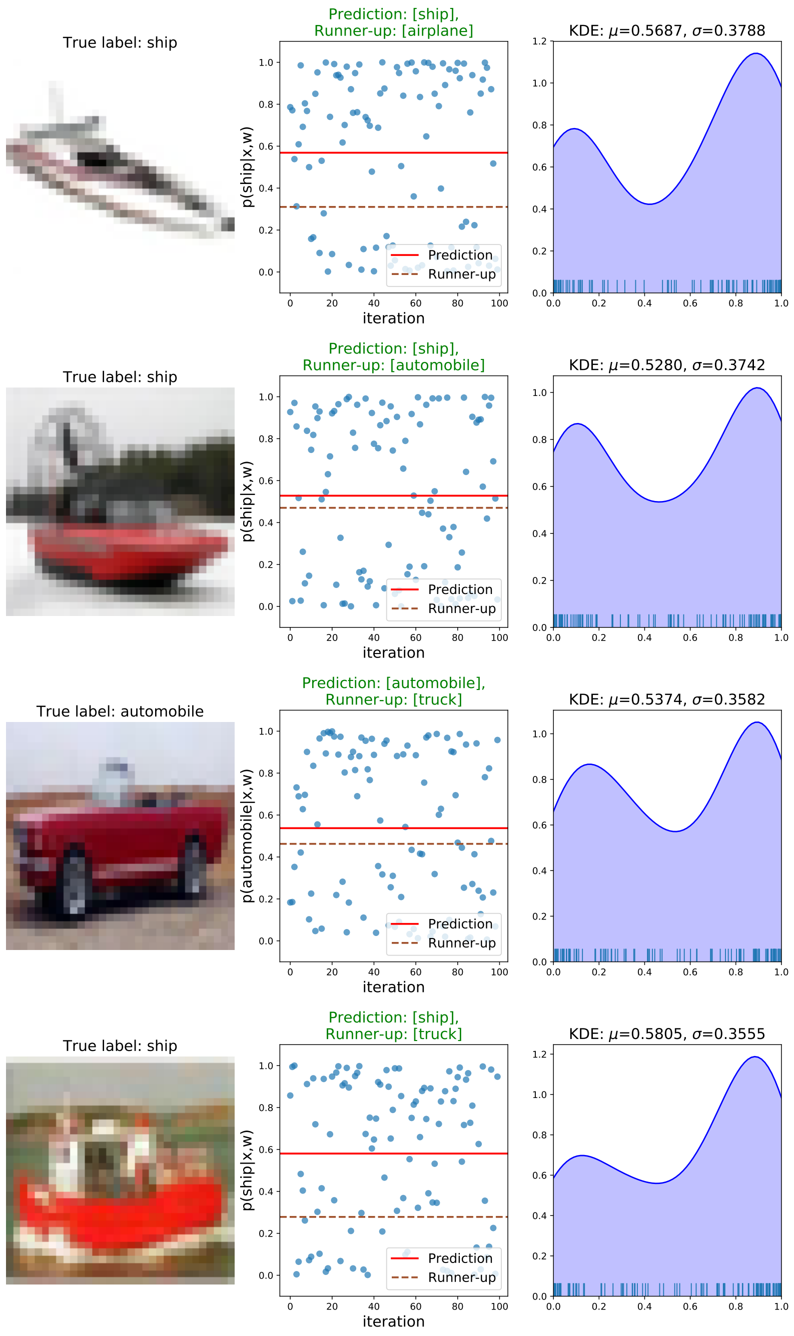


Fig. 5.3: Correctly classified images associated with high uncertainty. Here the softmax outputs exhibit high variance. Interestingly, the KDE plots indicate a bimodality in the distributions of $\hat{\mu}_{\text{pred}}$. This suggests that the model has at least two candidates for $\hat{\mu}_{\text{pred}}$, but ended up predicting the correct class.

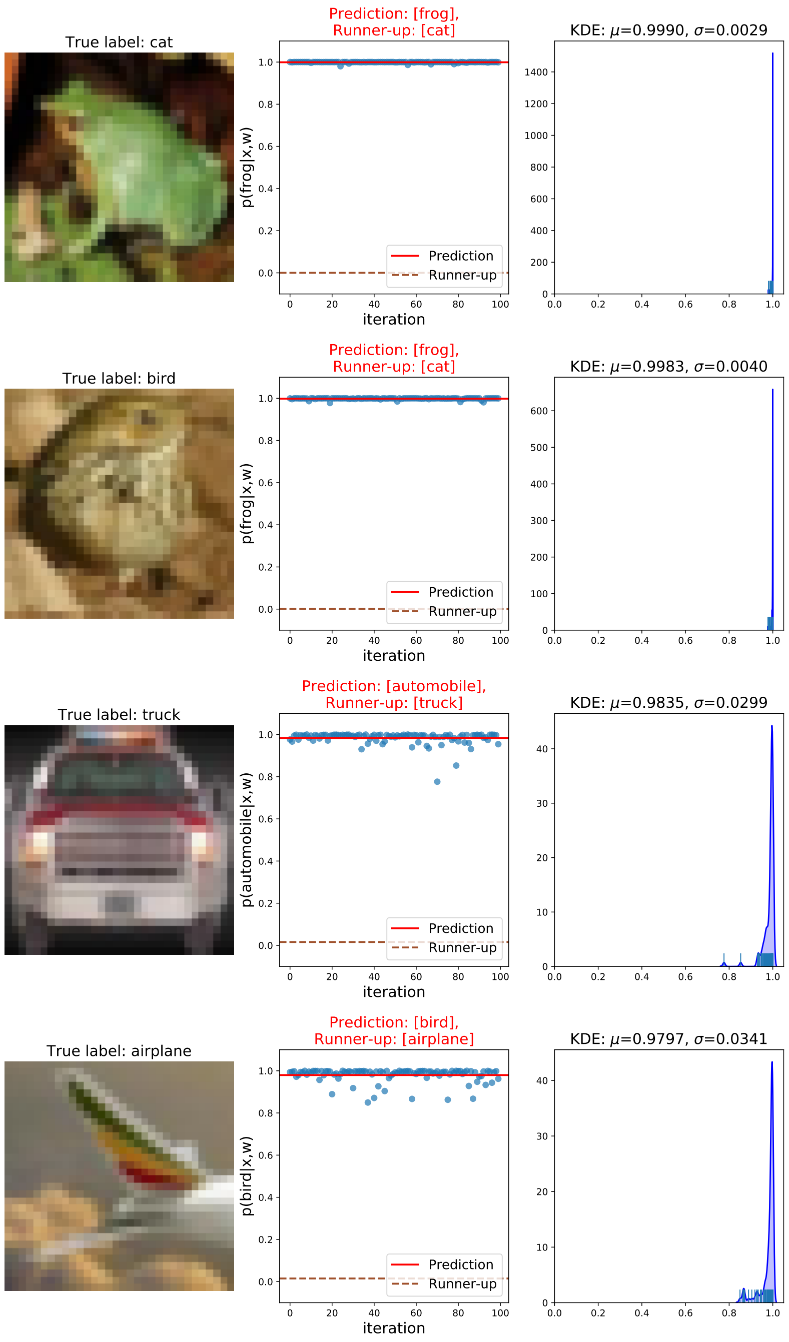


Fig. 5.4: Incorrectly classified images associated with low uncertainty. The softmax outputs exhibit low variance, even though the prediction is incorrect. This plot is interesting, because it highlights the fact that some images are genuinely hard to classify, even for humans. For instance, the most confident incorrect prediction is an image of a cat. We believe that most humans would in fact agree with the model's incorrect, but ultimately reasonable, guess that this is an image of a frog. Furthermore, it is interesting to note that the runner-up prediction is the true label in 3 out of 4 images.

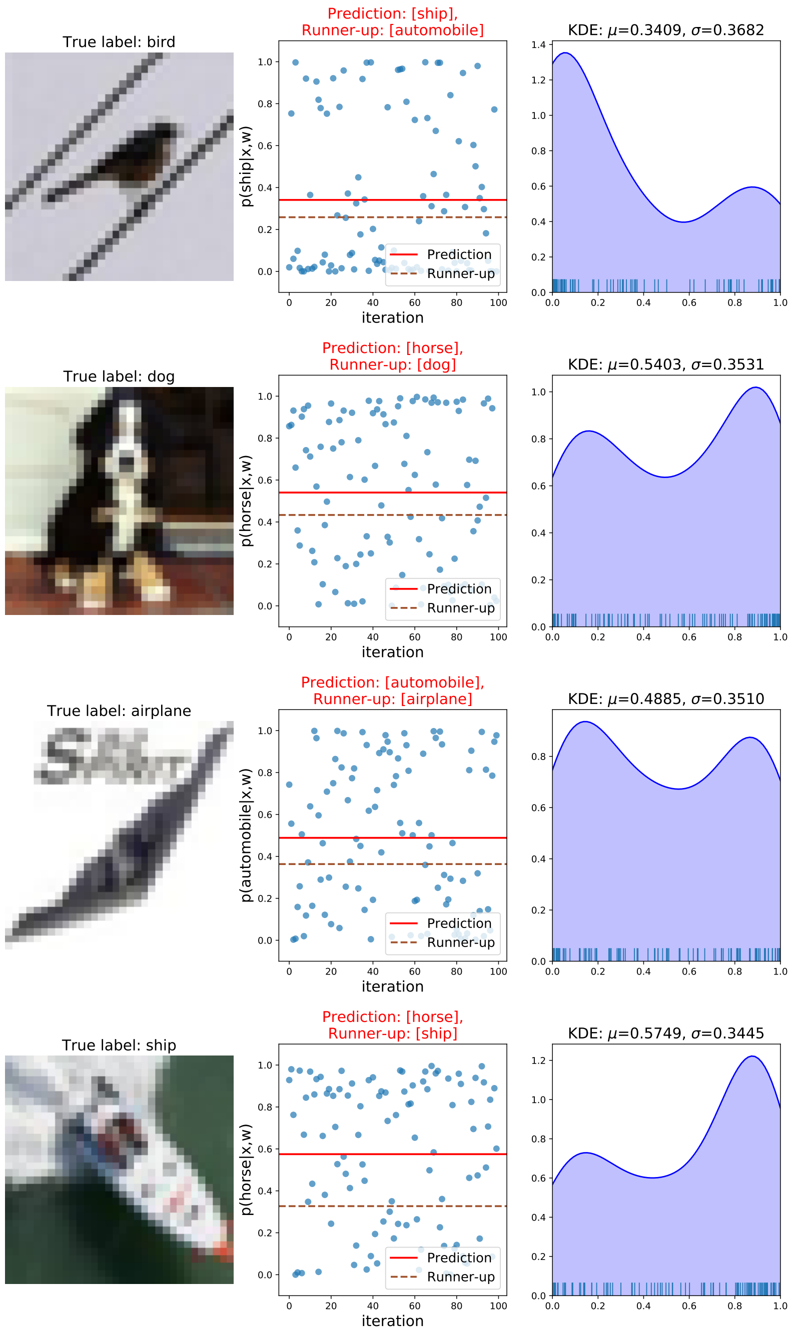
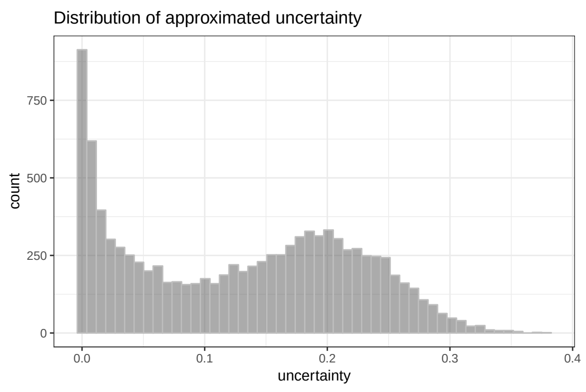
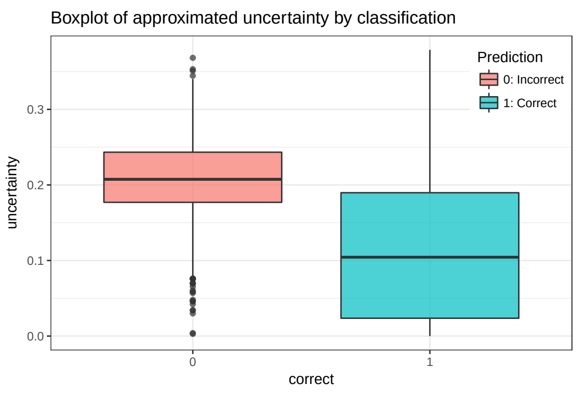


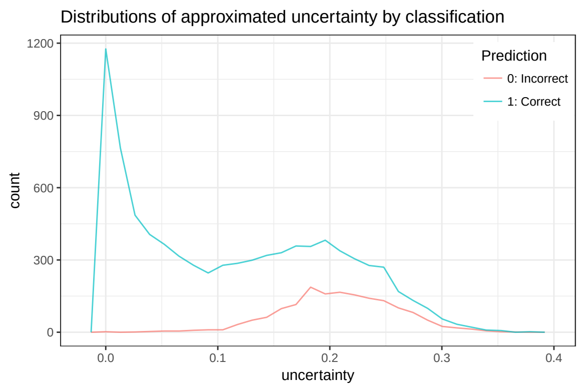
Fig. 5.5: Incorrectly classified images associated with high uncertainty. The softmax outputs exhibit high variance. Again, the KDE plots indicate a bimodality in the distribution of $\hat{\mu}_{\text{pred}}$, suggesting that the model had at least two candidates for $\hat{\mu}_{\text{pred}}$. This time the MC dropout estimate chose the incorrect label. Again, the runner-up prediction is the true label in 3 out of 4 images.



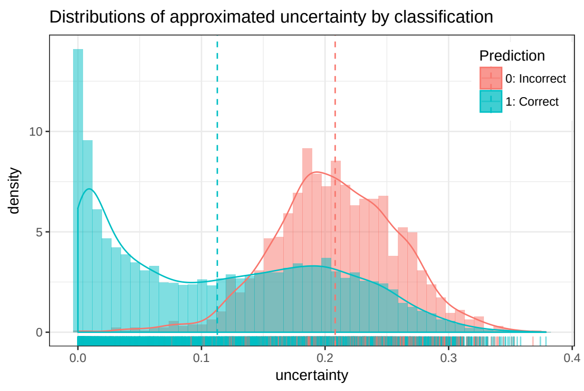
(a)



(b)



(c)



(d)

Fig. 5.6: Distribution of $\hat{\sigma}_{\text{pred}}$. Fig. 5.6a shows the total distribution of $\hat{\sigma}_{\text{pred}}$ for all $N = 10.000$ images. Figs. 5.6b and 5.6c show the distributions of $\hat{\sigma}^0_{\text{pred}}$ (in red) and $\hat{\sigma}^1_{\text{pred}}$ (in blue). Fig. 5.6d shows the density plots of $\hat{\sigma}^0_{\text{pred}}$ and $\hat{\sigma}^1_{\text{pred}}$. The plots indicate that $\hat{\sigma}_{\text{pred}}$ is a meaningful approximation of predictive uncertainty. However, there is substantial overlap between correctly and incorrectly classified images, which could pose problems for uncertainty-informed referrals.

5.3.1 Distribution of Predictive Uncertainty Estimates

If $\hat{\sigma}_{\text{pred}}$ is a useful approximation of predictive uncertainty, then we would expect the distributions of $\hat{\sigma}^0_{\text{pred}}$ and $\hat{\sigma}^1_{\text{pred}}$ to reflect this somehow. Fig. 5.6 gives several perspectives on the empirical distribution of $\hat{\sigma}_{\text{pred}}$ for all $N = 10.000$ test images.

In fig. 5.6a we see that the total distribution of $\hat{\sigma}_{\text{pred}}$ is clearly bimodal, peaking around $\hat{\sigma}_{\text{pred}} \approx 0$ and $\hat{\sigma}_{\text{pred}} \approx 0.2$. Figs. 5.6b and 5.6c give us an idea of how the correctly and incorrectly classified images contribute to predictive uncertainty, respectively. Fig. 5.6b shows a clear distinction between the two classification categories. Note the presence of outliers in both categories. The outliers present among the incorrectly classified images which are close to $\hat{\sigma}^0_{\text{pred}} = 0$ indicate observations for which the model outputs low uncertainty estimates while simultaneously misclassifying (fig. 5.4).

Fig. 5.6c gives a clearer picture of how the different classification categories contribute to the total distribution of $\hat{\sigma}_{\text{pred}}$. We see that the incorrect predictions are centred around a higher uncertainty, whereas far more of the correctly predicted classes are concentrated around a low uncertainty value. The incorrect classifications greatly contribute to the bimodality in fig. 5.6a, but it is clear that the distribution of $\hat{\sigma}^1_{\text{pred}}$ is itself bimodal. The density histogram with overlaid kernel density estimates in fig. 5.6d gives us further insight into how the two distributions compare to each other. The dashed lines indicate the mean values of $\hat{\sigma}^0_{\text{pred}}$ and $\hat{\sigma}^1_{\text{pred}}$.

From fig. 5.6 it seems clear that the distributions of $\hat{\sigma}^0_{\text{pred}}$ and $\hat{\sigma}^1_{\text{pred}}$ are meaningfully different from one another, at least for this particular choice of model and data set. This indicates that $\hat{\sigma}_{\text{pred}}$ contains some valueable information about the model's confidence in its predictions. However, there is a substantial amount of overlap between the two distributions, which could cause problems in an uncertainty-based referral setting (see section 5.7).

5.3.2 Class-Specific Predictive Uncertainty

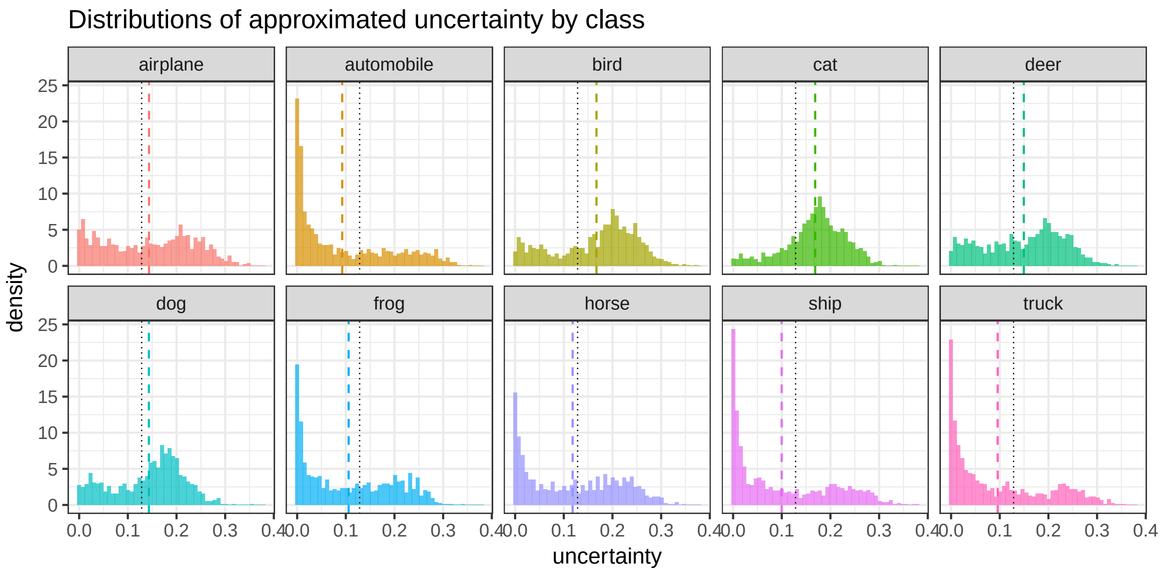
As mentioned in chapter 2.5.2, the confusion matrix is a widely applied performance measure and is often used to provide details on how standard dropout networks fail at test time. Thus, if $\hat{\sigma}_{\text{pred}}$ captures predictive uncertainty, then we would expect the class-specific distributions of $\hat{\sigma}_{\text{pred}}$ to somehow echo the confusion matrix in fig. 5.1. Let $\hat{\sigma}_k$ denote the class-specific uncertainty, where $k \in \{\text{airplane}, ..., \text{truck}\}$ is the corresponding class label.

The relationship between uncertainty and predictive accuracy is reflected in table 5.3 and in the distributions of $\hat{\sigma}_k$ in fig. 5.7a. Both indicate that the model has less confidence in the classes which it performs relatively poorly on. As noted in section 5.2, the model clearly mistakes some cats for dogs and some dogs for cats. This confusion seems to be reflected in the respective distributions of $\hat{\sigma}_{cat}$ and $\hat{\sigma}_{dog}$. It is interesting to note that, on average, the highest uncertainty is associated with images of birds, not cats. Clearly there is some feature which contributes more uncertainty to the bird class than to the cat class, which is where we would hope to see the largest average uncertainty values (since images of cats are most frequently misclassified). The boxplots in fig. 5.7b provide a different perspective on the spread of $\hat{\sigma}_k$ for each class label k.

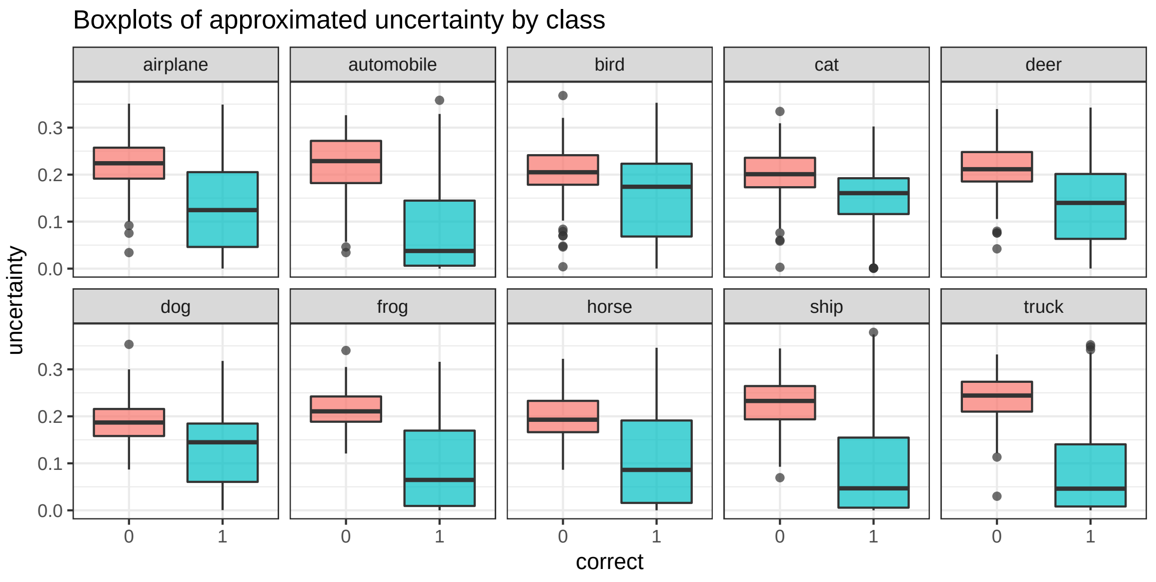
k	n	avg($\hat{\sigma}_k$)
cat	670	0.1514
bird	**742**	**0.1540**
dog	752	0.1280
deer	830	0.1366
airplane	841	0.1288
horse	880	0.1074
frog	898	0.0936
ship	902	0.0859
truck	922	0.0838
automobile	**926**	**0.0825**

Table 5.3: Number of correct predictions n by class k, and their associated mean uncertainty avg($\hat{\sigma}_k$) sorted by n. The bold-faced rows correspond to the minimum and maximum uncertainty. Note that the class with the least correct predictions (cat) is among the most uncertain, while the class with the most correct predictions (automobile) is the least uncertain. Also note how increasing uncertainty tends to coincide with lower number of correct predictions.

In summary, the class-specific approximations of uncertainty are consistent with the confusion matrix in fig. 5.1: Even though there is not a one-to-one correspondence between $\hat{\sigma}_k$ and the number of correct classifications n, on average higher uncertainty does seem to be associated with classes that tend to be mislabelled. Again, this indicates that $\hat{\sigma}_{\text{pred}}$ captures some useful measure of uncertainty at test time.



(a) Each panel represents the distribution of $\hat{\sigma}_{\text{pred}}$ for a particular class, indicated by the panel title. The dotted black lines indicate the mean uncertainty of the entire test set. The dashed lines correspond to class-specific mean uncertainty. Note the distributions of the most uncertain classes, indicating that the classes which tend to be mislabelled most frequently are also those where the distribution $\hat{\sigma}_k$ tends to be centred around larger values.



(b) The red box indicates the uncertainty distribution of incorrectly classified images. The blue box corresponds to correctly classified images.

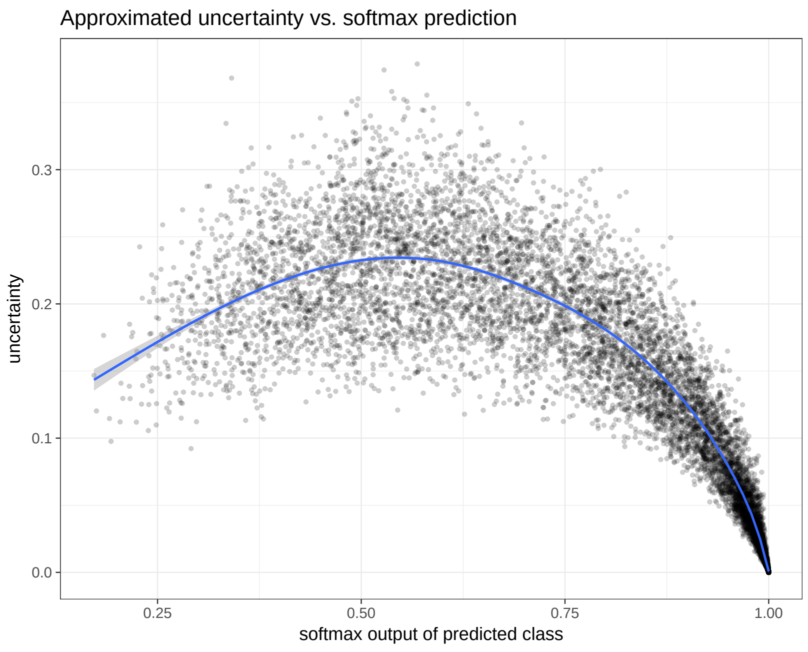
Fig. 5.7: Distributions of class-specific uncertainty (Leibig).

5.3.3 Predictive Uncertainty and the Predictive Mean

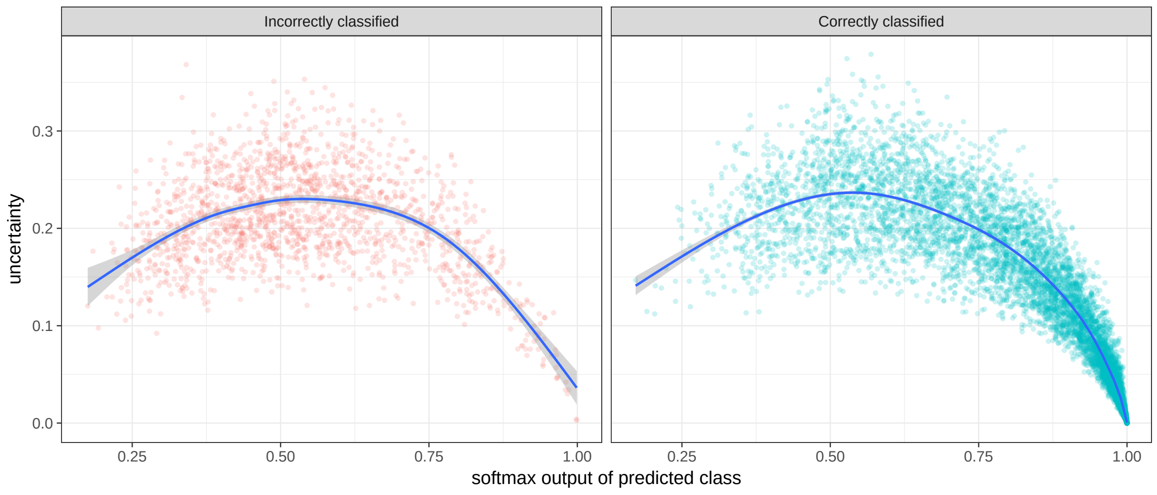
In his PhD thesis, Gal points out that the softmax outputs of neural networks are often mistakenly interpreted as a model's confidence in its predictions [Gal, 2016]. However, recent work by Guo et. al. suggests that simpler models such as LeNet are well-calibrated, meaning that the probability of the predicted class closely matches the expected accuracy of the network [Guo et al., 2017]. Consequently, the softmax probability of the predicted class in a well-calibrated model can be interpreted as confidence.

Following the results in [Guo et al., 2017], we would naively expect lower values of $\hat{\mu}_{\text{pred}}$ to be associated with higher uncertainty (this observation is not necessarily restricted to the multi-class setting). Fig. 5.8 shows the relationship between $\hat{\sigma}_{\text{pred}}$ and $\hat{\mu}_{\text{pred}}$. The plot indicates a concave shape, which contradicts our initial assumption. This is interesting, because if $\hat{\mu}_{\text{pred}}$ is to be interpreted as model confidence, then it is obvious from fig. 5.8 that predictions based on relatively low softmax values can be associated with low uncertainty estimates. This highlights a drawback of using $\hat{\sigma}_{\text{pred}}$ for uncertainty-informed referrals: The model can be quite certain (in terms of $\hat{\sigma}_{\text{pred}}$) that it is uncertain (in terms of $\hat{\mu}_{\text{pred}}$).

Furthermore, $\hat{\sigma}_{\text{pred}}$ seems to peak around $\hat{\mu}_{\text{pred}} \approx 0.5$. By colour-coding the observations based on the value of the runner-up prediction $\hat{\mu}_{\text{run}}$ (as shown in fig. 5.9) it becomes clear that the largest values of $\hat{\mu}_{\text{run}}$ cluster around $\hat{\mu}_{\text{pred}} \approx 0.5$. This is not surprising, simply because when $\hat{\mu}_{\text{pred}} = 0.5$ we have a maximum of left-over probability available to be assigned to $\hat{\mu}_{\text{run}}$. Interestingly, the points for which both $\hat{\mu}_{\text{pred}}$ and $\hat{\mu}_{\text{run}}$ have large softmax values are spread across a wide range of $\hat{\sigma}_{\text{pred}}$, suggesting two scenarios: (1) there is a tight race between two candidate classes, with little variation in the softmax probabilities or (2) there is a continuous back-and-forth between two candidate classes, resulting in greater variation in the softmax probabilities.

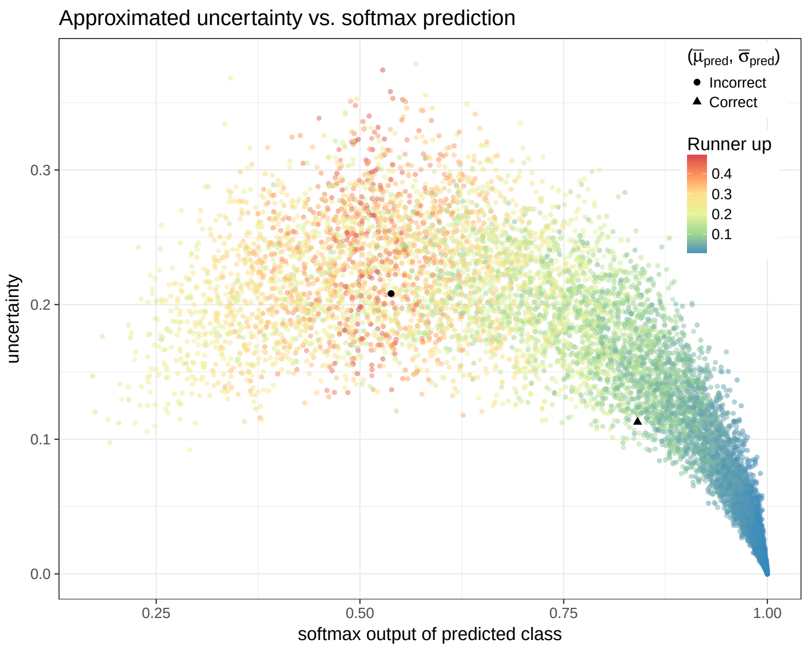


(a) In the above we have plotted $\hat{\sigma}_{\text{pred}}$ against $\hat{\mu}_{\text{pred}}$ for all $N = 10.000$ test observations. The blue line is a smoothed approximation to aid interpretability. Observe the concave shape, which highlights a potential drawback of using $\hat{\sigma}_{\text{pred}}$ for uncertainty-informed referrals: The model can be quite certain (in terms of $\hat{\sigma}_{\text{pred}}$) that it is uncertain (in terms of $\hat{\mu}_{\text{pred}}$).

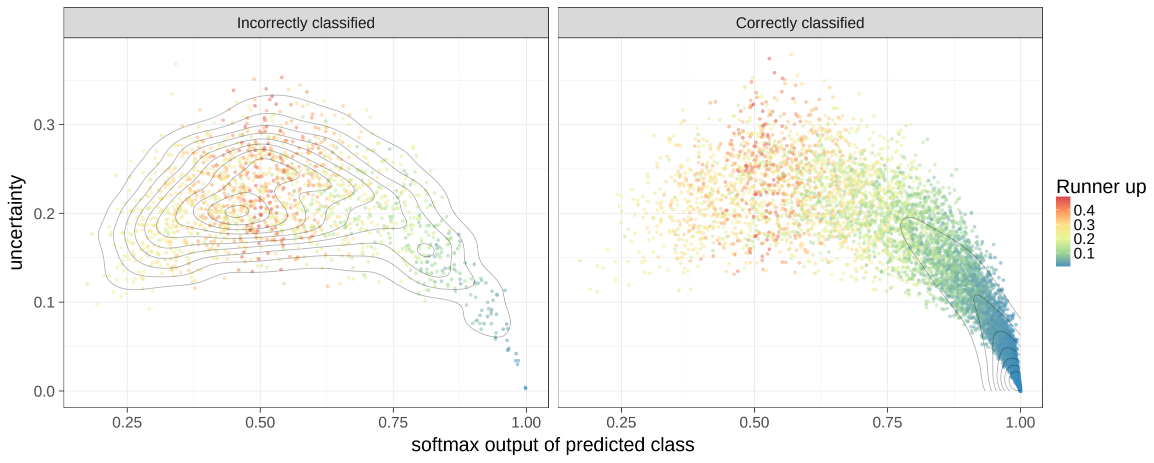


(b) Above we have done the same as in fig. 5.8a, but in addition we have partitioned the observations by classification status.

Fig. 5.8: $\hat{\sigma}_{\text{pred}}$ vs. softmax prediction.



(a) The observations are colour-coded by the values of $\hat{\mu}_{\text{run}}$. A blue point corresponds to an observation with a low value of $\hat{\mu}_{\text{run}}$. A red point corresponds to an observation with a high value of $\hat{\mu}_{\text{run}}$. Additionally, we have plotted the average mean and uncertainty for correctly (black dot) and incorrectly (black triangle) classified observations. The points for which both $\hat{\mu}_{\text{pred}}$ and $\hat{\mu}_{\text{run}}$ have large softmax values are spread across a wide range of $\hat{\sigma}_{\text{pred}}$, suggesting two scenarios: (1) there is a tight race between two candidate classes, with little variation in the predictions or (2) there is a back-and-forth between two candidate classes resulting in more predictive variance.



(b) We follow the same procedure as in fig. 5.9a, and additionally partition the observations by classification status and overlay the respective plots with a 2D kernel density estimate, which gives an idea of the joint distribution of $\hat{\mu}_{\text{pred}}$ and $\hat{\sigma}_{\text{pred}}$.

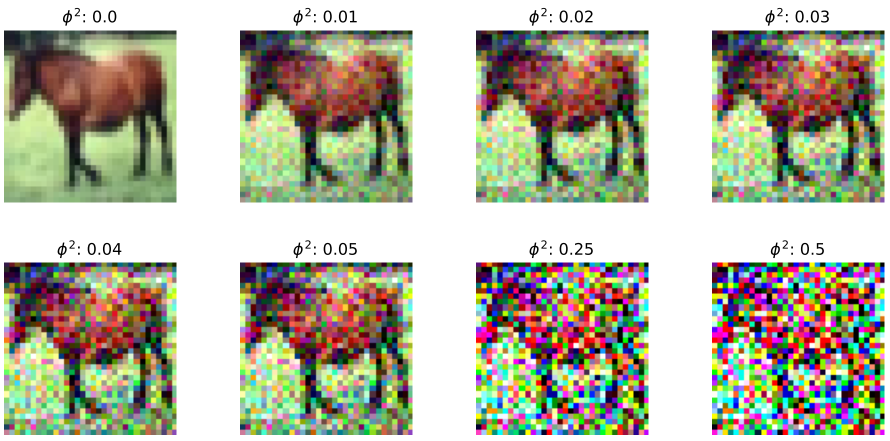
Fig. 5.9: $\hat{\sigma}_{\text{pred}}$ vs. softmax prediction and runner-up probabilities.

5.3.4 Predictive Uncertainty in Noisy Images

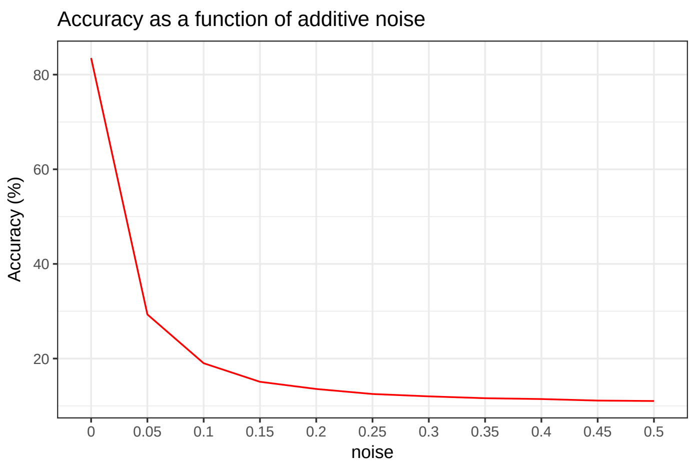
Another way to examine if $\hat{\sigma}_{\text{pred}}$ captures useful uncertainty information is to add random noise to the images (see 6.2.1 for details). The idea is that a meaningful uncertainty estimate should increase as the images lose their distinguishing features due to noise corruption. In the following ϕ^2 denotes the variance of the added Gaussian noise.

Initially, we expect the classification accuracy of the model to stabilise at around 10 % as the amount of added noise increases. This is due to the fact that the most noisy images are essentially stripped of any meaningful features learned by the network. The model should consequently achieve an accuracy that corresponds to a random guess in a situation with 10 classes. Fig. 5.10b shows that this is indeed the case. We see a rapid decrease in accuracy as noise is added, and we see the that the classification accuracy stabilises at 10 % as expected. Note that the most rapid decrease in accuracy happens for relatively low values of ϕ^2, and stabilises as ϕ^2 approaches 0.5. For illustrative purposes we therefore examine the cases where $\phi^2 \in \{0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.25, 0.5\}$ as shown in fig. 5.10a.

Adding noise clearly has an immediate effect on estimated uncertainty for the entire data set, as shown in fig. 5.11a. Grouping the data by classification status reveals that noise corruption has the strongest effect on the estimated uncertainty of correctly classified images, as shown in table 5.4, and in fig. 5.11b. The predictive uncertainty increases as ϕ^2 increases for either classification status, but $\text{avg}(\hat{\sigma}^1_{\text{pred}})$ increases much more than $\text{avg}(\hat{\sigma}^0_{\text{pred}})$. Here $\hat{\sigma}_{\text{pred}}$ is clearly demonstrating a desirable property: As the predictions approach random guessing, the model becomes increasingly uncertain about the predictions it gets right. This is also reflected in fig. 5.12, where we see that the distributions of $\hat{\sigma}^0_{\text{pred}}$ and $\hat{\sigma}^1_{\text{pred}}$ approach each other as ϕ^2 approaches 0.5.

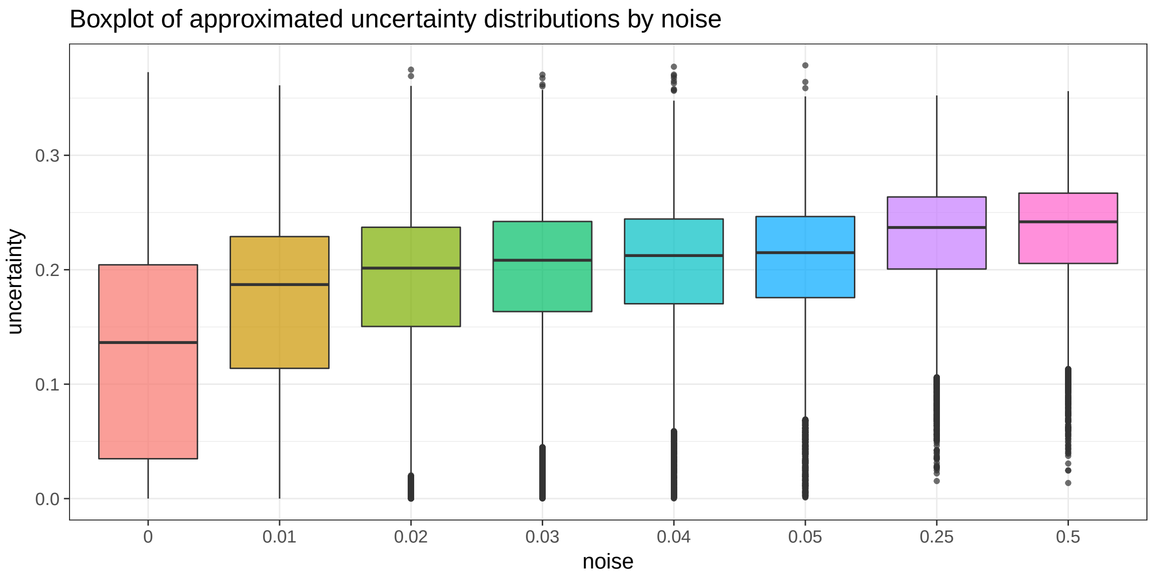


(a) The distinguishing features of an image are gradually lost as the amount of additive noise ϕ^2 increases.

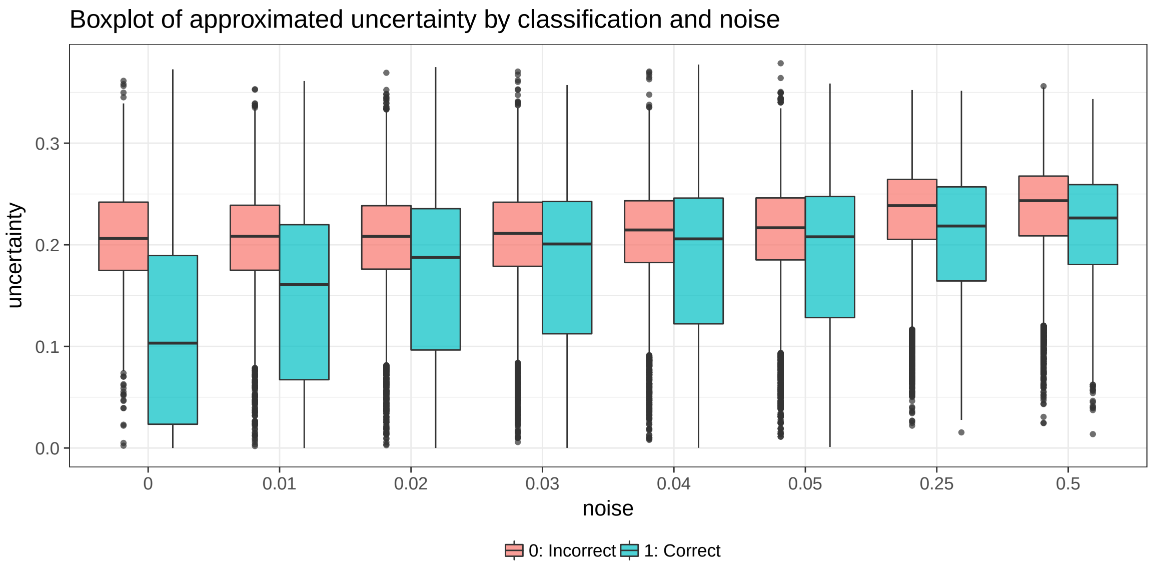


(b) The accuracy approaches 10 % as ϕ^2 increases. This is due to the fact that the most noisy images are essentially stripped of any meaningful features learned by the network. Consequently, the predictions for the most noisy images are essentially random guesses.

Fig. 5.10: We can distort an image by adding Gaussian noise $\mathcal{N}(0, \phi^2)$, as shown in fig 5.10a. The effect of the added noise on classification accuracy is shown in fig. 5.10b.



(a) Adding noise clearly has an effect on $\hat{\sigma}_{\text{pred}}$. The median $\hat{\sigma}_{\text{pred}}$ increases as ϕ^2 increases. Additionally, the distribution of $\hat{\sigma}_{\text{pred}}$ gets more concentrated around the median.



(b) The distribution of $\hat{\sigma}_{\text{pred}}$ for incorrect classifications is shown in red. The median increases slightly and the spread decreases somewhat. The correctly classified cases are shown in blue. Here, the median clearly increases and there is a significant reduction in the spread of the distribution.

Fig. 5.11: Boxplots of uncertainty vs. added noise.

ϕ^2	n		avg($\hat{\mu}_{\text{pred}}$)		avg($\hat{\mu}_{\text{run}}$)		avg($\hat{\sigma}_{\text{pred}}$)		Median $\hat{\sigma}_{\text{pred}}$		IQR	
0.00	1636	**8364**	0.5372	**0.8409**	0.2532	**0.0969**	0.2075	**0.1129**	0.2062	**0.1032**	0.0672	**0.1660**
0.01	3680	**6320**	0.5479	**0.7688**	0.2222	**0.1293**	0.2040	**0.1471**	0.2085	**0.1607**	0.0640	**0.1526**
0.02	5092	**4908**	0.5564	**0.7284**	0.2090	**0.1456**	0.2038	**0.1668**	0.2084	**0.1876**	0.0624	**0.1391**
0.03	6002	**3998**	0.5617	**0.7037**	0.1997	**0.1543**	0.2065	**0.1770**	0.2113	**0.2008**	0.0631	**0.1302**
0.04	6602	**3398**	0.5671	**0.6874**	0.1928	**0.1584**	0.2090	**0.1823**	0.2146	**0.2058**	0.0608	**0.1237**
0.05	7053	**2947**	0.5708	**0.6808**	0.1883	**0.1581**	0.2116	**0.1857**	0.2167	**0.2078**	0.0610	**0.1192**
0.25	8742	**1258**	0.6194	**0.6943**	0.1682	**0.1498**	0.2305	**0.2087**	0.2385	**0.2185**	0.0590	**0.0926**
0.50	8892	**1108**	0.6359	**0.6956**	0.1652	**0.1503**	0.2343	**0.2164**	0.2434	**0.2264**	0.0588	**0.0786**

Table 5.4: Summary statistics for incorrect and correct classifications in noisy test sets. The bold-faced columns indicate the statistics for correctly labelled images. As the amount of added noise increases, the average uncertainty of the correct predictions approaches the average uncertainty of the incorrect predictions. Furthermore, avg($\hat{\mu}_{\text{pred}}$) and avg($\hat{\mu}_{\text{run}}$) appear to approach the same values for both correct and incorrect classifications.

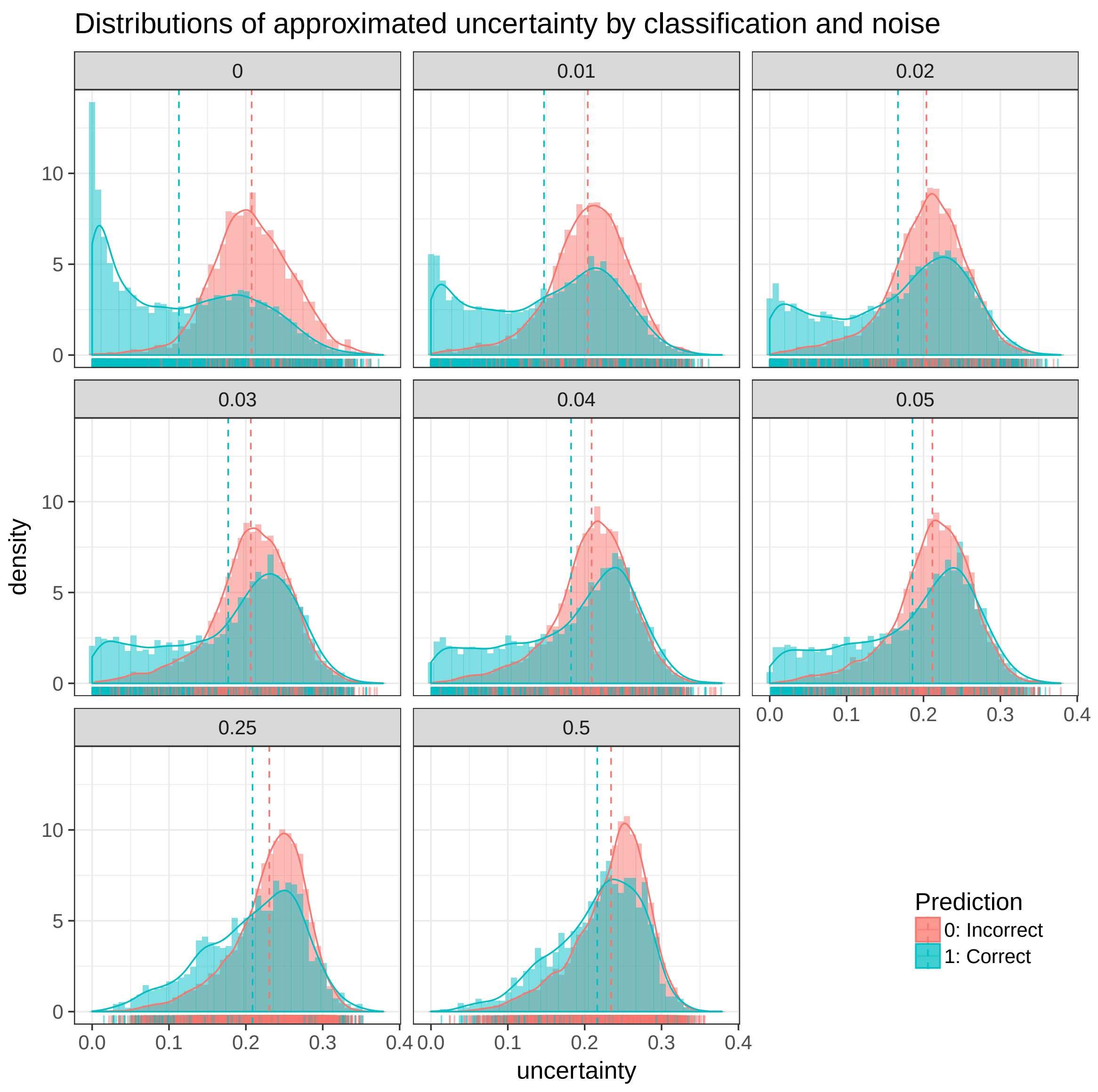
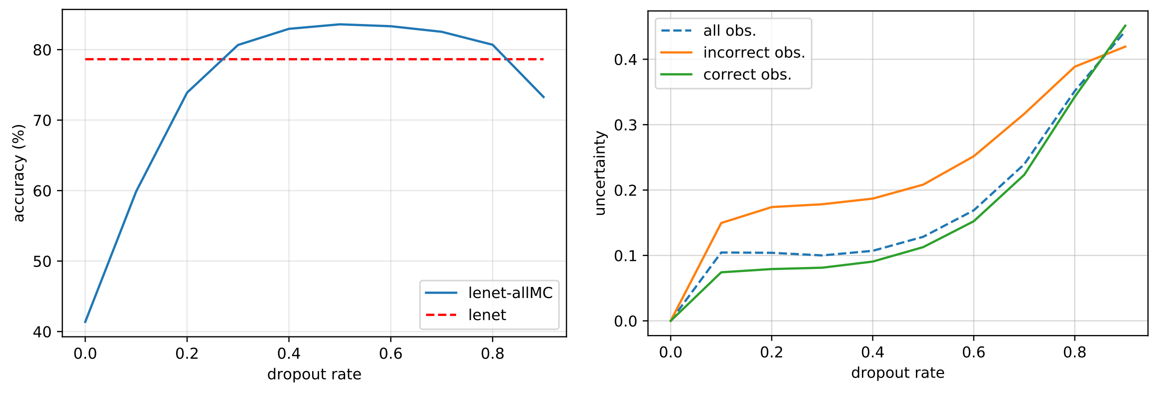


Fig. 5.12: The distributions of $\hat{\sigma}^1_{\text{pred}}$ (shown in blue) and $\hat{\sigma}^0_{\text{pred}}$ (shown in red) converge as the amount of added noise increases. The dotted lines indicate the average values of the respective distributions.



(a) Model accuracy. **(b)** Predictive uncertainty.

Fig. 5.13: The figures show the effect of adjusting the dropout rate at test time on model accuracy (5.13a) and predictive uncertainty (5.13b). The dotted red line in fig. 5.13a is the baseline accuracy of **lenet**. The blue curve is the accuracy of **lenet-allMC**. In fig. 5.13b, the dotted blue line represents the average predictive uncertainty of all observations. The orange line indicates the average predictive uncertainty for the incorrectly classified observations, whereas the green line corresponds to correctly classified observations.

.

5.3.5 The Effect of Varying Dropout Rates

Finally, let us return to our original experimental setting where there is no additive noise in the images. In this section we are interested in investigating the effect that varying the dropout rate p at test time has on model accuracy and predictive uncertainty. Specifically, we have tested $p \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$.

Recall the observation made in section 5.2, where we saw that **lenet-all** performed very poorly at test time. This is equivalent to $p = 0$. Furthermore, as $p \to 1$ almost every node in the network is switched off, which should degrade performance. Consequently, we expect model accuracy to be low when p is very small or very large. What to expect of the predictive uncertainty is more unclear. However, it seems reasonable to assume that $\hat{\sigma}_{\text{pred}}$ will increase with p. When a very large fraction of the network is inactive at test time, we would expect a larger degree of variation in the predictions.

Fig. 5.13 shows the effect of p on model accuracy and predictive uncertainty. As expected, fig. 5.13a indicates that model accuracy is low for low values of p. However, when $p \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$, model accuracy surpasses that of our baseline model **lenet**. Somewhat surprisingly, accuracy remains high for relatively large values of p. Fig. 5.13b shows how the predictive uncertainty is affected by p. We see that avg($\hat{\sigma}_{\text{pred}}$) increases with p, and that avg($\hat{\sigma}^0_{\text{pred}}$) and avg($\hat{\sigma}^1_{\text{pred}}$) approach each other as p increases.

Fig. 5.13 suggests that we can find an optimal combination of dropout rate p and average predictive uncertainty, where accuracy is high and the separation between avg($\hat{\sigma}^0_{\text{pred}}$) and avg($\hat{\sigma}^1_{\text{pred}}$) is at a maximum. This would be an interesting problem to explore in future research.

5.4 Feinman's Predictive Uncertainty

In the previous section we examined $\hat{\sigma}_{\text{pred}}$, which essentially captures the standard deviation of $T = 100$ softmax probabilities associated with the predicted class k. Feinman and colleagues present an alternative approximation of predictive uncertainty [Feinman et al., 2017], given by

$$\hat{\sigma}^2 = \frac{1}{K}\sum_{k=1}^{K}\hat{\sigma}^2_k,$$

where K is the number of classes and

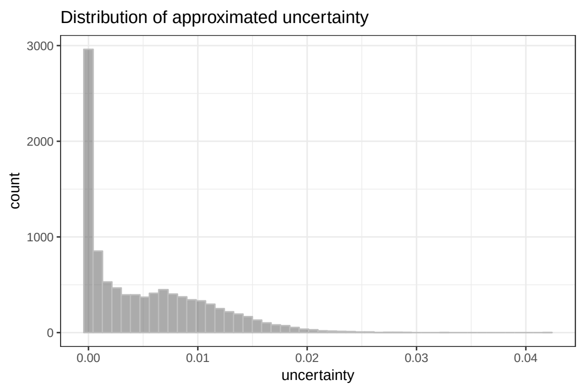
$$\hat{\sigma}^2_k = \frac{1}{T}\sum_{t=1}^{T}[p(\hat{y} = k|\boldsymbol{x}^*, \hat{\boldsymbol{\omega}}_t) - \hat{\mu}_k]^2, \quad k = 1, ..., K.$$

This approach has one obvious difference compared to our variation of Leibig's approach: $\hat{\sigma}^2$ takes into account the uncertainty associated with every class in the predictive mean vector $\hat{\boldsymbol{\mu}}$. In the following we will briefly compare and contrast $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{pred}}$.

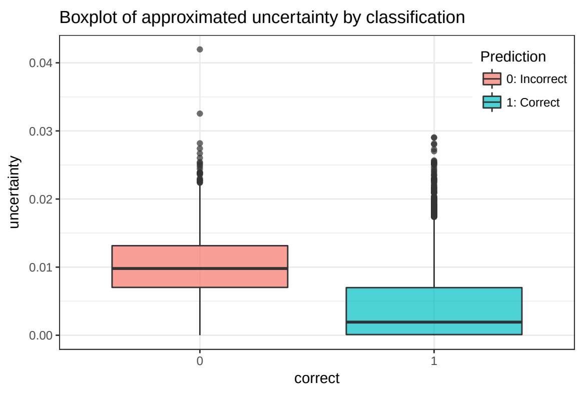
The summary statistics and distribution plots for $\hat{\sigma}^2$ are given in table 5.5 and fig. 5.14, respectively. There are two things that are immediately apparent: (1) The values

Table 5.5: Summary statistics for $\hat{\sigma}^2$ for the correctly and incorrectly classified images. Again, there appears to be a difference in the uncertainty associated with the mislabelled classes, compared to those that have been correctly classified. Note that the values of n, avg($\hat{\mu}_{\text{pred}}$) and avg($\hat{\mu}_{\text{run}}$) are the same as before. They are included for completeness.

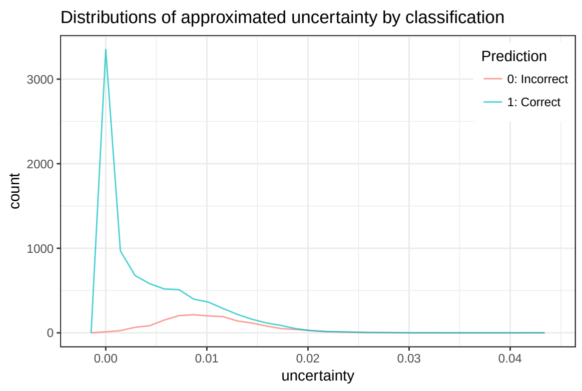
Label	n	avg($\hat{\mu}_{\text{pred}}$)	avg($\hat{\mu}_{\text{run}}$)	avg($\hat{\sigma}^2$)	Median $\hat{\sigma}^2$	IQR
0: Incorrect	1637	0.5388	0.2544	0.01032	0.00980	0.00612
1: Correct	8363	0.8407	0.0967	0.00412	0.00192	0.00690



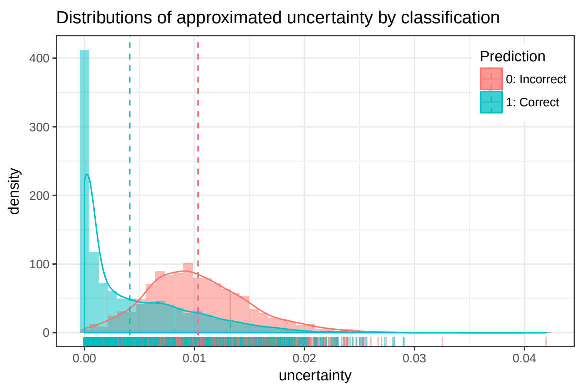
(a)



(b)



(c)



(d)

Fig. 5.14: Distribution of $\hat{\sigma}^2$. The plots indicate that $\hat{\sigma}^2$, like Leibig's $\hat{\sigma}_{\text{pred}}$, captures some meaningful estimate of the predictive uncertainty. Similarly to $\hat{\sigma}_{\text{pred}}$, there is substantial overlap between the distributions of $\hat{\sigma}^2$ for correctly and incorrectly classified images, which could pose problems for uncertainty-informed referrals.

k	n	avg($\hat{\sigma}_k$)
cat	670	0.00593
bird	**742**	**0.00606**
dog	752	0.00439
deer	830	0.00487
airplane	841	0.00495
horse	880	0.00384
ship	898	0.00303
frog	902	0.00311
truck	922	0.00302
automobile	**926**	**0.00301**

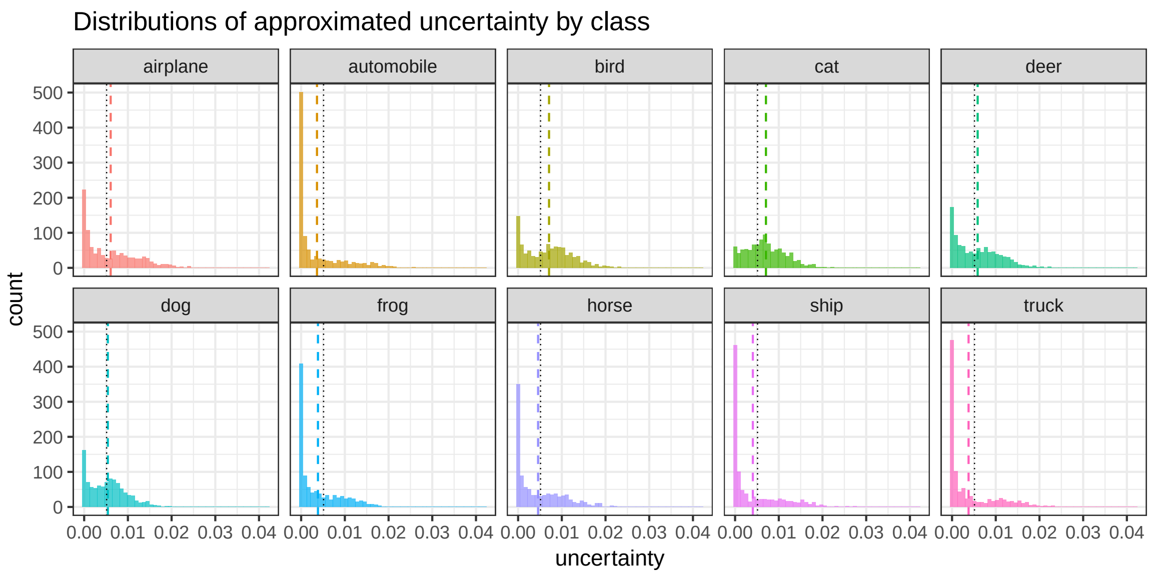
Table 5.6: Number of correct predictions n in each class and their associated average uncertainty, sorted by n. Like $\hat{\sigma}_{\text{pred}}$, Feinman's $\hat{\sigma}^2$ appears to echo the confusion matrix in fig. 5.1

of $\hat{\sigma}^2$ are much smaller in magnitude than the corresponding values for $\hat{\sigma}_{\text{pred}}$ and (2) $\hat{\sigma}^2$ has a larger number of outliers as compared to $\hat{\sigma}_{\text{pred}}$. Regarding (1), this could simply be due to the scaling associated with the taking the square root in the case of $\hat{\sigma}_{\text{pred}}$.

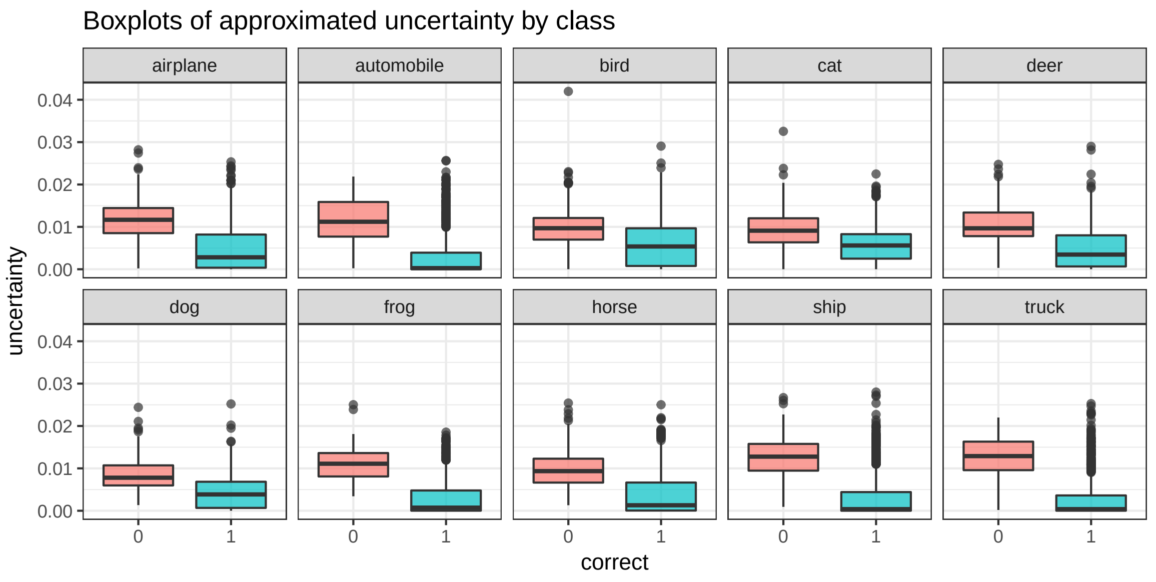
We recognise some of the same patterns as in $\hat{\sigma}_{\text{pred}}$, namely a slight bimodality in the distribution of $\hat{\sigma}^2$. Even though there is a large amount of overlap, it is clear that the distributions of $\hat{\sigma}^2$ differ depending on classification, where $\hat{\sigma}^2$ associated with incorrect predictions is centred at a larger value. Again, this variant of approximated uncertainty seems to be ascribing a larger amount of uncertainty on average to the incorrectly classified predictions, which is useful.

Upon examining the values of $\hat{\sigma}^2$ associated with the different classes (table 5.6), we see that some are plagued by a number of large outlying values (figs. 5.15a and 5.15b). Interestingly, the classes which are least uncertain in terms of $\hat{\sigma}^2$ seem to have the largest outlying values, skewing the distributions of $\hat{\sigma}^2$ somewhat for the correctly classified images. This could indicate a problematic feature of $\hat{\sigma}^2$; namely that the correct predictions seem to be associated with relatively large uncertainty values. Large outlying values could pose a challenge in situations where we would refer images to a human expert based on average values of $\hat{\sigma}^2$ (see section 5.7). On the other hand, the class-specific values of $\hat{\sigma}^2$ reflect the confusion matrix in fig. 5.1, which indicates that $\hat{\sigma}^2$ is sensitive to when the model fails. Note that $\hat{\sigma}^2$, like $\hat{\sigma}_{\text{pred}}$, is most uncertain about classifying birds on average.

Finally, figs. 5.16 and 5.17 show $\hat{\sigma}^2$ plotted against $\hat{\mu}_{\text{pred}}$. The values of $\hat{\sigma}^2$ do not seem to dip downwards to the same degree as $\hat{\sigma}_{\text{pred}}$ when $\hat{\mu}_{\text{pred}}$ gets smaller. Instead, $\hat{\sigma}^2$ tends to stay more or less within the same range of values before decreasing as $\hat{\mu}_{\text{pred}}$ gets large. As stated earlier, this could be due to the scaling associated with $\hat{\sigma}_{\text{pred}}$. Colour-coding the data by the value of $\hat{\mu}_{\text{run}}$ gives us by and large the same pattern as for $\hat{\sigma}_{\text{pred}}$, namely that the largest values of $\hat{\mu}_{\text{run}}$ tend to cluster around $\hat{\mu}_{\text{pred}}$ and that these pairs are spread over a relatively wide range of values for $\hat{\sigma}^2$.

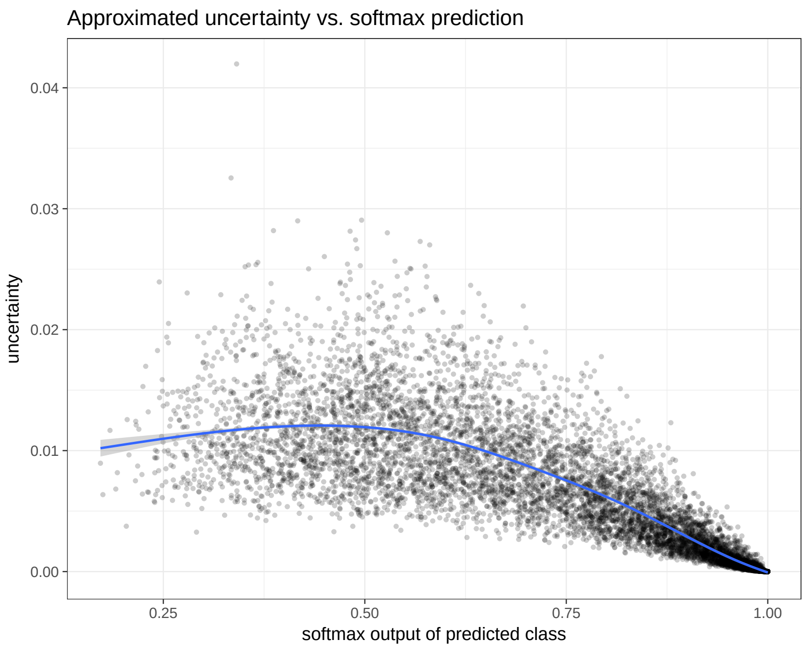


(a) Interestingly, the classes which are least uncertain in terms of $\hat{\sigma}^2$ seem to have the largest outlying values, skewing the distributions of $\hat{\sigma}^2$ somewhat for correctly classified images. Large outlying values could pose a challenge in situations where we would refer images to a human expert.

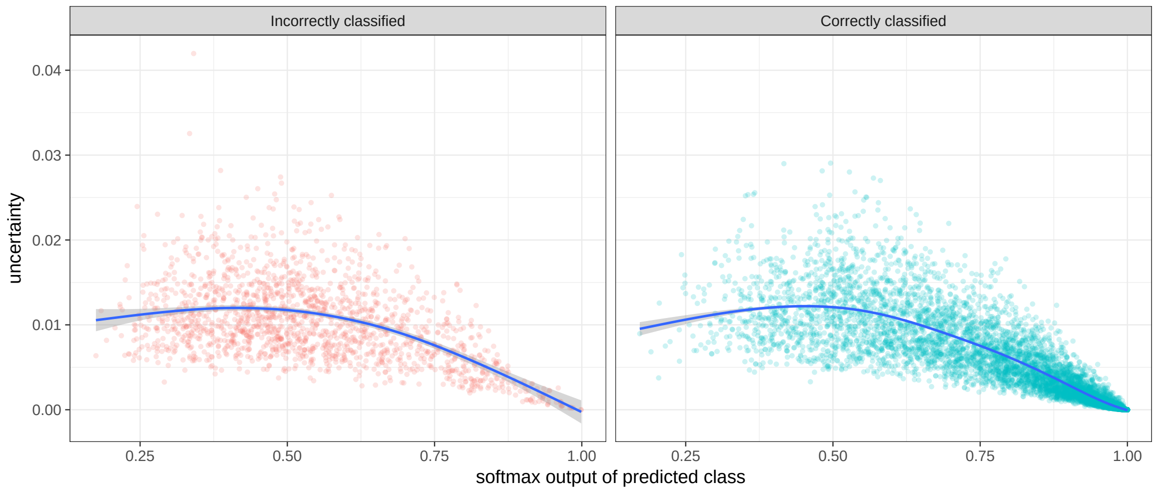


(b) The red box indicates the uncertainty distribution of incorrectly classified images. The blue box corresponds to correctly classified images.

Fig. 5.15: Distributions of class-specific uncertainty (Feinman).

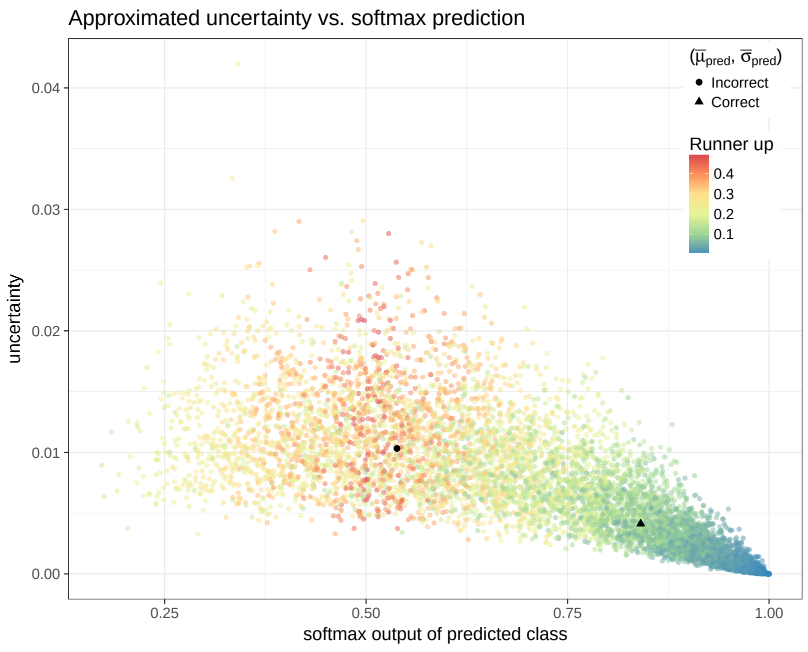


(a) $\hat{\sigma}^2$ does not exhibit the same pronounced concave shape as $\hat{\sigma}_{\text{pred}}$, but this could simply be due to the scaling that follows from taking the square root in $\hat{\sigma}_{\text{pred}}$.

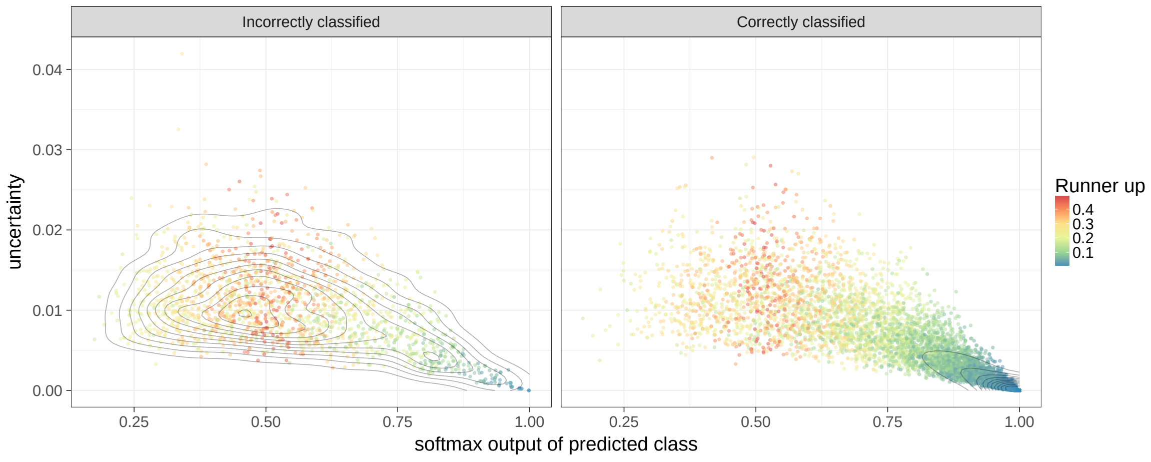


(b) The observations partitioned by classification status.

Fig. 5.16: $\hat{\sigma}^2$ vs. softmax prediction.



(a) Like $\hat{\sigma}_{\text{pred}}$, large pairs of $\hat{\mu}_{\text{pred}}$ and $\hat{\mu}_{\text{run}}$ are spread over a wide range of $\hat{\sigma}^2$.



(b) The observations have been partitioned by classification status and overlayed with a 2D kernel density estimate. The KDE gives an idea of the joint distribution of $\hat{\mu}_{\text{pred}}$ and $\hat{\sigma}_{\text{pred}}$.

Fig. 5.17: Uncertainty (Feinman) vs. softmax prediction and runner-up probabilities.

5.5 A New Approximation of Predictive Uncertainty

In the previous section we briefly explored the empirical properties of $\hat{\sigma}^2$ as defined in [Feinman et al., 2017]. In this section we present a novel approximation of predictive uncertainty, and briefly explore how it compares to $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$. First, let the model uncertainty be defined as

$$\hat{\sigma}_{\text{model}} = \frac{1}{K}\sum_{k=1}^{K}\sqrt{\frac{1}{T}\sum_{t=1}^{T}\left[p(\hat{y}=k|\boldsymbol{x}^*,\hat{\boldsymbol{\omega}}_t)-\hat{\mu}_k\right]^2}.$$

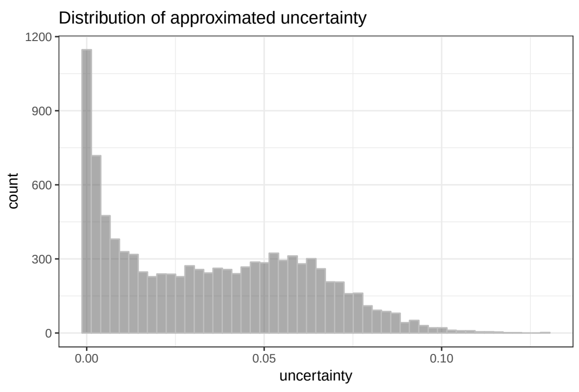
The definition of $\hat{\sigma}_{\text{model}}$ is motivated by a curiosity about what would happen if one were to combine, in a sense, $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$. Our approximated uncertainty differs from $\hat{\sigma}_{\text{pred}}$ in that we take into account the uncertainty associated with every class in the predictive mean $\hat{\boldsymbol{\mu}}$. Furthermore, our approximation differs from $\hat{\sigma}^2$ in that we take the mean of the standard deviations of the class probabilities, instead of the variance.

Table 5.7: Summary statistics for $\hat{\sigma}_{\text{model}}$ for correctly and incorrectly classified images. On average $\hat{\sigma}_{\text{model}}$, like $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$, is higher for mislabelled images.

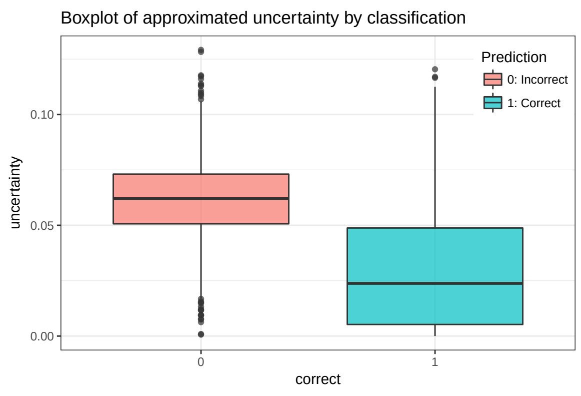
Label	n	avg($\hat{\mu}_{\text{pred}}$)	avg($\hat{\mu}_{\text{run}}$)	avg($\hat{\sigma}_{\text{model}}$)	Median $\hat{\sigma}_{\text{model}}$	IQR
0: Incorrect	1637	0.5388	0.2544	0.0618	0.0620	0.0224
1: Correct	8363	0.8407	0.0967	0.0289	0.0238	0.0435

Table 5.8: Note that the most uncertain class (cat) is also the most frequently misclassified, as opposed to both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$ which associated the highest average uncertainty to images of birds.

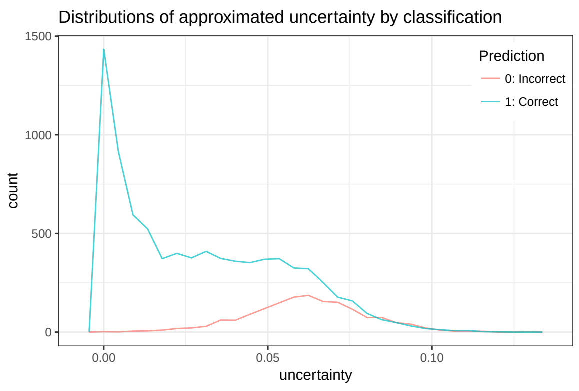
k	n	avg($\hat{\sigma}_k$)
cat	**670**	**0.0430**
bird	742	0.0407
dog	752	0.0325
deer	830	0.0354
airplane	841	0.0335
horse	880	0.0271
frog	898	0.0233
ship	902	0.0208
truck	922	0.0205
automobile	**926**	**0.0198**



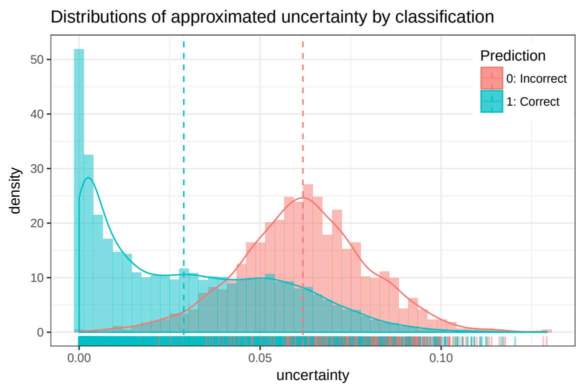
(a)



(b)



(c)

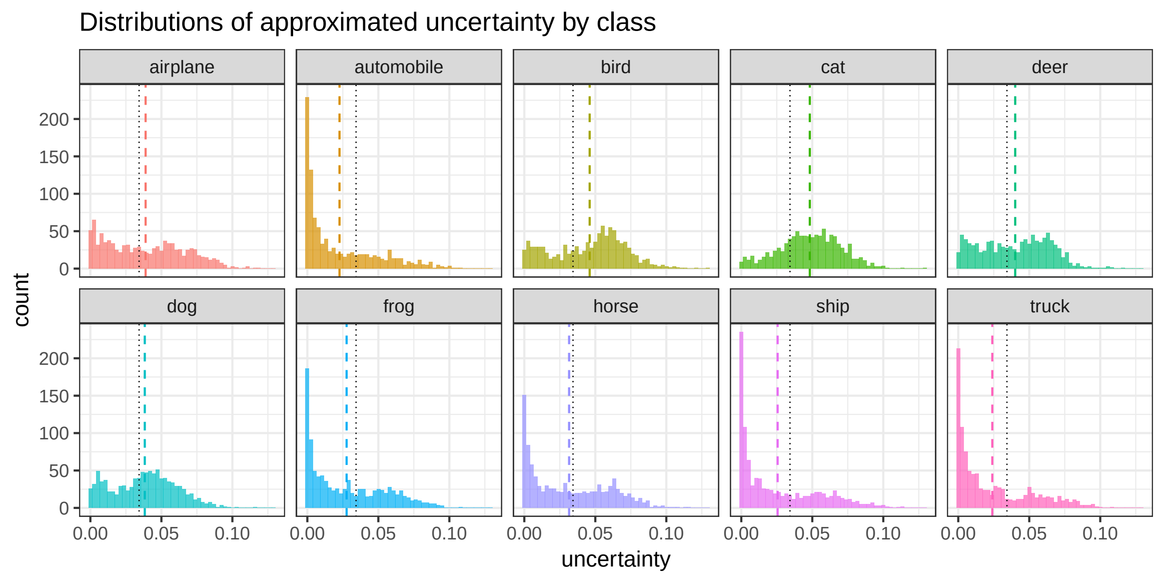


(d)

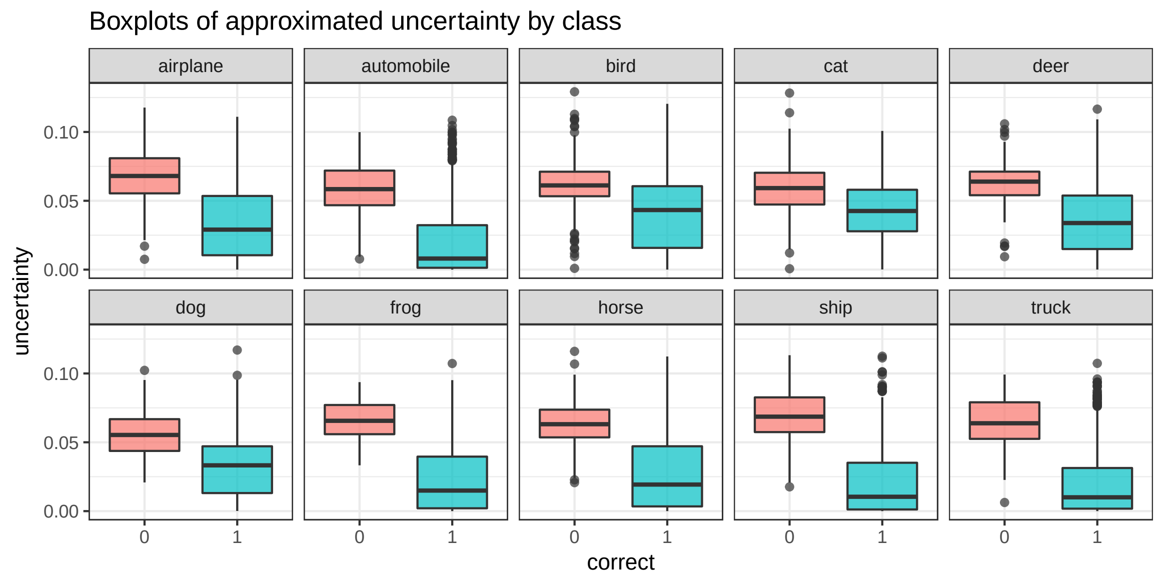
Fig. 5.18: Distribution of $\hat{\sigma}_{\text{model}}$. Like $\hat{\sigma}_{\text{pred}}$, there is a clear difference between the distributions of $\hat{\sigma}_{\text{model}}$ for correctly (blue) and incorrectly (red) labelled images. There is quite a large amount of overlap in the distributions, but compared to $\hat{\sigma}_{\text{pred}}$ there is a less pronounced bimodality, perhaps suggesting less contribution to high uncertainty values from correctly classified images. Also, $\hat{\sigma}_{\text{model}}$ is not affected by outliers to the same degree as $\hat{\sigma}^2$.

The summary statistics for $\hat{\sigma}_{\text{model}}$ are given in table 5.7 and the distribution is shown in fig. 5.18. It is clear the the distribution of $\hat{\sigma}_{\text{model}}$ bears a striking resemblance to that of $\hat{\sigma}_{\text{pred}}$, suggesting that the two different approximations capture some of the same inherent uncertainty in the predictions. However, the bimodality of correct predictions (see fig. 5.18c) seems less pronounced for $\hat{\sigma}_{\text{model}}$, perhaps suggesting less contribution to high uncertainty values from the correctly classified images.

Fig. 5.19 shows the class-specific model uncertainty. Again, it seems that $\hat{\sigma}_{\text{model}}$ captures some of the same class-specific confusion as $\hat{\sigma}_{\text{pred}}$. However, $\hat{\sigma}_{\text{model}}$ is less pronounced for the most uncertain classes, perhaps implying that $\hat{\sigma}_{\text{pred}}$ is more sensitive to the variation in predictions when **lenet-allMC** performs poorly. On the other hand, $\hat{\sigma}_{\text{model}}$ assigns the highest average uncertainty to the most frequently mislabelled class (cat), unlike $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{pred}}$. There are outliers present for the correctly predicted classes, but to a lesser extent than $\hat{\sigma}^2$ (continued on page 105).

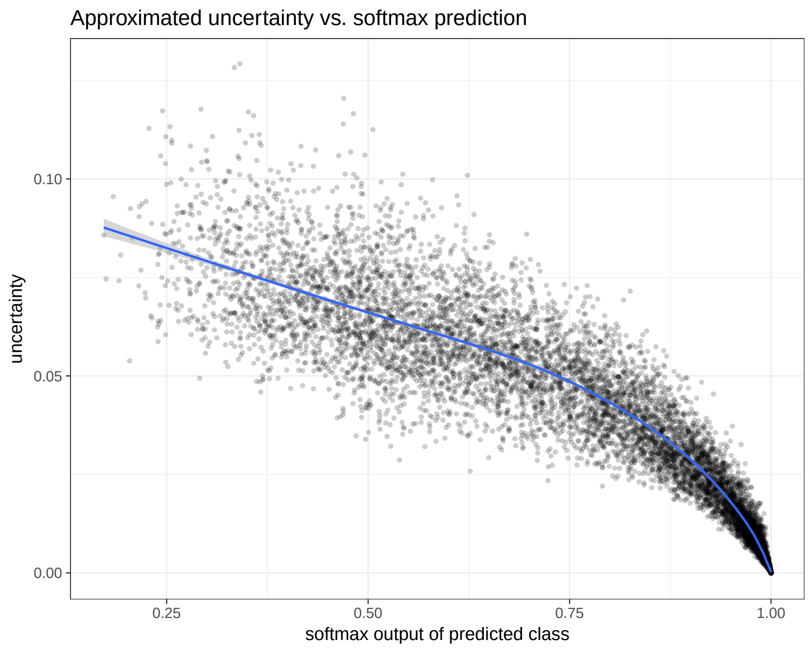


(a) In the most uncertain cases $\hat{\sigma}_{\text{model}}$, like $\hat{\sigma}_{\text{pred}}$, is centred around larger values. $\hat{\sigma}_{\text{model}}$ assigns the highest average uncertainty to cat images, which is the most frequently mislabelled class.

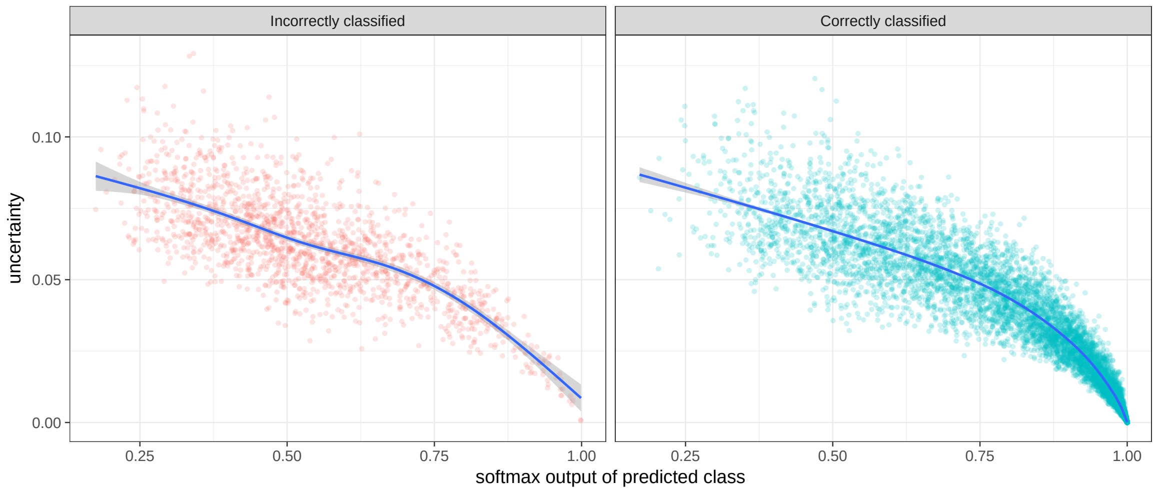


(b) The red box indicates the uncertainty distribution of incorrectly classified images. The blue box corresponds to correctly classified images.

Fig. 5.19: Distributions of class-specific uncertainty (Murray).

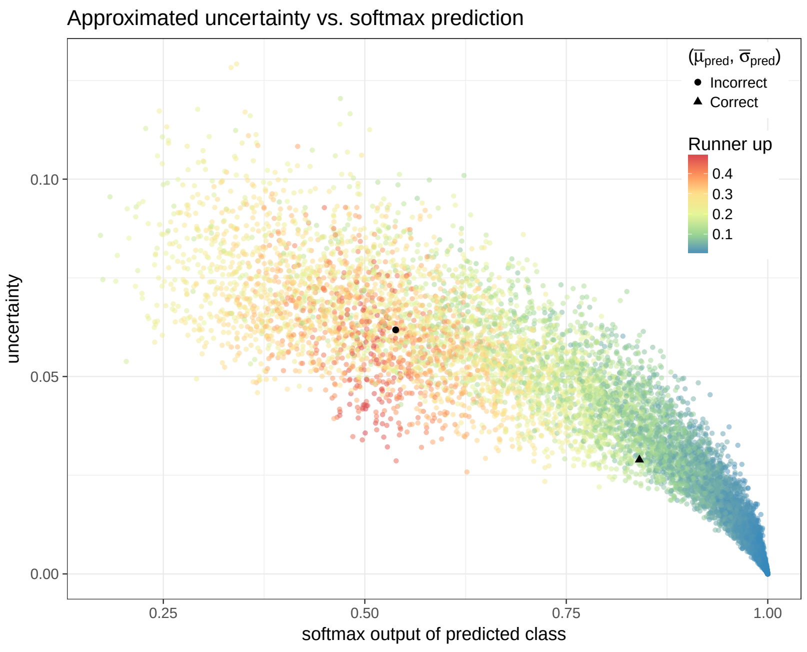


(a) The largest values of $\hat{\sigma}_{\text{model}}$ seem to be associated with the lowest values of $\hat{\mu}_{\text{pred}}$, which distinguishes this uncertainty approximation from both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$.

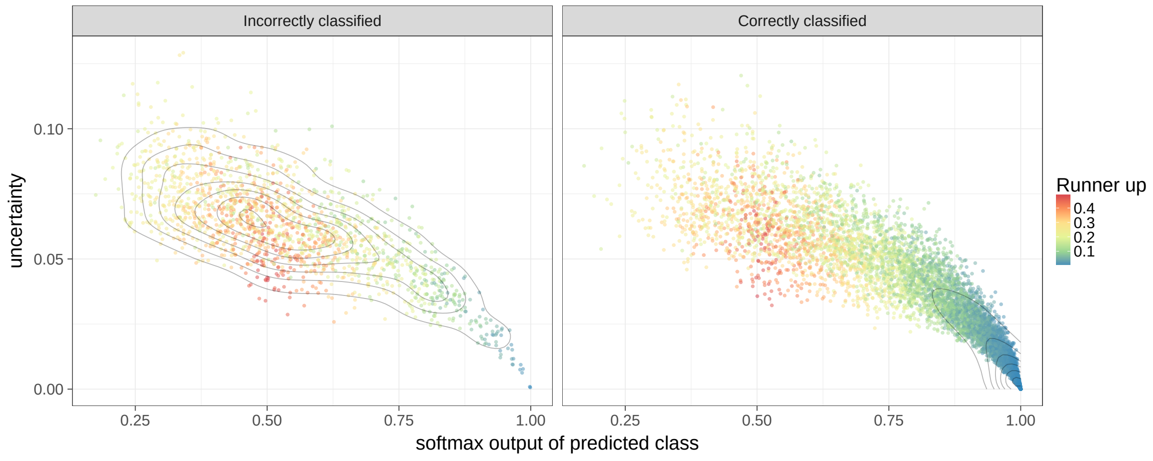


(b) Above we have partitioned the observations by classification status.

Fig. 5.20: $\hat{\sigma}_{\text{model}}$ vs. softmax prediction.



(a) Many large pairs of $(\hat{\mu}_{\text{pred}}, \hat{\mu}_{\text{run}})$ appear to cluster in a small valley associated where $\hat{\sigma}_{\text{model}} \approx 0.05$, which is slightly below the average for incorrect predictions.



(b) The observations are partitioned by classification status and overlayed with a 2D kernel density estimate.

Fig. 5.21: $\hat{\sigma}_{\text{model}}$ vs. softmax prediction and runner up probabilities.

Fig. 5.20 indicates that $\hat{\sigma}_{\text{model}}$ tends to increase as $\hat{\mu}_{\text{pred}}$ decreases. This is more in line with how we would want a reasonable uncertainty estimate to behave, and differs clearly from the parabolic shape of $\hat{\sigma}_{\text{pred}}$ in fig. 5.8 and the flatter shape of $\hat{\sigma}^2$ in fig. 5.20. In this sense, $\hat{\sigma}_{\text{model}}$ provides some information that $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$ do not: The largest values of $\hat{\sigma}_{\text{model}}$ seem to be associated with the lowest values of $\hat{\mu}_{\text{pred}}$. From fig. 5.21 it is apparent that large values of $\hat{\mu}_{\text{run}}$ tends to cluster around $\hat{\mu}_{\text{pred}}$, which was also the case for $\hat{\sigma}_{\text{pred}}$. Observations where $(\hat{\mu}_{\text{pred}}, \hat{\mu}_{\text{run}}) \approx (0.5, 0.5)$ gave a wide range of values for $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$. In the case of $\hat{\sigma}_{\text{model}}$, many large pairs of $(\hat{\mu}_{\text{pred}}, \hat{\mu}_{\text{run}})$ appear to cluster in a small valley associated where $\hat{\sigma}_{\text{model}} \approx 0.05$, which is slightly below the average for incorrect predictions.

5.6 Comparison with Other Uncertainty Measures

Recall from section 4.4.1 that Gal suggests using the predictive entropy (PE, eq. 4.12), the mutual information (MI, eq. 4.13) or the variation-ratio (VR, eq. 4.14) as appropriate approximations of uncertainty in a classification setting. We have estimated these quantities and compared them to $\hat{\sigma}_{\text{pred}}$, $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{model}}$ in fig. 5.22. We would like to draw particular attention to the apparent similarity between $\hat{\sigma}_{\text{model}}$ and the MI. As stated in section 4.4.3, a recent paper by Smith and Gal establishes a connection between the MI and $\hat{\sigma}^2$ [Smith and Gal, 2018], but from a cursory visual inspection it appears that the MI more closely resembles $\hat{\sigma}_{\text{model}}$ than $\hat{\sigma}^2$. Perhaps this could indicate that $\hat{\sigma}_{\text{model}}$ captures some of the same uncertainty as the MI. A more detailed study comparing the uncertainty measures represented by VI, PE and MI and ad hoc approximations such as $\hat{\sigma}_{\text{pred}}$, $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{model}}$ would be an interesting avenue for further research.

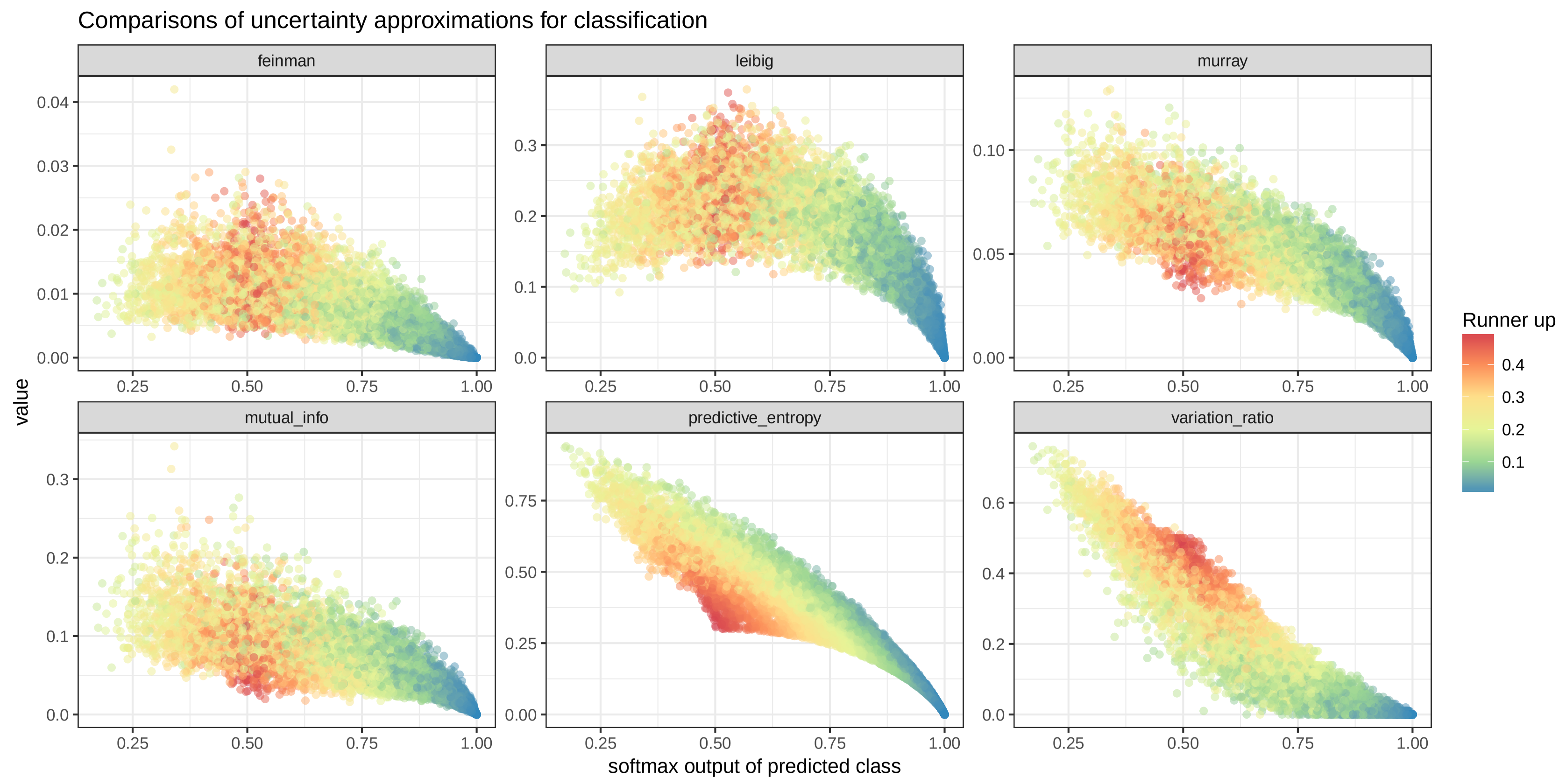


Fig. 5.22: The top row shows the distributions of $\hat{\sigma}^2$, $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}_{\text{model}}$ plotted against $\hat{\mu}_{\text{pred}}$ and colour-coded by the value of $\hat{\mu}_{\text{run}}$. The bottom row does the same for the mutual information (MI), predictive entropy (PE) and variation ratio (VR). Observe the similarity between $\hat{\sigma}_{\text{model}}$ and the MI, perhaps suggesting that $\hat{\sigma}_{\text{model}}$ captures some of the same uncertainty.

5.7 Uncertainty-informed Referral Experiments

In a practical setting, uncertainty estimates could provide useful information when deciding whether or not to refer an image to a human expert. To this end, researchers have suggested different referral schemes based on thresholding the approximated uncertainty at a predetermined value and flagging inputs that exceed this threshold [Feinman et al., 2017; Leibig et al., 2017]. The flagged observations can then be examined downstream with the aim of increasing the overall performance of the network.

In this chapter we will set some simple thresholds using summary statistics of $\hat{\sigma}_{\text{pred}}$, $\hat{\sigma}_{\text{model}}$ and $\hat{\sigma}^2$ in order to explore the practical use of the different uncertainty quantifications. We will also do the same for the VR, PE and MI and compare the results.

5.7.1 Referral criteria

A simple approach to threshold selection is to set the threshold equal to the average uncertainty of the incorrect classifications, and refer any images exceeding this value. Alternatively, we could refer images that exceed the median uncertainty of the incorrectly classified images. In the following we will try both approaches.

The thresholds given in table 5.9 are calculated on a subset of $n = 9000$ randomly selected images from the test data. The thresholds will then be used with their corresponding uncertainty approximation to make uncertainty-informed referrals for the remaining $n = 1000$ images. This is done to ensure that the summary statistics are independent of the uncertainty values of the images we may want to refer.

Table 5.9: Thresholds, based on mean and median of $n = 9000$ randomly selected images.

	Mean	Median
$\hat{\sigma}_{\text{pred}}$	0.20806	0.20727
$\hat{\sigma}_{\text{model}}$	0.06185	0.06211
$\hat{\sigma}^2$	0.01031	0.00980
VR	0.32388	0.34000
MI	0.09535	0.08982
PE	0.50349	0.50628

	Data referred	Incorrect	Correct	Successful referrals
VR	**16.60%**	**90**	**76**	**54.22%**
PE	18.40%	93	91	50.54%
$\hat{\sigma}_{\text{model}}$	**18.90%**	**90**	**99**	**47.62%**
MI	17.40%	81	93	46.55%
$\hat{\sigma}^2$	19.10%	83	108	43.46%
$\hat{\sigma}_{\text{pred}}$	23.50%	89	146	37.87%

Table 5.10: Referral results when using mean threshold on a randomly selected subset of $n = 1000$ images. $\hat{\sigma}_{\text{model}}$ is the best performing ad hoc approximation of predictive uncertainty, but both the variation-ratio (VR) and predictive entropy (PE) perform better.

	Data referred	Incorrect	Correct	Successful referrals
VR	**15.10%**	**80**	**71**	**52.98%**
PE	17.80%	88	90	49.43%
$\hat{\sigma}_{\text{model}}$	**18.50%**	**88**	**97**	**47.57%**
MI	20.70%	93	114	44.92%
$\hat{\sigma}^2$	20.80%	90	118	43.27%
$\hat{\sigma}_{\text{pred}}$	24.00%	93	146	38.75%

Table 5.11: Referral results when using median threshold on a randomly selected subset of $n = 1000$ images. Again, $\hat{\sigma}_{\text{model}}$ outperforms both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$, but both the VR and PE are better for referring images.

5.7.2 Results

The baseline accuracy of the sample is 82%. Tables 5.10 and 5.11 show the results of setting the threshold to the mean and median values of our uncertainty approximations, respectively. Each table summarises the number of incorrectly and correctly classified cases when the thresholds given in table 5.9 are exceeded. Furthermore, we calculate the percentage of referred data (out of $n = 1000$) and the percentage of these referrals that are successful. A success is the case where an incorrectly classified image is referred.

First we will consider the results for $\hat{\sigma}_{\text{pred}}$, $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{model}}$. The difference between the mean and the median thresholds appears to be marginal. Using these simple summary statistics, $\hat{\sigma}_{\text{pred}}$, $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{model}}$ are able to detect close to half of the incorrectly classified images. Our ad hoc approximations refer between 18.5 % and 24 % of the test images, and differ significantly in the amount of correctly classified images referred. Both $\hat{\sigma}^2$ and $\hat{\sigma}_{\text{pred}}$ refer a larger proportion of correctly labelled images than incorrectly labelled images. As noted in section 5.3.1, the distribution of $\hat{\sigma}_{\text{pred}}$ for correctly classified is distinctly bimodal, suggesting that there exists a group of correctly classified images which contribute high uncertainty values. This peak coincides with the average uncertainty value of the incorrectly labelled images, and could explain why $\hat{\sigma}_{\text{pred}}$ refers the largest amount of correctly classified images of the three estimators. As for $\hat{\sigma}^2$, in section 5.4 we noted the presence of large outlying values for correctly classified images. This offers a possible explanation for the amount of correctly labelled referrals when using $\hat{\sigma}^2$. $\hat{\sigma}_{\text{model}}$ appears to reduce the number of correctly classified images, which could be due to the fact that larger values of $\hat{\sigma}_{\text{model}}$ tend to be associated with lower values of $\hat{\mu}_{\text{pred}}$ as mentioned in section 5.5.

It is immediately clear that of the three, $\hat{\sigma}_{\text{model}}$ is the most effective. In the real world, a practical threshold and corresponding uncertainty estimate should ideally refer a manageable amount of images, where as many as possible of these should be incorrect classifications. As a referral tool, $\hat{\sigma}_{\text{model}}$ ticks both boxes for this model and data set, referring the least amount of images while simultaneously having the largest proportion of successful referrals. Note that this result applies to our particular choice of model and data set, and further research is required to verify if this is a general tendency. However, assuming a human expert could correctly identify the incorrectly classified images referred using $\hat{\sigma}_{\text{model}}$, then the error rate could be halved.

On the other hand, none of the ad hoc approximations perform as well as the VR and PE, which Gal suggests for use in a classification setting. This is particularly interesting, because it implies that researchers are developing uncertainty approximations that actually perform worse than established methods already outlined for use in classification by the developers of MC dropout. An interesting avenue for further research would be a more detailed comparison of the performance of ad hoc approximations and VI/PE/MI.

Chapter 6

Experimental Setup

In this chapter we will give a brief overview over the programming languages and dependencies used for model building and analysis in chapter 5. Furthermore we will provide details on the data set (including preprocessing steps), our implementations of LeNet (including training setup) and MC dropout. Finally, we will give some examples of the resulting data from MC dropout.

6.1 Software

We have implemented LeNet using the programming language **Python 3.6** (`https://www.python.org/`) and the deep learning library **Keras** using the **TensorFlow** [Abadi et al., 2016] backend. Keras is a high-level API that emphasises user-friendliness and modularity, allowing for fast prototyping and experimentation. TensorFlow is a machine learning library developed by Google. We have used the scientific computing package **NumPy** in our implementation of MC dropout and the data analysis package **pandas** for data preparation. We have used **Matplotlib** for plotting. The analysis in chapter 5 is mostly done using the statistical programming language **R** (`https://www.r-project.org/`). In R we relied heavily on the **tidyverse** collection of packages for data science [Wickham, 2014]. In particular, we made extensive use of **dplyr** for data wrangling and **ggplot2** for graphics.

6.2 CIFAR-10

Our models were trained on the CIFAR-10 tiny image data set [Krizhevsky and Hinton, 2009]. CIFAR-10 consists of 60.000 labelled 32x32 colour images, each belonging to one of 10 mutually exclusive classes. The classes represented in the data are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The data was split into a training set of 50.000 images and a test set of 10.000 images. The 50.000 training images were further split into a final training and validation set.

6.2.1 Preprocessing

All images were normalised by subtracting the mean and dividing by the standard deviation of the pixel values. Data augmentation was performed using the `ImageDataGenerator` class in Keras. Specifically, we let the images be randomly shifted by up to 10% in the horisontal or vertical directions. In section 5.3.4 we added random noise to the images using the image processing library **scikit-image**. We used the function `random_noise()` to produce seven versions of our test set where each iteration had an increasing amount of additive Gaussian noise. Concretely, we made a new test set for each value $\phi^2 \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.25, 0.5\}$, where ϕ^2 denotes the variance of the Gaussian distribution.

6.3 Models and training

We follow the implementation[1] of LeNet as described in [Gal and Ghahramani, 2015], with the same settings of hyperparameters (momentum $\alpha = 0.9$, weight decay with $\lambda = 0.0005$ and dropout $p = 0.5$) and nonlinear activations (ReLU). There are several different approaches to implementing MC dropout. We have chosen to define the custom layer `DropoutMC()` to preserve the user-friendliness and modularity of Keras. `DropoutMC()` simply applies dropout to whatever input is received during both training and test time.

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Lambda
```

[1] Available at https://github.com/yaringal/DropoutUncertaintyCaffeModels/tree/master/cifar10_uncertainty. Note that Gal's implementation uses the Caffe deep learning framework.

```python
from keras.optimizers import SGD
import keras.backend as K

def Dropout_MC(p):
    layer = Lambda(lambda x: K.dropout(x, p),
                   output_shape=lambda shape: shape)
    return(layer)

def lenet_allMC(input_shape, nb_classes, p=0.5):
    """Bayesian implementation of LeNet using MC dropout.

       Parameters
       ----------
       input_shape : tuple
           Tuple containing image dimension (width, height, depth).
       nb_classes : int
           Number of classes.
       p : float
           Dropout rate (default p=0.5).

       Example
       -------
       >>> img_dims = (32, 32, 3)
       >>> model = lenet_allMC(input_shape=img_dims, nb_classes=10, p=0.5)
    """
    model=Sequential([
        Conv2D(input_shape=input_shape, filters=192, kernel_size=(5,5)),
        Dropout_MC(p),
        MaxPooling2D(strides=2),

        Conv2D(192, kernel_size=(5,5)),
        Dropout_MC(p),
        MaxPooling2D(strides=2),

        Flatten(),
        Dense(1000, activation="relu"),
        Dropout_MC(p),
        Dense(nb_classes, activation="softmax")
    ])
    model.compile(SGD(momentum=0.9, decay=0.0005),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model
```

We have also implemented `lenet()` and `lenet_all()`, two deterministic variants of LeNet, to highlight the differences in test time performance (see section 5.2). They are defined in the following:

```python
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense
from keras.optimizers import SGD

def lenet(input_shape, nb_classes, p=0.5):
    """Standard dropout LeNet. See docstring of lenet_allMC."""
    model=Sequential([
        Conv2D(input_shape=input_shape, filters=192, kernel_size=(5,5)),
        MaxPooling2D(strides=2),

        Conv2D(192, kernel_size=(5,5)),
        MaxPooling2D(strides=2),

        Flatten(),
        Dense(1000, activation="relu"),
        Dropout(p),
        Dense(nb_classes, activation="softmax")
    ])
    model.compile(SGD(momentum=0.9, decay=0.0005),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model

def lenet_all(input_shape, nb_classes, p=0.5):
    """Standard dropout variant of LeNet, but with dropout applied
      after every convolution. See docstring of lenet_allMC."""
    model=Sequential([
        Conv2D(input_shape=input_shape, filters=192, kernel_size=(5,5)),
        Dropout(p),
        MaxPooling2D(strides=2),

        Conv2D(192, kernel_size=(5,5)),
        Dropout(p),
        MaxPooling2D(strides=2),

        Flatten(),
        Dense(1000, activation="relu"),
        Dropout(p),
        Dense(nb_classes, activation="softmax")
    ])
    model.compile(SGD(momentum=0.9, decay=0.0005),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model
```

`lenet_allMC()` and `lenet()` were trained for 100 epochs using a batch size of 128. This gave results comparable to the those in [Gal and Ghahramani, 2015]. The weights corresponding to the best validation scores were saved for both models, and are the weights used at test time. Furthermore, the weights obtained from `lenet_allMC()` were used in `lenet_all()`, since the two are identical at training time. During training we employed the same learning rate schedule as defined in [Gal and Ghahramani, 2015]:

```python
from keras.callbacks import LearningRateScheduler

init_lr = 0.01  # Initial learning rate
gamma = 0.0001
p = 0.75

def schedule(epoch):
    """Function determining the learning rate annealing schedule."""
    return init_lr * (1 + gamma * epoch) ** (-p)

lr_schedule = LearningRateScheduler(schedule)
```

6.4 Inference

To perform MC dropout we have defined the function `inference()` which outputs a Python dictionary containing

- the unnormalised image (for plotting),
- a $(T \times K)$ matrix of softmax predictions,
- a $(1 \times K)$ vector of predictive means,
- the predicted class,

- the true class,
- our three approximations of predictive uncertainty,
- an indicator variable which is 1 if the true class equals the predicted class and 0 otherwise.

We applied `inference()` to all images in the test set and pickled the results using Python's **pickle** module. The function is defined in the following:

```python
import numpy as np

def batch(img, T):
    ''' Creates mini-batch of T identical images for use in inference.'''
    img_batch = np.array([img for t in range(T)])
    return(img_batch)

def inference(model, X, y, T=100, normalise=False):
    '''Function that performs Bayesian inference by applying MC dropout
       with T stochastic forward passes on given input.

       Parameters
       ----------
       model : object
           A Keras model.
       X : array-like
           Test data to be classified.
       y : array-like
           Labels associated with test data.
       T : int
           Number of stochastic forward passes (default T=100).
       normalise : boolean
           Set to True if the input is to be normalised (default normalise=False)


       Example
       -------
       >>> img_dims = (32, 32, 3)
       >>> model = lenet_allMC(input_shape=img_dims, nb_classes=10, p=0.5)
       >>> results = inference(model, X_test, y_test, normalise=True)
       '''
    # Get images, labels
    imgs, labels = X, y.argmax(axis=1)

    if normalise:
        # Preprocess input
        imgs = imgs.astype("float32")
        imgs -= np.mean(X)
        imgs /= np.std(X, axis=0)

    # Empty dictionary to store all output
    output = {}

    # Iterator index to keep track of images in dictionary
    k=0

    # Gathering MC samples
    for (img, label, x) in list(zip(imgs, labels, X)):

        # Get image batch
        img_batch = batch(img, T)

        # T x K matrix of softmax predictions
        results = model.predict(img_batch, batch_size=T)

        # Gathering results
        probs = results
        probs_mean = np.mean(probs, axis=0)     # Vector of pred. means
        pred_std = np.std(probs, axis=0)        # Vector of std. dev.
        mean_std = pred_std.mean()              # Our uncertainty
        mean_var = np.var(probs, axis=0).mean() # Feinman uncertainty
        prediction = probs_mean.argmax()        # Prediction
        uncertainty = pred_std[prediction]      # Leibig uncertainty
        correct = 1 if prediction == label else 0

        output[k] = {
            "img": x,
            "softmax_dist": probs,
            "probs": probs_mean,
            "prediction": prediction,
            "truth": label,
            "leibig_uncertainty": uncertainty,
            "murray_uncertainty": mean_std,
            "feinman_uncertainty": mean_var,
            "correct": correct
        }

        k+=1

    return output
```

6.5 Data Analysis in R

The resulting data obtained from `inference()` was converted to a pandas dataframe, where simple feature engineering (such as retrieving $\hat{\mu}_{\text{pred}}$ and $\hat{\mu}_{\text{run}}$) was performed. Finally, we exported the data a CSV-file for further analysis using R. Other quantities, such as the predictive entropy (PE), variation-ratio (VR) and mutual information[2] (MI), were calculated on `softmax_dist` retrieved from the pickled data and joined with the data set when needed. Table 6.1 shows an example of five randomly sampled rows from the data (excluding the PE, VR and MI):

ID	Prediction	Truth	Correct	Runner-up	$\hat{\mu}_{\text{pred}}$	$\hat{\mu}_{\text{run}}$	$\hat{\sigma}_{\text{model}}$	$\hat{\sigma}_{\text{pred}}$	$\hat{\sigma}^2$
3372	2	2	1	5	0.7677	0.1734	0.0522	0.2336	0.0097
9648	3	3	1	5	0.4231	0.3122	0.0834	0.2076	0.0149
7212	3	3	1	5	0.6669	0.2257	0.0484	0.1791	0.0060
1705	5	3	0	3	0.4374	0.3503	0.0727	0.2407	0.0124
5208	8	8	1	2	0.7936	0.0670	0.0530	0.1898	0.0062

Table 6.1: The above table shows five randomly sampled data points from the results on our test set ($N = 10.000$). The *ID* column keeps track of the specific image. *Prediction* and *Truth* indicate the predicted label and the ground truth label of a given image. If they are the same, then *Correct* will be equal to 1, and 0 otherwise. The *Runner-up* column indicates which class has the second-highest softmax probability in the final predictive mean vector. Finally, the last five columns summarise the statistics of interest.

[2]The formulas used to calculate the PE, VR and MI are given in section 4.4.1.

Chapter 7

Conclusion

7.1 Summary

In this thesis we have explored a relatively recent approach to estimating predictive uncertainty in dropout neural networks. In part 1 we gave an overview of the necessary background theory. In chapter 2 we highlight some important terms and ideas from machine learning. In chapter 3 we introduced neural networks in general and convolutional neural networks specifically. In chapter 4 we presented some important methods from Bayesian statistics which are the building blocks for MC dropout, the method that allows us to obtain uncertainty estimates from dropout neural networks. We also saw that MC dropout has been extended to Bayesian convolutional neural networks (BCNNs), which appears to boost accuracy at the cost of losing the Gaussian process approximation interpretation. Finally we reviewed practical applications of MC dropout, focusing on ad hoc uncertainty approximations in BCNNs.

In part 2 we examined the ad hoc approximations of predictive uncertainty used in recent research, with particular focus on extending the work in [Leibig et al., 2017] to a multi-class setting. We extended $\hat{\sigma}_{\text{pred}}$ presented by Leibig and colleagues from a binary setting to a multi-class setting using the CIFAR-10 dataset of labelled images, and investigated how $\hat{\sigma}_{\text{pred}}$ relates to the predictions of a simple convolutional neural network. Despite being somewhat ad hoc, $\hat{\sigma}_{\text{pred}}$ seems to capture a meaningful measure of uncertainty at test time. $\hat{\sigma}_{\text{pred}}$ appears to reflect class-specific uncertainties in a way that mirrors the confusion matrix, but we also see that a model can output low values for both $\hat{\sigma}_{\text{pred}}$ and $\hat{\mu}_{\text{pred}}$. Ideally we would want $\hat{\sigma}_{\text{pred}}$ to be large when $\hat{\mu}_{\text{pred}}$ is small, for this model and data set that does not seem to be the case. Furthermore, we saw that $\hat{\sigma}_{\text{pred}}$ increased in response to additive noise, which is what we would expect from a useful quantification of uncertainty. Finally we investigated what effect varying the dropout rate p at test time has on model accuracy and approximated uncertainty. We saw that model accuracy was quite high for relatively large values of p, and $\hat{\sigma}_{\text{pred}}$ increased as p increased.

While exploring $\hat{\sigma}_{\text{pred}}$ we became aware of the work in [Feinman et al., 2017], which can be interpreted as a quantification of uncertainty in a multi-class setting. We briefly explored $\hat{\sigma}^2$, which takes into account the variance in the predictions of every class in $\hat{\boldsymbol{\mu}}$. We saw that $\hat{\sigma}^2$ displays many of the same characteristics as $\hat{\sigma}_{\text{pred}}$, but with more outlying observations and a less pronounced paraboloid shape (possibly due to scaling) compared to $\hat{\mu}_{\text{pred}}$. Again, $\hat{\sigma}^2$ reflects the confusion matrix of **lenet-allMC** at test time, indicating that some sort of a meaningful uncertainty estimate is provided. However, the presence of outlying values of $\hat{\sigma}^2$ among the correctly classified labels could indicate a shortcoming of this particular uncertainty approximation.

In section 5.5 we introduced a novel variation of approximated predictive uncertainty, denoted by $\hat{\sigma}_{\text{model}}$, which averages the standard deviations of the predictions in the predictive mean vector of class probabilities. We saw that this measure bears a stronger resemblance to $\hat{\sigma}_{\text{pred}}$, but differs favourably in that it appears to increase as $\hat{\mu}_{\text{pred}}$ decreases.

Next, we briefly examined how the ad hoc approximations compare to other measures of uncertainty which were recommended in [Gal, 2016] for use in classification. Recent research [Smith and Gal, 2018] shows that $\hat{\sigma}^2$ can be viewed as the first term in a Taylor expansion of the mutual information (MI). However, upon visual inspection, we saw that $\hat{\sigma}_{\text{model}}$ and the mutual information score appear much more similar. This could perhaps suggest that these to quantities capture some of the same uncertainty.

In section 5.7 we tested uncertainty-based referrals using thresholds based on the average and median uncertainty of a subsample of data, where $\hat{\sigma}_{\text{model}}$ outperformed both $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$. However, the Gal's suggestions to quantifying uncertainty in classification tasks, here represented by the variation-ratio (VR) and the predictive entropy (PE), outperform all the ad hoc approximations in our simple referral experiment, perhaps suggesting that these should be used instead.

7.2 Future Work

There are several interesting avenues for further research:

- Are ad hoc approximations of predictive uncertainty necessary? Can conventional measures of uncertainty, such as the VR, PE and MI, be used instead? Can we identify any settings where ad hoc approximations give better results?
- How do we find useful uncertainty thresholds? It is conceivable that simple summary statistics such as the mean and median of incorrectly labelled images fail to capture subtle differences in predictive uncertainty. Could we get better referrals by taking other statistics, such as $\hat{\mu}_{\text{pred}}$, into account?
- How do the different ad hoc approximations of uncertainty respond to more advanced models and data sets?
- What causes $\hat{\sigma}_{\text{model}}$ to behave differently with respect to $\hat{\mu}_{\text{pred}}$, as compared to $\hat{\sigma}_{\text{pred}}$ and $\hat{\sigma}^2$? What, if any, is the connection between $\hat{\sigma}_{\text{model}}$ and the MI score?
- Could we develop an optimal MC dropout rate, such that the accuracy of the model is as high as possible while maintaining a maximum separation between predictive uncertainty for correctly and incorrectly classified images (as outlined in section 5.3.5)? In other words, could a “dropout-finder” be developed?

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877.

Breiman, L. et al. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231.

Casella, G. and Berger, R. L. (2002). *Statistical inference*, volume 2. Duxbury Pacific Grove, CA.

Cubuk, E. D., Zoph, B., Mané, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Damianou, A. and Lawrence, N. (2013). Deep Gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Feinman, R., Curtin, R. R., Shintre, S., and Gardner, A. B. (2017). Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*.

Gal, Y. (2016). *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge.

Gal, Y. and Ghahramani, Z. (2015). Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference. *arXiv preprint arXiv:1506.02158*.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1050–1059, New York, New York, USA. PMLR.

Gelman, A., Robert, C., Chopin, N., and Rousseau, J. (2013). *Bayesian Data Analysis*. Chapman and Hall/CRC Texts in Statistical Science, third edition.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Gonzalez, R. C. and Woods, R. E. (2017). *Digital image processing*. Pearson, fourth edition.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Guo, C., Pleiss, G., Sun, Y., and Weinberger, K. Q. (2017). On calibration of modern neural networks. *arXiv preprint arXiv:1706.04599*.

Han, J. and Moraga, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*, pages 195–201. Springer.

Hanley, J. A. and McNeil, B. J. (1982). The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2014). *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated.

Karpathy, A. (2016). Cs231n: Convolutional neural networks for visual recognition. *Retrieved from http://cs231n.github.io.*

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404.

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer.

Leibig, C., Allken, V., Ayhan, M. S., Berens, P., and Wahl, S. (2017). Leveraging uncertainty information from deep neural networks for disease detection. *Scientific reports*, 7(1):17816.

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400.*

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3.

MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472.

MacKay, D. J. C. (2002). *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, New York, NY, USA.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective.* The MIT Press.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks.* Springer-Verlag New York, Inc.

Pawitan, Y. (2013). *In All Likelihood: Statistical Modelling and Inference Using Likelihood.* Oxford University Press.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

Rashmi, K. and Gilad-Bachrach, R. (2015). Dart: Dropouts meet multiple additive regression trees. In *International Conference on Artificial Intelligence and Statistics*, pages 489–497.

Rasmussen, C. E. and Williams, C. K. I. (2005). *Gaussian Processes for Machine Learning*. The MIT Press.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

Silverman, B. W. (2018). *Density estimation for statistics and data analysis*. Routledge.

Smith, L. and Gal, Y. (2018). Understanding Measures of Uncertainty for Adversarial Example Detection. *arXiv preprint arXiv:1803.08533*.

Smith, L. N. (2015). Cyclical Learning Rates for Training Neural Networks. *arXiv preprint arXiv:1506.01186*.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 464–472. IEEE.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82.

Appendix A

Derivations

A.1 Bias-variance decomposition

We derive the bias-variance decomposition in example 2.3 by expanding the terms

$$\begin{aligned}
\mathbb{E}[(y^* - \hat{f}(x^*))^2] &= \mathbb{E}[y^{*2} - 2y^*\hat{f}(x^*) + \hat{f}(x^*)^2] \\
&= \mathbb{E}[y^{*2}] - 2\mathbb{E}[y^*\hat{f}(x^*)] + \mathbb{E}[\hat{f}(x^*)^2] \\
&= \mathbb{E}[(f(x^*) + \epsilon)^2] - 2\mathbb{E}[(f(x^*) + \epsilon)\hat{f}(x^*)] + \mathbb{E}[\hat{f}(x^*)^2] \\
&= f(x^*)^2 + \sigma_\epsilon^2 - 2f(x^*)\mathbb{E}[\hat{f}(x^*)] + \mathbb{E}[\hat{f}(x^*)^2] \\
&= f(x^*)^2 + \sigma_\epsilon^2 - 2f(x^*)\mathbb{E}[\hat{f}(x^*)] + \mathbb{E}[\hat{f}(x^*)^2] \\
&\qquad + \mathbb{E}[\hat{f}(x^*)]^2 - \mathbb{E}[\hat{f}(x^*)]^2 \\
&= \sigma_\epsilon^2 + (\mathbb{E}[\hat{f}(x^*)] - f(x^*))^2 + \mathbb{E}[\hat{f}(x^*)^2] - \mathbb{E}[\hat{f}(x^*)]^2 \\
&= \sigma_\epsilon^2 + \mathrm{Bias}^2[\hat{f}(x^*)] + \mathrm{Var}[\hat{f}(x^*)],
\end{aligned}$$

where we have used $\mathbb{E}(\epsilon) = 0$, $\mathrm{Var}(\epsilon) = \sigma_\epsilon^2$ and

$$\begin{aligned}
\mathrm{Var}[\hat{f}(x^*)] &= \mathbb{E}[\hat{f}(x^*)^2] - (\mathbb{E}[\hat{f}(x^*)])^2, \\
\mathrm{Bias}[\hat{f}(x^*)] &= \mathbb{E}[\hat{f}(x^*)] - f(x^*).
\end{aligned}$$

A.2 Kullback-Leibler divergence

To retreive the terms of the evidence lower bound (ELBO) as described in eq. 4.4, observe that the KL divergence can be rewritten as

$$\begin{aligned}
\mathrm{KL}(q_\phi(\boldsymbol{w}) \| p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})) &= \int q_\phi(\boldsymbol{w}) \log \frac{q_\phi(\boldsymbol{w})}{\frac{p(\boldsymbol{w})p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w})}{p(\boldsymbol{y}|\boldsymbol{X})}} d\boldsymbol{w} \\
&= \int q_\phi(\boldsymbol{w}) \log q_\phi(\boldsymbol{w}) d\boldsymbol{w} - \int q_\phi(\boldsymbol{w}) \log \frac{p(\boldsymbol{w})p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w})}{p(\boldsymbol{y}|\boldsymbol{X})} d\boldsymbol{w} \\
&= \int q_\phi(\boldsymbol{w}) \log \frac{q_\phi(\boldsymbol{w})}{p(\boldsymbol{w})} d\boldsymbol{w} - \int q_\phi(\boldsymbol{w}) \log p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w}) d\boldsymbol{w} \\
&\qquad + \int q_\phi(\boldsymbol{w}) \log p(\boldsymbol{y}|\boldsymbol{X}) d\boldsymbol{w} \\
&= \mathrm{KL}(q_\phi(\boldsymbol{w})) \| p(\boldsymbol{w})) - \mathbb{E}_{q_\phi(\boldsymbol{w})}[\log p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w})] + \log p(\boldsymbol{y}|\boldsymbol{X}),
\end{aligned}$$

where we have used

$$\begin{aligned}
&\log\left(\frac{a}{b}\right) = \log a - \log b, \\
&\log(ab) = \log a + \log b, \\
&\mathbb{E}_{q_\phi(\boldsymbol{w})}[\log p(\boldsymbol{y}|\boldsymbol{X})] = \log p(\boldsymbol{y}|\boldsymbol{X}),
\end{aligned}$$

in last two equalities.

A.3 Multivariate Normal Distribution

In example 4.2 we used properties of the conditional distribution of a multivariate Gaussian. We follow the notation provided in section 4.3 of [Murphy, 2012]. A vector $\boldsymbol{x} \in \mathbb{R}^D$ follows a multivariate normal distribution if

$$f(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\intercal \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right), \tag{A.1}$$

where $\boldsymbol{\mu} = \mathbb{E}(\boldsymbol{x}) \in \mathbb{R}^D$ is the mean vector, $\boldsymbol{\Sigma}$ is the $D \times D$ covariance matrix and $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. In this case, we write $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

Let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be sub-vectors of $\boldsymbol{x}$. Then

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix} \right), \tag{A.2}$$

where $\boldsymbol{\Sigma}_{ij} = \mathrm{Cov}(\boldsymbol{x}_i, \boldsymbol{x}_j)$. The marginals are given by $\boldsymbol{x}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_{ii})$ for $i = 1, 2$. The conditional distribution of $\boldsymbol{x}_2$ given $\boldsymbol{x}_1$ is

$$\boldsymbol{x}_2 | \boldsymbol{x}_1 \sim \mathcal{N}(\boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*), \tag{A.3}$$

where

$$\boldsymbol{\mu}^* = \boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}(\boldsymbol{x}_1 - \boldsymbol{\mu}_1), \tag{A.4}$$

$$\boldsymbol{\Sigma}^* = \boldsymbol{\Sigma}_{22} - \boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}\boldsymbol{\Sigma}_{12}. \tag{A.5}$$

For a complete proof of this, refer to section 4.3.4 of [Murphy, 2012].