Chapter 1

Introduction: The Importance of Knowing What We Don't Know

In the *Bayesian machine learning* community we work with probabilistic models and uncertainty. Models such as Gaussian processes, which define probability distributions over functions, are used to learn the more likely and less likely ways to generalise from observed data. This probabilistic view of machine learning offers confidence bounds for data analysis and decision making, information that a biologist for example would rely on to analyse her data, or an autonomous car would use to decide whether to brake or not. In analysing data or making decisions, it is often necessary to be able to tell whether a model is certain about its output, being able to ask "maybe I need to use more diverse data? or change the model? or perhaps be careful when making a decision?". Such questions are of fundamental concern in Bayesian machine learning, and have been studied extensively in the field [Ghahramani, 2015]. When using *deep learning models* on the other hand [Goodfellow et al., 2016], we generally only have point estimates of parameters and predictions at hand. The use of such models forces us to sacrifice our tools for answering the questions above, potentially leading to situations where we can't tell whether a model is making sensible predictions or just guessing at random.

Most deep learning models are often viewed as *deterministic functions*, and as a result viewed as operating in a very different setting to the probabilistic models which possess uncertainty information. Perhaps for this reason it is quite surprising to see how close modern deep learning is to probabilistic modelling. In fact, we shall see that we can get uncertainty information from existing deep learning models for free—without changing

a thing. The main goal of this thesis is to develop such practical tools to reason about uncertainty in deep learning.

1.1 Deep learning

To introduce deep learning, I shall start from the simplest of the statistical tools: *linear regression* [Gauss, 1809; Legendre, 1805; Seal, 1967]. In linear regression we are given a set of N input-output pairs $\{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_N, \mathbf{y}_N)\}$, for example CO_2-temperature observations, or the average number of accidents for different driving speeds. We assume that there exists a *linear* function mapping each $\mathbf{x}_i \in \mathbb{R}^Q$ to $\mathbf{y}_i \in \mathbb{R}^D$ (with $\mathbf{y}_i$ potentially corrupted with observation noise). Our *model* in this case is a linear transformation of the inputs: $\mathbf{f}(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$, with $\mathbf{W}$ some Q by D matrix over the reals and $\mathbf{b}$ a real vector with D elements. Different parameters $\mathbf{W}, \mathbf{b}$ define different linear transformations, and our aim is to find parameters that, for example, would minimise the average squared error over our observed data: $\frac{1}{N}\sum_i ||\mathbf{y}_i - (\mathbf{x}_i\mathbf{W} + \mathbf{b})||^2$.

In more general cases where the relation between $\mathbf{x}$ and $\mathbf{y}$ need not be linear, we may wish to define a *non-linear* function $\mathbf{f}(\mathbf{x})$ mapping the inputs to the outputs. For this we can resort to *linear basis function regression* [Bishop, 2006; Gergonne, 1815; Smith, 1918], where the input $\mathbf{x}$ is fed through K fixed scalar-valued non-linear transformations $\phi_k(\mathbf{x})$ to compose a *feature* vector $\Phi(\mathbf{x}) = [\phi_1(\mathbf{x}), ..., \phi_K(\mathbf{x})]$. We then perform linear regression with this vector instead of $\mathbf{x}$ itself. The transformations ϕ_k are our basis functions, and with scalar input x, these can be wavelets parametrised by k, polynomials of different degrees x^k, or sinusoidals with various frequencies. When $\phi_k(\mathbf{x}) := x_k$ and $K = Q$, basis function regression reduces to linear regression. The basis functions are often assumed to be fixed and orthogonal to each other, and an optimal combination of these functions is sought.

Relaxing the constraint for the basis functions to be fixed and mutually orthogonal, we can use *parametrised* basis functions instead [Bishop, 2006]. For example, we may define the basis functions to be $\phi_k^{\mathbf{w}_k, b_k}$ where the scalar-valued function ϕ_k is applied to the inner-product $\langle \mathbf{w}_k, \mathbf{x} \rangle + b_k$. In this case ϕ_k are often defined to be identical for all k, for example $\phi_k(\cdot) = \sin(\cdot)$ giving $\phi_k^{\mathbf{w}_k, b_k}(\mathbf{x}) = \sin(\langle \mathbf{w}_k, \mathbf{x} \rangle + b_k)$. The feature vector composed of the basis functions' output is again fed as the input to a linear transformation. The model output can then be written more compactly as $\mathbf{f}(\mathbf{x}) = \Phi^{\mathbf{W}_1, \mathbf{b}_1}(\mathbf{x})\mathbf{W}_2 + \mathbf{b}_2$ with $\Phi^{\mathbf{W}_1, \mathbf{b}_1}(\mathbf{x}) = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$, $\mathbf{W}_1$ a matrix of dimensions Q by K, $\mathbf{b}_1$ a vector with K elements, $\mathbf{W}_2$ a matrix of dimensions K by D, and $\mathbf{b}_2$ a vector with D elements. To

perform regression now we can find $\mathbf{W}_1, \mathbf{b}_1$ as well as $\mathbf{W}_2, \mathbf{b}_2$ that minimise the average squared error over our observed data, $||\mathbf{y} - \mathbf{f}(\mathbf{x})||^2$.

The most basic model in *deep learning* can be described as a hierarchy of these parametrised basis functions (such a hierarchy is referred to as a *neural network* for historical reasons, with each feature vector in the hierarchy referred to as a *layer*). In the simplest setting of regression we would simply compose multiple basis function regression models, and for classification we would further compose a logistic function at the end (which "squashes" the linear model's output to obtain a probability vector). Each layer in the hierarchy can be seen as a "building block", and the modularity in the composition of such blocks embodies the versatility of deep learning models. The simplicity of each block, together with the many possibilities of model combinations, might be what led many engineers to work in the field. This in turn has led to the development of tools that *work well* and *scale well*.

We will continue with a review of simple neural network models, relating the notation in the field of deep learning to the mathematical formalism of the above linear basis function regression models. We then extend these to specialised models designed to process image data and sequence data. In the process we will introduce some of the terminology and mathematical notation used throughout the work. We will describe the models formally but succinctly, which will allow us to continue our discussion in the introduction using a more precise language.

Feed-forward neural networks (NNs). We will first review a neural network model [Rumelhart et al., 1985] for a *single hidden layer*. This is done for ease of notation, and the generalisation to multiple layers is straightforward. We denote by $\mathbf{x}$ the model input (referred to as *input layer*, a row vector with Q elements), and transform it with an affine transformation to a row vector with K elements. We denote by $\mathbf{W}_1$ the linear map (referred to as a *weight matrix*) and by $\mathbf{b}$ the translation (referred to as a *bias*) used to transform the input $\mathbf{x}$ to obtain $\mathbf{x}\mathbf{W}_1 + \mathbf{b}$. An element-wise non-linearity $\sigma(\cdot)$ (such as the rectified linear[1] (ReLU) or TanH) is then applied to the transformation output, resulting in a *hidden layer* with each element referred to as a *network unit*. This is followed by a second linear transformation with weight matrix $\mathbf{W}_2$ mapping the hidden layer to the model output (referred to as *output layer*, a row vector with D elements). These two layers are also referred to as *inner-product layers*. We thus have $\mathbf{W}_1$ is a $Q \times K$ matrix, $\mathbf{W}_2$ is a $K \times D$ matrix, and $\mathbf{b}$ is a K dimensional vector. A standard

[1] $\text{relu}(x) = \max(x, 0)$.

network would output

$$\widehat{\mathbf{y}} = \sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b})\mathbf{W}_2$$

given some input[2] $\mathbf{x}$.

To use the network for regression we might use the Euclidean loss,

$$E^{\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}}(\mathbf{X},\mathbf{Y}) = \frac{1}{2N}\sum_{i=1}^{N}||\mathbf{y}_i - \widehat{\mathbf{y}}_i||^2 \tag{1.1}$$

where $\{\mathbf{y}_1,...,\mathbf{y}_N\}$ are N observed outputs, and $\{\widehat{\mathbf{y}}_1,...,\widehat{\mathbf{y}}_N\}$ are the outputs of the model with corresponding observed inputs $\{\mathbf{x}_1,...,\mathbf{x}_N\}$. Minimising this loss w.r.t. $\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}$ would hopefully result in a model that can generalise well to unseen test data $\mathbf{X}_{\text{test}}, \mathbf{Y}_{\text{test}}$.

To use the model for classification, predicting the probability of $\mathbf{x}$ being classified with a label in the set $\{1,...,D\}$, we pass the output of the model $\widehat{\mathbf{y}}$ through an element-wise softmax function to obtain normalised scores: $\widehat{p}_d = \exp(\widehat{y}_d)/\left(\sum_{d'}\exp(\widehat{y}_{d'})\right)$. Taking the log of $\widehat{p}_d$ (with d being the observed label) results in a *softmax* loss,

$$E^{\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}}(\mathbf{X},\mathbf{Y}) = -\frac{1}{N}\sum_{i=1}^{N}\log(\widehat{p}_{i,d_i}) \tag{1.2}$$

where $d_i \in \{1,2,...,D\}$ is the observed class for input i.

A big difficulty with the models above is their tendency to overfit—decrease their loss on the training set $\mathbf{X}, \mathbf{Y}$ while increasing their loss on the test set $\mathbf{X}_{\text{test}}, \mathbf{Y}_{\text{test}}$. For this reason a regularisation term is often added during optimisation. We often use L_2 regularisation for each parameter weighted by some weight decays λ_i, resulting in a minimisation objective (often referred to as a *cost*),

$$\mathcal{L}(\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}) := E^{\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}}(\mathbf{X},\mathbf{Y}) + \lambda_1||\mathbf{W}_1||^2 + \lambda_2||\mathbf{W}_2||^2 + \lambda_3||\mathbf{b}||^2. \tag{1.3}$$

The above single hidden layer NN with the Euclidean loss is identical to a basis function regression model. Extending this simple NN model to multiple layers results in a more expressive model.

Remark (Model expressiveness)**.** An intuitive definition for model expressiveness might be the complexity of functions a model can capture (defining what complex functions are is not trivial by itself, although when dealing with polynomials one might define polynomials of high degree to be more complex than polynomials of

[2]Note that the output bias was omitted here; this is equivalent to centring the observed outputs.

low degree). In this sense hierarchical basis function models are more expressive than their "flat" counter-parts. It is interesting to note that even though "flat" basis function regression can model any function up to any given precision with a large enough number of basis functions [Cybenko, 1989; Hornik, 1991], with a hierarchy we can use much smaller models. Consider the example of basis function regression with polynomial basis functions $\phi_k \in \{1, x, x^2, ..., x^{K-1}\}$: the set of functions expressible with these K basis functions is {all polynomials up to degree $K-1$}. Composing the basis function regression model L times results in a model that can capture (a subset of) functions from the set {all polynomials up to degree $(K-1)^L$}. A "flat" model capturing polynomials up to degree $(K-1)^L$ would require K^L basis functions, whereas a hierarchical model with similar (but not identical) expressiveness would only require $K \times L$ basis functions. Model expressiveness is further discussed in [Bengio and LeCun, 2007] for example, where binary circuits are used as an illustrative example.

The simple model structure presented above can be extended to specialised models, aimed at treating image inputs or sequence inputs. We will next review these models quickly.

Convolutional neural networks (CNNs). CNNs [LeCun et al., 1989; Rumelhart et al., 1985] are popular deep learning tools for image processing, which can solve tasks that until recently were considered to lie beyond our reach [Krizhevsky et al., 2012; Szegedy et al., 2014]. The model is made of a recursive application of convolution and pooling layers, followed by inner product layers at the end of the network (simple NNs as described above). A convolution layer is a linear transformation that preserves spatial information in the input image (depicted in figure 1.1). Pooling layers simply take the output of a convolution layer and reduce its dimensionality (by taking the maximum of each $(2, 2)$ block of pixels for example). The convolution layer will be explained in more detail in section §3.4.1.

Similarly to CNNs, recurrent neural networks are specialised models designed to handle sequence data.

Recurrent neural networks (RNNs). RNNs [Rumelhart et al., 1985; Werbos, 1988] are sequence-based models of key importance for natural language understanding, language generation, video processing, and many other tasks [Kalchbrenner and Blunsom, 2013; Mikolov et al., 2010; Sundermeyer et al., 2012; Sutskever et al., 2014]. The model's

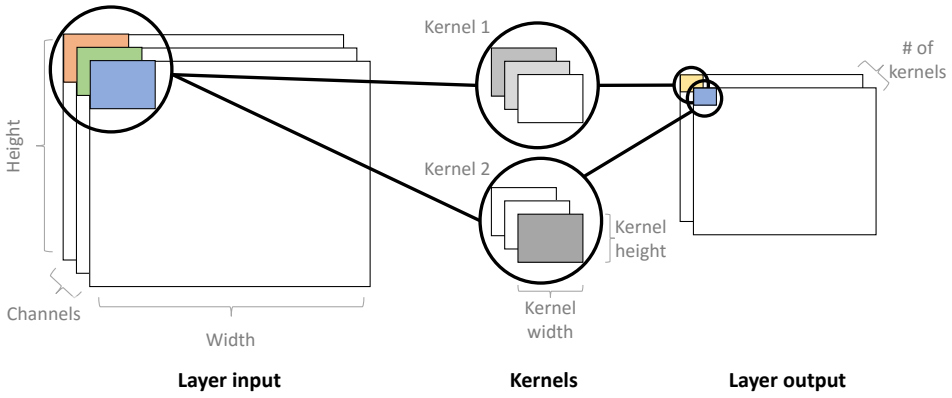


Fig. 1.1 A convolution layer in a CNN. The input image (Layer input) has a given height, width, and channels (RGB for example). Each kernel is convolved with each image patch (a single image patch is depicted under the left-most circle highlight). For example, kernel 2 preserves the blue channel only, resulting in a blue pixel in Layer output. Kernel 1, on the other hand, ignores the blue channel resulting in a yellow pixel in the output layer. This is a simplified view of convolutions: kernels are often not composed of the same value in each spatial location, but rather act as edge detectors or feature detectors.

input is a sequence of symbols, where at each time step a simple neural network (*RNN unit*) is applied to a single symbol, as well as to the network's output from the previous time step. RNNs are powerful models, showing superb performance on many tasks.

We will concentrate on simple RNN models for brevity of notation. Given input sequence $\mathbf{x} = [\mathbf{x}_1, ..., \mathbf{x}_T]$ of length[3] T, a simple RNN is formed by a repeated application of a function $\mathbf{f_h}$. This generates a hidden state $\mathbf{h}_t$ for time step t:

$$\mathbf{h}_t = \mathbf{f_h}(\mathbf{x}_t, \mathbf{h}_{t-1}) = \sigma(\mathbf{x}_t\mathbf{W_h} + \mathbf{h}_{t-1}\mathbf{U_h} + \mathbf{b_h}).$$

for some non-linearity σ. The model output can be defined, for example, as

$$\widehat{\mathbf{y}} = \mathbf{f_y}(\mathbf{h}_T) = \mathbf{h}_T\mathbf{W_y} + \mathbf{b_y}.$$

The definition of LSTM and GRU—more complicated RNN models—is given later in section §3.4.2.

[3]Note the overloading of notation used here: $\mathbf{x}_i$ is a vector in $\mathbb{R}^Q$, and $\mathbf{x}$ is a sequence of vectors of length T.

1.2 Model uncertainty

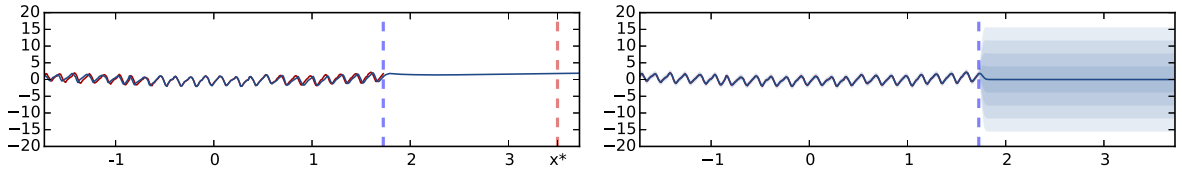
The models above can be used for applications as diverse as skin cancer diagnosis from lesion images, steering in autonomous vehicles, and dog breed classification in a website where users upload pictures of their pets. For example, given several pictures of dog breeds as training data—when a user uploads a photo of his dog—the hypothetical website should return a prediction with rather high confidence. But what should happen if a user uploads a photo of a cat and asks the website to decide on a dog breed?

The above is an example of *out of distribution* test data. The model has been trained on photos of dogs of different breeds, and has (hopefully) learnt to distinguish between them well. But the model has never seen a cat before, and a photo of a cat would lie outside of the data distribution the model was trained on. This illustrative example can be extended to more serious settings, such as MRI scans with structures a diagnostics system has never observed before, or scenes an autonomous car steering system has never been trained on. A possible desired behaviour of a model in such cases would be to return a prediction (attempting to extrapolate far away from our observed data), but return an answer with the added information that the point lies outside of the data distribution (see a simple depiction for the case of regression in figure 1.2). I.e. we want our model to possess some quantity conveying a high level of *uncertainty* with such inputs (alternatively, conveying low *confidence*).

Other situations that can lead to uncertainty include

- noisy data (our observed labels might be noisy, for example as a result of measurement imprecision, leading to *aleatoric uncertainty*),
- *uncertainty in model parameters* that best explain the observed data (a large number of possible models might be able to explain a given dataset, in which case we might be uncertain which model parameters to choose to predict with),
- and *structure uncertainty* (what model structure should we use? how do we specify our model to extrapolate / interpolate well?).

The latter two uncertainties can be grouped under *model uncertainty* (also referred to as *epistemic uncertainty*). Aleatoric uncertainty and epistemic uncertainty can then be used to induce *predictive uncertainty*, the confidence we have in a prediction.



(a) Standard deep learning model (b) Probabilistic model

Fig. 1.2 **Predictive mean and uncertainties on the Mauna Loa CO_2 concentrations dataset, for various models, with out of distribution test point x^*.** In red is the observed function (left of the dashed blue line); in blue is the predictive mean plus/minus two standard deviations. Different shades of blue represent half a standard deviation. Marked with a dashed red line is a point far away from the data: standard deep learning models confidently predict an unreasonable value for the point; the probabilistic model predicts an unreasonable value as well but with the additional information that the model is uncertain about its prediction.

Remark (A note on terminology)**.** The word *epistemic* comes from "episteme", Greek for "knowledge", i.e. epistemic uncertainty is "knowledge uncertainty". *Aleatoric* comes from the Latin "aleator", or "dice player", i.e. aleatoric uncertainty is the "dice player's" uncertainty. Epistemic and aleatoric uncertainties are sometimes referred to as *reducible* and *irreducible* uncertainties respectively, since epistemic uncertainty can be reduced with more data (knowledge), while aleatoric uncertainty cannot (the stochasticity of a dice roll cannot be reduced by observing more rolls). We will avoid this terminology though, since aleatoric uncertainty can also be seen as "reducible" through an increase in measurement precision, i.e. by changing the underlying system with which we perform the experiment.

Uncertainty information is often used in the life sciences, as discussed in the Nature papers by Herzog and Ostwald [2013]; Krzywinski and Altman [2013]; Nuzzo [2014], as well as the entertaining case in [Trafimow and Marks, 2015]. In such fields it is quite important to quantify our confidence about the models' predictions. Uncertainty information is also important for the practitioner. Understanding if a model is underconfident or falsely over-confident (i.e. its uncertainty estimates are too small) can help get better performance out of it. Recognising that test data is far from the training data we could augment the training data for example.

But perhaps much more important, model uncertainty information can be used in systems that make decisions that affect human life—either directly or indirectly—as discussed next.

1.3 Model uncertainty and AI safety

With recent engineering advances in the field of machine learning, systems that until recently were only applied to toy data are now being deployed in real-life settings. Among these settings are scenarios in which control is handed-over to automated systems in situations which have the possibility to become life-threatening to humans. These include automated decision making or recommendation systems in the medical domain, autonomous control of drones and self driving cars, the ability to affect our economy on a global scale through high frequency trading, as well as control of critical systems. These can all be considered under the umbrella field of *AI safety*.

This interpretation of AI safety is rather different to other interpretations given in the field, which mostly concentrate on *reinforcement learning* (RL) settings. For example, some look at the design of environments for self-learning agents that do not allow the agent to exploit deficiencies in the learning process itself (such as "reward hacking"; see for example [Amodei et al., 2016]). In contrast, I will discuss scenarios in which certain decisions made by a machine learning model trained in a *supervised* setting might endanger human life (i.e. settings in which a model incorrectly mapping inputs to outputs can lead to undesirable consequences). In some of these scenarios, relying on model uncertainty to adapt the decision making process might be key to preventing unintended behaviour.

1.3.1 Physician diagnosing a patient

When a physician advises a patient to use certain drugs based on a medical record analysis, the physician would often rely on the confidence of the expert analysing the medical record. The introduction of systems such as automated cancer detection based on MRI scans though could make this process much more complicated. Even at the hands of an expert, such systems could introduce biases affecting the judgement of the expert. A system encountering test examples which lie outside of its data distribution could easily make unreasonable suggestions, and as a result unjustifiably bias the expert. However, given model confidence an expert could be informed at times when the system is essentially guessing at random.

1.3.2 Autonomous vehicles

Autonomous systems can range from a simple robotic vacuum scuttering around the floor, to self-driving cars transporting people and goods from one place to another. These

systems can largely be divided into two groups: those relying on rule-based systems to control their behaviour, and those which can learn to adapt their behaviour to their environment. Both can make use of machine learning tools. The former group through low-level use of machine learning algorithms for feature extraction, and the latter one through reinforcement learning.

With self-driving cars, low level feature extraction such as image segmentation and image localisation are used to process raw sensory input [Bojarski et al., 2016]. The output of such models is then fed into higher-level decision making procedures. The higher-level decision making can be done through expert systems for example relying on a fixed set of rules ("if there is a cyclist to your left, do not steer left"). However, mistakes done by lower-level machine learning components can propagate up the decision making process and lead to devastating results. One concrete example demonstrating the risk of such approaches, in an assisted driving system, is the failure of a low-level component to distinguish the white side of a turning trailer from a bright sky, which led to the first fatality of an assisted driving system [NHTSA, 2017]. In such modular systems, one could use the model's confidence in low-level components and make high-level decisions given this uncertainty information. For example, a segmentation system which can identify its uncertainty in distinguishing between the sky and another vehicle could alert the user to take control over the steering.

1.3.3 Critical systems and high frequency trading

As a final example, it is interesting to notice that control over critical systems is slowly being handed over to machine learning systems. This can happen in a post office, sorting letters according to their zip code [LeCun and Cortes, 1998; LeCun et al., 1998], or in a nuclear power plant with a system responsible for critical infrastructure [Linda et al., 2009]. Even in high frequency trading—where computers are given control over systems that could potentially destabilise entire economic markets—issuing an unusual trading command could cause a calamity. A possible solution, when using rule-based decision making, is to rely on formal program verification systems. Such systems allow the developer to verify the program is working as intended before deployment. But machine learning-based decision making systems do not allow this. What should a system do in the case of outputs with high uncertainty?

With model confidence at hand one possible solution would be to treat uncertain outputs as special cases explicitly. In the case of a critical system we might decide to pass the input to a human to make a decision. Alternatively, one could use a simple and

fast model to perform predictions, and use a more elaborate but slower model only for inputs on which the weak model is uncertain.

In this work we will develop the tools required to reason about model confidence in deep learning, which could be applied in the scenarios discussed above. This will be by developing a general framework that can be applied to existing tools. This in turn allows the continued use of existing systems which have proven themselves useful, and for which large amounts of research have already been dedicated. Concrete examples of the use of these developments in the settings mentioned above will be given in section §5.1.

1.4 Applications of model uncertainty

Beside AI safety, there exist many applications which rely on model uncertainty. These applications include choosing what data to learn from, or exploring an agent's environment efficiently. Common to both these tasks is the use of model uncertainty to learn from *small amounts of data*. This is often a necessity in settings in which data collection is expensive (such as the annotation of individual examples by an expert), or time consuming (such as the repetition of an experiment multiple times).

1.4.1 Active learning

How could we use machine learning to aid an expert working in a laborious field? One approach is to automate small parts of the expert's work, such as mundane cell counting, or cancer diagnosis based on MRI scans. This can be a difficult problem in machine learning though. Many machine learning algorithms, including deep learning, often require large amounts of labelled data to generalise well. The amount of labelled data required increases with the complexity of the problem or the complexity of the input data: image inputs for example often require large models to be processed, and in turn these require large amounts of data (Krizhevsky et al. [2012] for example use hundreds of gigabytes of labelled images). To automate MRI scan analysis for example, this would require an expert to annotate a large number of MRI scans, labelling them to indicate a patient having cancer or not. But expert time is expensive, and often obtaining the amount of required labelled data is not feasible. How can we learn in settings where labelled data is scarce and expert knowledge is expensive?

One possible approach to this task could rely on active learning [Settles, 2010]. In this learning framework the model itself would choose what unlabelled data would be most

informative for it, and ask an external "oracle" (for example a human annotator) for a label only for these new data points. The choice of data points to be labelled is done through an acquisition function, which ranks points based on their potential informativeness. Different acquisition functions exist, and many make use of model uncertainty about the unlabelled data points in order to decide on their potential informativeness [Houlsby et al., 2011]. Following this learning framework we can decrease the amount of required data by orders of magnitude, while still maintaining good model performance (as we will see below in §5.2).

Returning to the example above of cancer diagnosis from MRI scans, we would seek a model that can produce *good uncertainty estimates* for image data, and rely on these to design an appropriate acquisition function. Deep learning provides superb tools for image processing that generalise well, but these rely on huge amounts of labelled data and do not provide model uncertainty. In this work we will develop extensions of such tools that can be deployed in small data regimes, and provide good model confidence. With these tools we will demonstrate the feasibility of the ideas above in active learning (section §5.2, joint work with Riashat Islam as part of his Master's project).

1.4.2 Efficient exploration in deep reinforcement learning

Reinforcement learning (RL) algorithms learn control tasks via trial and error, much like a child learning to ride a bicycle [Sutton and Barto, 1998]. But trials of real world control tasks often involve time and resources we wish not to waste. Alternatively, the number of trials might be limited due to wear and tear of the system, making data-efficiency critical.

As a simple introduction to reinforcement learning, consider an agent (a Roomba vacuum for example) that needs to learn about its environment (a living room) based on its actions (rolling around in different directions). It can decide to go forward and might bump into a wall. Encouraging the Roomba not to crash into walls with positive rewards, over time it will learn to avoid them in order to maximise its rewards. The Roomba has to explore its environment looking for these rewards, and trade-off between this exploration, and exploitation of what it already knows.

Recent advances in deep learning approaches to RL (referred to as *deep RL*) have demonstrated impressive results in game playing [Mnih et al., 2013]. Such approaches make use of NNs for Q-value function approximation. These are functions that estimate the quality of different actions an agent can take. Epsilon greedy search is often used where the agent selects its best action with some probability and explores otherwise. But with uncertainty information an agent can decide when to exploit and when to explore

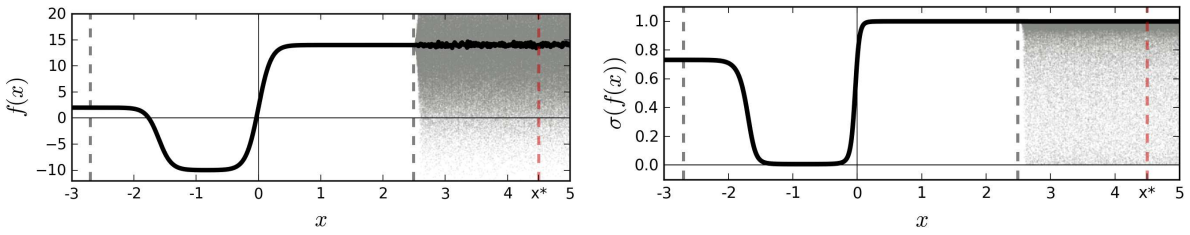
its environment. And with uncertainty estimates over the agent's Q-value function, techniques such as Thompson sampling [Thompson, 1933] can be used to learn much faster. This will be demonstrated below (section §5.3).

Even though exploration techniques such as Thompson sampling can help learn better policies faster, a much more drastic improvement in data efficiency can be achieved by modelling the system dynamics [Atkeson and Santamaria, 1997]. A dynamics model allows the agent to generalise its knowledge about the system dynamics to other, unobserved, states. *Probabilistic* dynamics models allow an agent to consider transition uncertainty throughout planning and prediction, improving data efficiency even further. PILCO [Deisenroth and Rasmussen, 2011], for example, is a data-efficient probabilistic model-based policy search algorithm. PILCO analytically propagates uncertain state distributions through a Gaussian process dynamics model. This is done by recursively feeding the output state distribution (*output uncertainty*) of one time step as the input state distribution (*input uncertainty*) of the next time step, until a fixed time horizon T. This allows the agent to consider the long-term consequences (expected cumulative cost) of a particular controller parametrisation w.r.t. all plausible dynamics models. PILCO relies on Gaussian processes (GPs), which work extremely well with small amounts of low dimensional data, but scale cubically with the number of trials. Further, PILCO's distribution propagation adds a squared term in the observation space dimensionality, making it hard to scale the framework to high dimensional observation spaces. This makes it difficult to use PILCO with tasks that require a larger number of trials. Even more so, PILCO does not consider temporal correlation in model uncertainty between successive state transitions. This means that PILCO underestimates state uncertainty at future time steps [Deisenroth et al., 2015], which can lead to diminished performance.

In section §5.4 we attempt to answer these shortcomings by replacing PILCO's Gaussian process with a Bayesian deep dynamics model, while maintaining the framework's probabilistic nature and its data-efficiency benefits (joint work with Rowan McAllister and Carl Rasmussen [Gal et al., 2016]).

1.5 Model uncertainty in deep learning

Having established that model confidence is a good quantity to possess, it is important to note that most deep learning models do not offer such information. Regression models output a single vector regressing to the mean of the data (as can be seen in figure 1.2). In classification models, the probability vector obtained at the end of the pipeline (the softmax output) is often erroneously interpreted as model confidence. A model can be



(a) Arbitrary function $f(\mathbf{x})$ as a function of data $\mathbf{x}$ (softmax *input*)

(b) $\sigma(f(\mathbf{x}))$ as a function of data $\mathbf{x}$ (softmax *output*)

Fig. 1.3 **A sketch of softmax input and output for an idealised binary classification problem.** Training data is given between the dashed grey lines. Function point estimate is shown with a solid line. Function uncertainty is shown with a shaded area. Marked with a dashed red line is a point x^* far from the training data. Ignoring function uncertainty, point x^* is classified as class 1 with probability 1.

uncertain in its predictions even with a high softmax output (figure 1.3). Passing a point estimate of a function (solid line 1.3a) through a softmax (solid line 1.3b) results in extrapolations with unjustified high confidence for points far from the training data. x^* for example would be classified as class 1 with probability 1. However, passing the distribution (shaded area 1.3a) through a softmax (shaded area 1.3b) better reflects classification uncertainty far from the training data.

Even though modern deep learning models used in practice do not capture model confidence, they are closely related to a family of probabilistic models which induce probability distributions over functions: the Gaussian process. Given a neural network, by placing a probability distribution over each weight (a standard normal distribution for example), a Gaussian process can be recovered in the limit of infinitely many weights (see Neal [1995] or Williams [1997]). For a finite number of weights, model uncertainty can still be obtained by placing distributions over the weights—these models are called *Bayesian neural networks*. These were extensively studied by [MacKay, 1992b], work which was further extended by [Neal, 1995]. More recently these ideas have been resurrected under different names with variational techniques by [Blundell et al., 2015; Graves, 2011; Kingma and Welling, 2013] (although such techniques used with Bayesian neural networks can be traced back as far as Hinton and Van Camp [1993] and Barber and Bishop [1998]). But some of these models are quite difficult to work with—often requiring many more parameters to be optimised—and haven't really caught-on within the deep learning community, perhaps because of their limited practicality.

What would make a tool for obtaining model uncertainty *practical* then? One requirement of such a tool would be to scale well to large data, and scale well to complex models (such as CNNs and RNNs). Much more important perhaps, it would be

impractical to change existing model architectures that have been well studied, and it is often impractical to work with complex and cumbersome techniques which are difficult to explain to non-experts. Existing approaches to obtain model confidence often do not scale to complex models or large amounts of data, and require us to develop new models for existing tasks for which we already have well performing tools.

We will thus concentrate on the development of practical techniques to obtain model confidence in deep learning, techniques which are also well rooted within the theoretical foundations of probability theory and Bayesian modelling. Specifically, we will make use of stochastic regularisation techniques (SRTs). SRTs are recently developed techniques for model regularisation that have been tremendously successful within deep learning, and are used in almost *all* modern deep learning models. These techniques adapt the model output stochastically as a way of model regularisation (hence the name *stochastic regularisation*). This results in the loss becoming a random quantity, which is optimised using tools from the stochastic non-convex optimisation literature. Popular SRTs include dropout [Hinton et al., 2012], multiplicative Gaussian noise [Srivastava et al., 2014], dropConnect [Wan et al., 2013], and countless other recent techniques[4,5].

As we will see below, we can take almost any network trained with an SRT, and given some input $\mathbf{x}^*$ obtain a predictive mean $\mathbb{E}[\mathbf{y}^*]$ (the expected model output given our input), and predictive variance $\mathrm{Var}[\mathbf{y}^*]$ (how much the model is confident in its prediction). To obtain these, we simulate a network output with input $\mathbf{x}^*$, treating the SRT as if we were using the model during training (i.e. obtain a random output through a *stochastic forward pass*). We repeat this process several times (for T repetitions), sampling i.i.d. outputs $\{\widehat{\mathbf{y}}_1^*(\mathbf{x}^*), ..., \widehat{\mathbf{y}}_T^*(\mathbf{x}^*)\}$. As will be explained below, these are empirical samples from an *approximate predictive distribution*. We can get an empirical estimator for the predictive mean of our approximate predictive distribution as well as the predictive variance (our uncertainty) from these samples:

$$\begin{aligned}\mathbb{E}[\mathbf{y}^*] &\approx \frac{1}{T}\sum_{t=1}^{T}\widehat{\mathbf{y}}_t^*(\mathbf{x}^*)\\ \mathrm{Var}[\mathbf{y}^*] &\approx \tau^{-1}\mathbf{I}_D + \frac{1}{T}\sum_{t=1}^{T}\widehat{\mathbf{y}}_t^*(\mathbf{x}^*)^T\widehat{\mathbf{y}}_t^*(\mathbf{x}^*) - \mathbb{E}[\mathbf{y}^*]^T\mathbb{E}[\mathbf{y}^*].\end{aligned} \tag{1.4}$$

Theoretical justification for these two simple equations will be given in chapter 3.

[4] The idea of adding noise to a model to avoid overfitting is quite old, and for input noise has been studied for example in [Bishop, 1995].

[5] In the dropout case, initial research studying the technique from a Bayesian perspective includes [Ba and Frey, 2013; Maeda, 2014; Wang and Manning, 2013].

Equation (1.4) results in uncertainty estimates which are practical with large models and big data, and that can be applied in image based models, sequence based models, and many different settings such as reinforcement learning and active learning. Further, the combination of these techniques allows us to perform tasks that were not possible until recently. For example, we demonstrate below how to perform active learning with *image data*, a task which is extremely challenging due to the lack of tools offering good uncertainty estimates from image data.

1.6 Thesis structure

The first part of this thesis (chapters 3–5) will be concerned with providing tools to obtain practical uncertainty estimates, and demonstrating how these tools could be used in many example applications. This part should be easily accessible for experts as well as non-experts in the field. The second part of this thesis (chapter 6) goes in depth into the theoretical implications of the work above.

Some of the work in this thesis was previously presented in [Gal, 2015; Gal and Ghahramani, 2015a,b,c,d, 2016a,b,c; Gal et al., 2016], but this thesis contains many new pieces of work as well. The most notable of these are a theoretical analysis of Monte Carlo estimator variance used in variational inference (§3.1.1–§3.1.2), a survey of measures of uncertainty in classification tasks (§3.3.1), an empirical analysis of different Bayesian neural network priors (§4.1) and posteriors with various approximating distributions (§4.2), new quantitative results comparing dropout to existing techniques (§4.3), tools for heteroscedastic predictive uncertainty in Bayesian neural networks (§4.6), applications in active learning (§5.2), a discussion of what determines what our model uncertainty looks like (§6.1–§6.2), an analytical analysis of the dropout approximating distribution in Bayesian linear regression (§6.3), an analysis of ELBO-test log likelihood correlation (§6.4), discrete prior models (§6.5), an interpretation of dropout as a proxy posterior in spike and slab prior models (§6.6), as well as a procedure to optimise the dropout probabilities based on the variational interpretation to separate the different sources of uncertainty (§6.7).

The code for the experiments presented in this work is available at `https://github.com/yaringal`.