Benchmarks for Bayesian Deep Learning: Image Segmentation

Jishnu Mukhoti
St Hugh’s College
University of Oxford

A thesis submitted for the degree of
Master of Science
Trinity 2018
This thesis is dedicated to my parents.
Acknowledgements

Firstly, I would like to express my immense gratitude to Dr Yarin Gal for his enthusiastic, excellent supervision and his constant support throughout the project. His insightful comments during our one-on-one meetings helped me a lot in finishing my project and also in writing up the thesis. Secondly, I am very grateful to my parents for the encouragement that they gave me whenever I was in need of some. Finally, a big thank you to St. Hugh’s college for the excellent facilities which really helped in providing the ideal work environment for the project.

Thank you very much
Abstract

Bayesian Deep Learning (BDL) is a field of Machine Learning involving models which, when trained, can not only produce predictions but can also generate values which express the model confidence on the predictions. In the recent past, BDL techniques have been extensively applied to several problems in computer vision including object detection [1] and semantic segmentation [2]. Due to the rising popularity of BDL techniques, there exists a need to develop tools which can be used to evaluate the performance of such models and also help in comparing different model architectures or inference techniques. In this work, we study and create such BDL evaluation tools for the problem of semantic segmentation. Furthermore, we also reproduce three well-known inference techniques in BDL (Monte Carlo dropout [3], concrete dropout [4] and deep ensembles [5]) and evaluate them using the tools and metrics which we develop.

The main contributions of the work include: i) new scale-based and probabilistic variants of the state-of-the-art DeepLab-v3 [6] architecture which was designed for the task of semantic image segmentation, ii) study of sanity check tools which can be used to ensure that the behaviour of a trained BDL model is as expected (for instance, checking the inverse relationship between model accuracy and model uncertainty) and iii) development of novel metrics which can be used to evaluate the performance of BDL inference techniques and model architectures and can also be used to compare one model against another.
# Contents

1 Introduction: Evaluating a Bayesian Deep Learning Model ......................................... 1  
  1.1 Background ............................................................................................................ 1  
  1.2 Problem definition ................................................................................................. 3  
  1.3 Approach ................................................................................................................ 4  
  1.4 Thesis Structure .................................................................................................... 5  

2 Preliminaries: Background on BDL and Semantic Segmentation ................................ 6  
  2.1 Bayesian Deep Learning (BDL) ............................................................................ 6  
     2.1.1 Bayesian Modelling ......................................................................................... 6  
     2.1.2 Dropout for approximate inferencing .............................................................. 8  
  2.2 Semantic Segmentation ......................................................................................... 9  
     2.2.1 Fully Convolutional Networks (FCN) ............................................................ 10  
     2.2.2 Encoder-Decoder architecture ........................................................................ 11  
     2.2.3 Atrous Convolutions ...................................................................................... 12  
  2.3 Bayesian SegNet .................................................................................................... 13  

3 Bayesian DeepLab ....................................................................................................... 17  
  3.1 DeepLab v3 ............................................................................................................ 17  
     3.1.1 Xception Module ............................................................................................ 18  
     3.1.2 Multi-scale Image Processing ......................................................................... 20  
     3.1.3 Decoder Module .............................................................................................. 22  
  3.2 MC Dropout in DeepLab: Bayesian DeepLab ....................................................... 23  
     3.2.1 Model Variants ............................................................................................... 24  
     3.2.2 Uncertainty Metrics ....................................................................................... 28  

4 Evaluation Metrics and Alternative Inference Techniques ........................................ 31  
  4.1 Metrics ..................................................................................................................... 31  
     4.1.1 Semantic Segmentation Metrics ..................................................................... 32  
     4.1.2 BDL Metrics .................................................................................................. 33
4.1.2.1 Sanity check metrics ............................................. 34
4.1.2.2 Performance Evaluation Metrics ................................. 35
4.2 Alternative Inference Techniques .................................... 39
  4.2.1 Concrete Dropout .................................................. 39
  4.2.2 Deep Ensembles .................................................. 42

5 Experiments and Results ........................................... 45
  5.1 Cityscapes .......................................................... 45
  5.2 Experimental parameters and a brief analysis of the output .... 47
  5.3 Experiment 1: Sanity Checks ........................................ 49
  5.4 Experiment 2: Performance Evaluation ............................ 49

6 Conclusions .......................................................... 57
  6.1 Summary and Contributions ......................................... 57
  6.2 Future Research ................................................... 58

A Using the BDL Benchmark Tool: Simple API Tutorial ........... 60
  A.1 BDL Benchmark Tool .............................................. 61
     A.1.1 Iterator APIs ................................................ 62
     A.1.2 Metrics APIs ............................................... 62
     A.1.3 The \texttt{generate\_report()} API ............................ 63

B Medium-scale Bayesian DeepLab variants ........................... 67

C Results of the Bayesian DeepLab variants using MC dropout inference .................................................. 69

Bibliography .............................................................. 73
List of Tables

3.1 Xception Module structures of the full-scaled probabilistic model variants of Bayesian DeepLab ........................... 28
3.2 Xception Module structures of the toy model variants of Bayesian DeepLab ......................................................... 28

5.1 Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with multiple inference techniques using conventional semantic segmentation metrics: pixel accuracy, mean accuracy and mean I/U (expressed as percentages). .......................... 52
5.2 Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with multiple inference techniques using BDL performance evaluation metrics: \( P(\text{accurate} | \text{certain}) \), \( P(\text{uncertain} | \text{inaccurate}) \) and PAVPU Accuracy. .................................................. 56

B.1 Xception Module structures of the medium-scaled probabilistic model variants of Bayesian DeepLab ................................................................. 68

C.1 Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with MC dropout inferencing using conventional semantic segmentation metrics: pixel accuracy, mean accuracy and mean I/U (expressed as percentages). .......................... 72
C.2 Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with MC dropout inferencing using BDL performance evaluation metrics: \( P(\text{accurate} | \text{certain}) \), \( P(\text{uncertain} | \text{inaccurate}) \) and PAVPU Accuracy. .......................... 72
List of Figures

2.1 The DAG FCN proposed in [7] ........................................ 11
2.2 UNet architecture [8] ................................................. 12
2.3 1-D atrous convolution with rate = 2 .......................... 14

3.1 Depthwise-separable convolution ................................... 19
3.2 Xception Modules ...................................................... 20
3.3 ASPP module with atrous rates 6, 12 and 18 ...................... 22
3.4 Decoder module structure used to upsample network outputs 23
3.5 Architecture of the Bayesian DeepLab full-scale Dropout-Entry-Flow variant ........................................... 27

4.1 Worked out example of evaluation metrics for semantic segmentation: pixel accuracy, mean accuracy, mean I/U .................... 33
4.2 Worked out example of computing the class-wise accuracy/frequency vs mean uncertainty and the accuracy for thresholded uncertainty metrics. 35
4.3 Worked out example of computing the patch based evaluation metrics 39
4.4 Typical module of a Bayesian DeepLab variant with concrete dropout layers. .................................................. 42
4.5 Illustration of how a deep ensemble produces predictions and uncertainty estimates ............................................. 44

5.1 Sample images and corresponding labels from the Cityscapes dataset 46
5.2 Images showing the ground-truth label, pixel-wise predictions and pixel-wise uncertainty maps for both predictive entropy and mutual information from a trained Bayesian DeepLab medium-scale model with MC dropout inference. 48
5.3 Plots of classwise accuracy vs uncertainty and classwise frequency vs uncertainty for full scale-models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques. 53
5.4 Plots of accuracy for the top x% most confident pixels (x varying from 5 to 100 in steps of 5) for the toy, medium-scale and full-scale models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques. There are two plots for each scale, one corresponding to predictive entropy and the other for mutual information. 54

5.5 Plots of the PAVPU accuracy for varying thresholds of predictive entropy. The figure includes plots for the toy, medium-scale and full-scale models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques. 55

A.1 High level design of the BDL Benchmark tool for semantic segmentation 61

C.1 Plots of accuracy for the top x% most confident pixels (x varying from 5 to 100 in steps of 5) for the toy, medium-scale and full-scale models using the various Bayesian DeepLab MC dropout variants. There are two plots for each scale, one corresponding to predictive entropy and the other for mutual information. 70

C.2 Plots of the PAVPU accuracy for varying thresholds of predictive entropy. The figure includes plots for the toy, medium-scale and full-scale models for different Bayesian DeepLab MC dropout variants. 71
Chapter 1

Introduction: Evaluating a Bayesian Deep Learning Model

1.1 Background

Over the past couple of years, deep learning techniques have had tremendous success in quite a few fields of machine learning including computer vision [9], natural language processing [10], speech recognition [11], bioinformatics [12] and others. These techniques have not only outperformed state-of-the-art machine learning methods but have also shown performance comparable to human experts. However, most deep learning models produce point-estimates as outputs and hence, we only obtain the value of the prediction made by the model without having any knowledge about how confident the model is on its prediction. There are many scenarios in which some additional knowledge about the model’s confidence is not only useful but quite essential.

AI systems are now being given control in scenarios which involve real-life problems and can affect and even harm human life if not properly controlled. For instance, let us consider the case of a physician diagnosing a patient. The physician can choose to prescribe drugs based on the medical record analysis of the patient. If this analysis is automated, the physician essentially trusts the AI expert and if the judgement of the expert is biased or wrong, it can lead to prescription of unwanted or even harmful drugs. Even more dangerous is the scenario of self driven cars. Such cars depend on low level feature extraction like image segmentation to process their raw inputs. Based on the outputs of such models, higher level decisions are carried out in the car. If the model responsible for the low level tasks like segmentation makes mistakes, this can propagate into high level decision making and can lead to catastrophic events. For instance, in May 2016, there was a fatal incident where an assisted driving sys-
tem confused the white side of a trailer with the bright sky [13]. If the system could have incorporated a high level of uncertainty (or low confidence) in its prediction, the disaster might have been avoided. There are several other real-world applications like high frequency trading, autonomous drones etc. where mistakes made by an AI can have dire consequences. A list of some of the problems in AI safety can be found in the paper [14].

The above problems indicate the need for our networks to not make overconfident predictions, i.e., to incorporate the model confidence into the prediction as well. This brings us to the topic of Bayesian modelling in neural networks. The main idea is to place a prior distribution over the network weights and then update the prior distribution given the training data using Bayesian methods [15]. The updated distribution on the network weights is known as the posterior. The posterior can in turn be used to compute the output or the predictive distribution which is the distribution over possible output values, thereby providing the predictive uncertainty of the network as well. With large networks however, classical Bayesian inference is a computationally intractable task. Hence, there have been several methods to generate approximated predictive distributions. Some of them include Laplace approximation [16], Markov chain Monte Carlo methods [17], assumed density filtering [18], expectation propagation [19], Langevin diffusion methods [20] and Hamiltonian methods [21]. In general however, most of these methods are much harder to implement and train compared to normal deep networks and hence haven’t gained a lot of popularity in the deep learning community.

In the paper [3], the authors suggested a relatively simple method to obtain such uncertainty estimates from any deep learning model. The method uses a Stochastic Regularisation Technique (SRT) like dropout or multiplicative Gaussian noise during the training phase. During the test phase, the same SRT is applied in multiple Monte Carlo forward passes for a given input and the mean and variance of the outputs generated from these multiple passes are used as model prediction and uncertainty estimates respectively. In their paper [3], the authors have provided a theoretical justification as to why the use of such SRTs can be seen as an approximate inference technique in deep learning models. The SRTs are simple to implement, do not require any additional parameters to produce uncertainty estimates and are quite well-known and frequently used in the deep learning community.
Following the paper on MC-dropout, there have been multiple developments on the topic of approximate inference techniques in deep networks. Some of them include dropout layer variants which can optimise the dropout probability as part of the gradient based optimisation process [4], deep ensemble networks as a means to generate uncertainty estimates [5], modified optimisation algorithms like Bayes by Backprop [22], deep kernel learning techniques [23] etc. Due to the rising popularity of these new scalable methods to generate uncertainty estimates from deep networks, we are faced with the question of how to evaluate these models and compare them against one another. Furthermore, such comparison tools and metrics may also depend on the machine learning task to which the networks are applied. This thesis is an attempt to study and create such tools and metrics in order to enable us to perform comparisons between Bayesian Deep Learning models designed for the task of semantic image segmentation.

1.2 Problem definition

Semantic segmentation is a computer vision problem where, given an image, the task is to classify each pixel of the image to the object which the pixel represents. Such predictions are also known as dense predictions in the literature. The pixel-wise classification automatically delineates the boundaries of different objects in the image, hence the name “segmentation”. A Bayesian Deep Learning model which has been designed for the task of semantic segmentation will not only produce predictions for each pixel but also generate pixel-wise uncertainty estimates.

Evaluating such a BDL model is a particularly challenging task because unlike predictions, we don’t have a strong definition of what a good uncertainty estimate is. There are no ground-truth uncertainty estimates. Hence, we have to judge the quality of model uncertainty based on how accurate the model is for the same input. In particular, we use the following two assumptions when judging the quality of an uncertainty estimate produced by a model:

1. **Assumption 1.** The model should know what it claims to know.

2. **Assumption 2.** The model may or may not know what it claims not to know.

The first assumption indicates that if a model is certain about its prediction, it should ideally be an accurate prediction. The second assumption suggests that if a model is not confident about its prediction, it may or may not be accurate. This hints at an
inverse relation between model accuracy and uncertainty, a property which we exploit while designing evaluation metrics for such models. Given this background we can finally define the problem which we tackle in this thesis. The problem is essentially two-fold:

1. First, we want to develop tools and metrics to measure the performance of BDL models which have been designed for the task of semantic segmentation.

2. Second, we want to reproduce some well-known inference techniques on state-of-the-art architectures and evaluate the resulting BDL networks using the tools we develop. This will help in building benchmarks which researchers can use as baselines to compare new models with.

### 1.3 Approach

In order to evaluate a Bayesian Deep Learning model, first, we need to design one specifically for the task of semantic segmentation. There has been a lot of research over the past couple of years on deep networks for semantic segmentation. Some of the most notable works include Fully Convolutional Networks (FCNs) [7], encoder-decoder architectures like UNet [8] and SegNet [24] and networks which use atrous convolution layers like DeepLab [25]. In this work, we modify the architecture of the DeepLab-v3 [26] to produce a probabilistic version of DeepLab which we call the Bayesian DeepLab. We choose DeepLab-v3 as it is one of the state-of-the-art and best performing networks in current literature.

The primary technique of approximate inference which we use is MC-dropout [3] which involves placing dropout layers in different regions of the network to produce Monte-Carlo samples during test time as points drawn from the posterior distribution. This leads to multiple probabilistic variants of Bayesian DeepLab depending on where the dropout layers are placed in the network. Furthermore, we also modify the DeepLab-v3 architecture using other inference techniques like inserting concrete dropout [4] layers instead of vanilla dropout and creating ensembles [5] of multiple deterministic and probabilistic DeepLab networks.

With the different versions of Bayesian DeepLab, we develop tools which can act as sanity checks for the model behaviour as well as metrics to evaluate the performance of the network. As mentioned above, we exploit the inverse relation between
model accuracy and model uncertainty to achieve this. These metrics are presented in detail in Chapter 4 of the thesis. We perform all our experiments on the well-known Cityscapes [27] dataset which contains several pictures of urban street scenes. In the experiments, we evaluate each of the Bayesian DeepLab variants and report the results on the inference techniques which perform the best, thereby laying down benchmarks against which other models can be compared.

1.4 Thesis Structure

The thesis has been organised as follows. Chapter 2 provides an explanation of the preliminary concepts on Bayesian Deep Learning as well as the state-of-the-art networks for semantic segmentation. The architecture of the DeepLab-v3 network along with its primary modification using dropout layers (Bayesian DeepLab) have been discussed in details in Chapter 3. In Chapter 4, we describe the tools and metrics which can be used to ensure expected model behaviour as well as evaluate model performance. Furthermore, we also discuss alternative inference techniques which have been applied on the DeepLab-v3 network to generate baselines for comparison with the normal Bayesian DeepLab having MC-dropout. Chapter 5 presents an account of the experiments which we undertake to compare the different BDL models and inference techniques. Finally, we present our conclusions about the work in Chapter 6.
Chapter 2

Preliminaries: Background on BDL and Semantic Segmentation

In order to explore the application of Bayesian Deep Learning (BDL) to the problem of semantic segmentation, an in-depth discussion of both the theoretical aspects of BDL and the current research on semantic segmentation is necessary. This chapter aims to provide the readers an insight into both of the above as well as discuss a work which has attempted to apply BDL techniques to a network designed for semantic segmentation.

2.1 Bayesian Deep Learning (BDL)

2.1.1 Bayesian Modelling

The training data for a supervised machine learning system includes a set of data points \( X = \{ x_1, x_2, ..., x_N \} \) and the corresponding outputs \( Y = \{ y_1, y_2, ..., y_N \} \). The outputs can be real-valued in case of regression problems or categorical values for classification problems. The aim of the system is to generate a function \( f^\theta \) based on parameters \( \theta \) such that ideally, \( y = f^\theta(x) \) for all points \( x \) in the input space. The idea behind Bayesian machine learning is to find the set of parameters \( \theta \) which is most likely to have generated the \( (x_i, y_i) \) pairs in the data set. In order to do so, an initial probability distribution over the parameter space is assumed. General candidates for this prior distribution \( p(\theta) \) are uniform or Gaussian distributions. Next, with the input of data, this distribution is updated to capture parameters which are more likely to have generated the dataset. This is done through the application of Bayes’s theorem:

\[
p(\theta|X, Y) = \frac{p(Y|X, \theta)p(\theta)}{p(Y|X)}. \quad (2.1)
\]
In the above equation, \( p(Y|X, \theta) \) is called the \textit{likelihood}, i.e., the conditional probability of observing the outputs given the inputs and the parameters. Finally, we get the \textit{posterior} distribution \( p(\theta|X, Y) \) over the parameter space which indicates the updated belief on which parameters are most likely to have generated the data.

Once the posterior is available, given a new data point \( x^* \), a distribution over possible output values can be computed as follows:

\[
p(y^*|x^*, X, Y) = \int p(y^*|x^*, \theta)p(\theta|X, Y)d\theta. \tag{2.2}
\]

However, in most sophisticated Bayesian models, computing the normaliser \( p(Y|X) \) as:

\[
p(Y|X) = \int p(Y|X, \theta)p(\theta)d\theta \tag{2.3}
\]

is not analytically possible. Hence, instead of computing the true posterior, an approximating \textit{variational} distribution \( q_\phi(\theta) \), parameterised by \( \phi \) is evaluated in its place. The goal is to get the approximating distribution as close to the posterior as possible. This is done by minimising the Kullback-Leibler (KL) divergence [28] between the posterior and the variational distributions. The KL divergence is given as follows:

\[
KL(q_\phi(\theta)||p(\theta|X, Y)) = \int q_\phi(\theta) \log \frac{q_\phi(\theta)}{p(\theta|X, Y)} d\theta. \tag{2.4}
\]

Minimising the above function is equivalent to maximising the \textit{evidence lower bound} (ELBO) with respect to the parameters defining the variational distribution \( q_\phi(\theta) \), i.e., \( \phi \). The ELBO objective function is given below:

\[
\mathcal{L}_{VI}(\phi) = \int q_\phi(\theta) \log p(Y|X, \theta)d\theta - KL(q_\phi(\theta)||p(\theta)). \tag{2.5}
\]

This results in an optimisation problem instead of the Bayesian modelling marginalisation problem where the normaliser given in equation (2.3) needs to be computed. This process is known as \textit{variational inference}. The minimum of the objective function provides a variational distribution \( q^*_\phi(\theta) \) which can be used in the place of the posterior in equation (2.2) to compute the output for a given new point.

Although, the variational inference approach replaces the calculation of integrals with derivatives which is much easier, the method does not scale well to large datasets and to complex models [29]. In fact, such Bayesian modelling approaches have been applied to neural networks where a distribution is placed over the network weights.
However, performing inference in Bayesian neural nets is difficult and computationally intensive. Hence these networks haven’t really caught on in the deep learning community where the networks primarily deal with point estimates.

2.1.2 Dropout for approximate inferencing

In the works [3], [29], the authors have described practical methods to obtain uncertainty estimates from deep convolutional neural networks. Their approach involves using Stochastic Regularisation Techniques (SRTs) [30] during both training and test phases to generate predictions and uncertainty estimates from the network. SRTs like dropout [31], multiplicative Gaussian Noise [32] and dropConnect [33] are popular methods of regularising deep learning models. These methods change the model output in a stochastic manner in every run. As a result, the loss of the model is also randomly affected and is then optimised using non-convex optimisation techniques. Due to the high number of parameters which need to be optimised in any deep neural network, the models are often susceptible to overfitting in the absence of large amounts of training data. Hence, these regularisation techniques are quite popularly used in the deep learning community to prevent the models from overfitting.

In our work, we will focus primarily on one of these SRTs, namely dropout [31]. In this technique, a binary variable is associated with every unit of the network. The binary variable in layer $i$ of the network takes the value 1 with a probability $p_i$. When the network is evaluated for a given input, if the binary variable for a unit takes the value 0, the unit is dropped. In deterministic networks, dropout is generally applied only during the training phase. Intuitively, due to several units being dropped out in each iteration of training, there are multiple “thinned” networks which are trained. During the test phase, dropout is no longer applied and the network behaves as an average of all the thinned networks which were trained.

In the works [3], [29], the authors have used dropout to perform approximate inferencing in convolutional neural networks without the need for any additional model parameters. If the network has $L$ layers, the set of parameters $\theta$ contains the weights from all the layers in the network, i.e., $\theta = (W_i)_{i=1}^L$ where $W_i$ is the set of weights in layer $i$. Now the posterior $p(\theta | X, Y)$ is not tractable and hence, a variational distribution to approximate the posterior is defined as $q(W_i)$ for each layer $i$ in the network:

$$W_i = M_i \cdot \text{diag}([z_{ij}]_{j=1}^{K_i}) \quad (2.6)$$
\[ z_{i,j} \sim \text{Bernoulli}(p_i) \text{ for } i = 1, \ldots, L, \ j = 1, \ldots, K_i. \]

In the above equations, \( K_i \) is the number of units in the \( i \)th layer, \( z_{i,j} \) are the Bernoulli random variables parameterised with probabilities \( p_i \) and \( M_i \) are the parameters to be optimised. It is interesting to note here that sampling from the distribution \( q(W_i) \) is equivalent to performing dropout on layer \( i \) with the weights of the layer being \( M_i \). Hence, samples from the posterior distribution can be drawn by performing multiple Monte-Carlo runs on the network with the same input and with dropout on each layer. These samples can be used to approximate the mean and variance of the predictive distribution.

In a nutshell, once the network is trained with an SRT like dropout, for a given input \( x^* \), we can obtain the predictive mean \( \mathbb{E}[y^*] \) and the predictive variance \( \text{Var}[y^*] \) by running the network with the same input \( x^* \) multiple times (say \( T \) times) with the SRT being active in the network (i.e., treating as if the network is performing stochastic forward passes in training). The \( T \) forward passes produce \( T \) outputs \( \{\hat{y}_1^*, \hat{y}_2^*, \ldots, \hat{y}_T^*\} \). As discussed above, these outputs are samples from the predictive distribution and hence, we can estimate the mean and variance (or uncertainty) from the samples as:

\[
\mathbb{E}[y^*] \approx \frac{1}{T} \sum_{t=1}^{T} \hat{y}_t^*, \tag{2.7}
\]

\[
\text{Var}[y^*] \approx \tau^{-1} I_D + \frac{1}{T} \sum_{t=1}^{T} \hat{y}_t^* \hat{y}_t^* - \mathbb{E}[y^*] \mathbb{E}[y^*]. \tag{2.8}
\]

In equation (2.8), \( I_D \) represents the identity matrix of dimension \( D \). Thus, with the addition of dropout layers during both the training and test phases, we can draw uncertainty information from the network without having to train any additional parameters. In the next section, we will be looking at the computer vision problem of semantic image segmentation to which we will be applying BDL techniques.

### 2.2 Semantic Segmentation

Semantic segmentation is one of the most interesting and difficult problems in the field of computer vision. It involves a pixel-level understanding of images where, for a given image, the task is to classify each pixel to its corresponding object. This kind of prediction is also known as dense prediction in the literature. As with most problems
in computer vision, deep learning has had tremendous success in semantic segmentation. In this section, we discuss some of the important and relevant architectures which have had a significant impact on current research related to this problem.

2.2.1 Fully Convolutional Networks (FCN)

The work [7] by Long et al. was the first of its kind where a convolutional network without any fully connected layers was trained in an end-to-end manner directly mapping images to their corresponding segmentation maps. In conventional CNN architectures, the positional information in an image is often ignored, mainly through pooling and fully-connected layers. This is okay for tasks like classification or object detection. However, for dense predictions, this positional information needs to be retained in the network.

It is worth noting that a fully connected layer can be viewed as a convolutional layer where the filter size covers the entire input dimensions. If we were able to reduce these filter sizes, we would end up with fully convolutional networks. Even in an FCN, the output dimensions are reduced either due to convolutions with stride greater than 1 or due to pooling layers. Hence, the output is coarse and needs to be resized to original input dimensions. In their paper [7], the authors have discussed a few ways of doing this including bilinear interpolation, de-convolutional or upsampling layers and the “shift and stitch” technique.

The paper discusses the performances of well-known networks like AlexNet [34], VGG [35] and GoogLeNet [36] when converted into FCNs and augmented with upsampling layers to generate proper segmentation maps from coarse outputs. It also proposes an FCN architecture which introduces skip connections from previous layers and combines the output of both deep and shallow layers to achieve fine-grained segmentation. This architecture has been presented in Figure 2.1.

It is interesting to note that the segmentation maps obtained by combining the outputs of pool3 with 2x upsampled outputs of pool4 and 4x upsampled outputs of conv7 led to much finer results as compared to the outputs of just the conv7 layer. This shows that pooling layers in a network do in fact hinder the performance when it comes to dense predictions. In order to get around this issue, researchers have developed two trains of thought: the encoder-decoder architecture and the dilated or atrous convolutions. We will discuss these architectures in the next sections.
2.2.2 Encoder-Decoder architecture

In the encoder-decoder style of architectures there are two parts to a network, the encoder and the decoder. The encoder generally consists of the conventional CNN style of conv-pool repetitions. The causes a reduction in the spatial resolution of the resulting output. The decoder’s task is to upsample the outputs till the original resolutions are obtained back. The difference between the upsampling in the FCN and the encoder-decoder architectures is that the decoder framework is a lot more expansive and is often almost a mirror image of the encoder framework.

Two of the most well-known architectures in the encoder-decoder category are the U-Net [8] and the SegNet [24]. A pictorial representation of the U-Net is provided in Figure 2.2. In the figure, the blocks represent the feature maps and the arrows represent the computations performed (like convolution or pooling). U-Net follows the FCN philosophy and has no fully connected layers. The encoder contains multiple blocks each having convolutional layers with 3x3 filters and ReLU [37] non-linearity, followed by a 2x2 max pooling layer with a stride of 2. The pooling layers reduce the dimension of the feature maps by half. The decoder consists of blocks each of which have an upsampling de-convolution layer which doubles the spatial dimensions of the feature map and reduces the number of channels by half. The remaining half of the channels are transferred through skip connections from the encoder framework, thereby maintaining a memory of the positional information which was lost in the encoder due to pooling. Finally, a 1x1 convolution is applied at the end of the network to have the number of channels equal the number of classes.
The SegNet [24] architecture also follows the encoder-decoder framework with the encoder having 13 convolutional layers which are taken from the first 13 layers of a pre-trained VGG-16 [35] network. Each encoder block has 2 or 3 conv layers followed by a single max-pooling layer similar to U-Net. The convolution layers are accompanied by batch normalisation [38] and element-wise ReLU. The decoder in this case is essentially a mirror image of the encoder where each decoder has an upsampling layer next to 2 or 3 conv layers. The upsampling technique is different in SegNet and makes use of pooling indices which are transferred through the encoder. Finally, the output of the last convolutional layer is passed through a softmax function. Thus, the encoder-decoder architecture is a popular way by which positional information which is lost through pooling can be restored in the network. The next section discusses a way by which one can avoid pooling layers in the network altogether.

### 2.2.3 Atrous Convolutions

In a typical convolutional network, the pooling layer plays the role of identifying if a certain feature exists in a region of the image while ignoring the information of where in the aforementioned region, the feature can be detected. Letting go of this positional information is fine for tasks which involve classifying or detecting an object. However, as mentioned before, the “where” information is quite important for dense
prediction tasks. The goal of the pooling layers however, is to increase the field of view while detecting features in an image. Dilated/atrous convolutions [39] is a way by which the field of view can be increased exponentially without having to decrease the spatial resolution of the image.

In order to understand how atrous convolutions work, let us consider the simple one-dimensional case. Let the output for index \( i \) be \( y[i] \) with the input \( x[i] \) and a filter \( w[k] \) of length \( K \). A convolution operation to compute \( y[i] \) would then be as follows:

\[
y[i] = \sum_{k=1}^{K} x[i + k]w[k].
\] (2.9)

An atrous convolution is a generalisation of equation (2.9) where an additional parameter known as the atrous rate \( r \) is introduced. In this method, \( y[i] \) is computed as:

\[
y[i] = \sum_{k=1}^{K} x[i + r \cdot k]w[k].
\] (2.10)

Clearly, the normal convolution operation is the same as atrous convolution with rate set to 1. An illustration of atrous convolutions in the 1-D case is provided in Figure 2.3. In the case of a 2-D image, the general method is to have a filter with holes in between. These holes in the filter are filled with values of 0. For instance, a 3x3 filter which has a hole of one pixel between any two consecutive filter values covers a 5x5 region of the image. The number of parameters to learn and the number of operations to perform still remain the same. In a nutshell, atrous convolutions provide an efficient way to have a large field of view over the image for feature detection at different scales, without reducing the spatial dimensions of the image. Hence, it is very useful for tasks like semantic segmentation. The DeepLab-v3 [26] architecture that we extend in this work uses atrous convolutions instead of pooling layers.

2.3 Bayesian SegNet

In this section, we discuss an interesting work which has applied dropout to the SegNet [24] architecture to generate both predictions and uncertainty estimates from the network. The Bayesian SegNet [2] extends the traditional encoder-decoder structure of SegNet by inserting dropout layers typically at the end of every encoder and decoder block throughout the network. All the dropout layers have a dropout rate of
During the test phase, for each given input, the model is run multiple times to sample points from the predictive distribution. The mean of these samples is taken as the prediction and the variance provides the uncertainty values. Each model prediction is a softmax output where the spatial dimensions (i.e., height and width) are similar to the input and the number of channels is equal to the number of classes. An \textit{argmax} operation over the channels of the softmaxed outputs provides the labels for each pixel. Furthermore, the uncertainty values are also obtained pixel-wise and hence, an uncertainty map can be generated.

It is worth noting that the paper [2] proposes several probabilistic variants of the Bayesian SegNet architecture. In an ideal BDL scenario, there should be a dropout layer after every convolutional layer. However, as observed in the paper, such a high number of dropout layers would result in a very strong regularisation of the network leading to slow training. Hence, dropout layers are inserted only in certain regions of the network and the choice of where to insert dropout layers produces multiple probabilistic variants of the Bayesian SegNet which are listed as follows:

- **Bayesian Encoder**: Dropout layers are inserted after every encoder block.
- **Bayesian Decoder**: Dropout layers are inserted after every decoder block.
- **Bayesian Encoder-Decoder**: Dropout layers are inserted after every encoder and decoder block.
- **Bayesian Center**: A single dropout layer is inserted between the central encoder-decoder pair.
• **Bayesian Central Four Encoder-Decoder:** Dropout layers are inserted after the central four encoder and decoder blocks.

• **Bayesian Classifier:** A single dropout layer is inserted after the last decoder block.

Experimental results revealed that the *Bayesian Central Four Encoder-Decoder* outperformed other variants in terms of the class average accuracy and the intersection-over-union [7] metrics. This is indicative of the fact that the shallow layers of the network which capture low level visual features like edges and corners are consistent across the distribution of different models and hence can be represented through deterministic weights. In our work, we also create multiple probabilistic variants of the DeepLab architecture and compare their individual performances. These variants will be discussed in the next chapter.

The paper on Bayesian SegNet [2] also discusses some interesting metrics and tools to evaluate a BDL model trained for semantic segmentation. Some of these are as follows:

1. **Model accuracy vs confidence:** In this method, we compare the accuracy for pixels which have been predicted at different levels of confidence. For instance we check the prediction accuracy for the most confident 10% of pixels, the most confident 50% of pixels, up to all pixels (i.e., the most confident 0% of pixels). This can provide us a graph of prediction accuracy vs uncertainty. Ideally, with increase in uncertainty, the prediction accuracy should drop. Furthermore, if the rate of the drop is low, it indicates that although the model is losing confidence in its predictions, the accuracy is still not decreasing significantly. Hence, a steeper drop on this plot might indicate a worse model than a gentler drop.

2. **Segmentation maps and Uncertainty maps:** This method can be used as a visual aid to understand the regions of the image where the model is least confident. In segmentation maps we shade or colour each pixel based on the object to which it has been classified. In uncertainty maps, we shade each pixel based on how uncertain the model is about its prediction on that pixel. Darker shades generally mean a higher level of uncertainty. Such maps can also help us distinguish between different kinds of uncertainty, i.e., uncertainty due to noisy data or multiple model parameters having similar loss.
In our work, we discuss the performance of the DeepLab model variants based on the above metrics. We also develop new metrics for easily comparing the performance of different BDL models trained on the task of semantic segmentation.
Chapter 3

Bayesian DeepLab

In the previous chapter, we discussed some preliminaries including an explanation of Bayesian Deep Learning as well as the types of CNN architectures that have been developed for the task of semantic segmentation. This chapter describes the state-of-the-art DeepLab v3 [26] which we extend into a probabilistic network, the Bayesian DeepLab using the MC dropout [3] inference technique. The chapter also presents multiple variants of the Bayesian DeepLab architecture and ends with a discussion of the metrics which we use to generate pixel-wise uncertainty maps from any given input image.

3.1 DeepLab v3

DeepLab v3 [26] is a state-of-the-art network specifically designed for the task of semantic segmentation. There have been multiple versions of DeepLab [25], [26], [40] over the years. However, all the versions possess certain common architectural features. Firstly, DeepLab makes use of atrous convolutions to widen the field of view on the input without increasing the number of parameters or computations. The second idea is to have a robust mechanism to segment objects at multiple scales of the image. For instance, in the autonomous driving scenario, a car which is far away and appears tiny and one which is near and appears large, both represent the same object but at very different scales. Multi-scale processing of images can be performed using an image-pyramid in which parallel CNNs are trained using images which are scaled to different levels. This method is quite computationally expensive simply due to the larger number of parameters to train. In the work [25], the authors propose using atrous convolutions with different atrous rates as a mechanism to perform multi-scale segmentation without significant increase in the number of parameters. The technique known as Atrous Spatial Pyramid Pooling (ASPP) will be discussed in detail.
in this section. Finally, the output of the network is refined to improve localization performance and have fine-grained segmentation. In the older DeepLab versions, this was done through a fully connected Conditional Random Field (CRF) [41]. However, in DeepLab v3, the localization and upsampling of network outputs is done through a small decoder module.

The above mentioned features can be implemented on different architectures and DeepLab versions v1 and v2 use popular image classification networks like VGG-16 [35] and ResNet-101 [42] as the backbone framework. DeepLab v3 uses the Xception architecture [43] and in this work, we define the Bayesian DeepLab variants based on this network. This section is dedicated to discussing the DeepLab v3 architecture in detail.

### 3.1.1 Xception Module

A normal convolutional layer learns parameters and filters in a 3-D space where the two spatial dimensions include the width and the height of the input image and the third dimension is the channel dimension. The philosophy behind the Inception module [36] is to simplify this operation by separating the convolution across the spatial dimensions from the one across the channel dimension. A 1x1 convolution generally convolves across the channel dimension and a larger filter size (like 3x3 or 5x5) convolves across all dimensions (i.e., both the spatial and the channel dimensions). Since in an Inception module, a 3x3 or 5x5 filter generally follows a 1x1 filter, it is not able to completely separate the convolutions across the spatial and channel dimensions.

An Xception (the name comes from “extreme inception”) module makes use of depthwise-separable convolutions [44] to separately map the spatial and channel correlations. A depthwise-separable convolution (also known as a separable convolution in deep learning literature) has two components. First, there is a depthwise convolution which uses a 3x3 or larger filter independently on each channel of the input, thereby convolving across the spatial dimensions. Next, there is a pointwise convolution which uses a 1x1 filter to convolve across the channel dimension. It is worth noting that there may or may not be a non-linearity operation like ReLU in between the depthwise and pointwise convolutions. However, in general, an element-wise non-linearity is applied before the separable convolution layer in an xception module. An illustration of the depthwise-separable convolution operation has been provided in Figure 3.1.
The Xception module is particularly appealing because it combines the simplicity of the VGG architecture with the advantages of ResNet-like skip connections and the philosophy of the Inception module where convolutions across the spatial and channel dimensions are completely separate. The Xception network has three parts: the entry flow, the middle flow and the exit flow. Each of these parts have multiple Xception modules in them. A typical module consists of three consecutive separable convolution layers stacked on top of one another with ReLU non-linearity being applied before each separable convolution. In the original Xception paper [43], the entry and exit flows contain modules where the third separable convolution layer is replaced by a max pooling layer. In our work, we do not use pooling layers and instead have a separable convolution layer with a stride of 2 to reduce the dimensions. The skip connections in the entry and exit flows contain a single 1x1 convolution layer with a stride of 2. The skip connections in the middle flow do not have any additional operations and simply add the inputs to the outputs of the separable convolution layers. Typical modules in the entry/exit flows and in the middle flow of the Xception network have been illustrated in Figure 3.2. In the original architecture [43], the entry flow has 3 modules, the middle flow has 8 and the exit flow has 2 Xception modules followed by
Figure 3.2: Xception Modules

In (a), a module in the middle flow of the Xception network is shown. All the separable convolutions have stride 1 and there are 728 channels. The output dimensions are exactly the same as the input dimensions and there is no convolution in the skip connection. In (b), a typical module in the entry flow has been shown. The last separable convolution has a stride of 2 and reduces the dimensions of the input by half. Furthermore, there is a 1x1 convolution with a stride of 2 which also reduces the dimensions of the input by half in the skip connection branch. The outputs are finally added. In the exit flow, the structure is similar to the entry flow module except that the number of channels is much higher.

fully connected layers and logistic regression. The extended version of the network which we develop has some architectural differences from the one discussed here. These will be described in a future section in this chapter. In the next section, we discuss a technique used in DeepLab to perform multi-scale image processing in a CNN without adding a significant amount of training parameters to the network.

3.1.2 Multi-scale Image Processing

One of the requirements for good performance in semantic segmentation is being able to recognise objects at different scales. In general, CNNs have proven to be good at learning features at varying scales if the training data contains the objects in different sizes. However, the performance can be improved further if a voluntary effort is made to incorporate multi-scale information within the network. There are a few ways of
multi-scale image processing in CNNs.

The first method is known as the *image pyramid*. In this case, the training set is augmented by scaling the images to various levels. Multiple CNNs are then trained parallely, each network on a particular scale of images. Next, the outputs of the different networks are resized to original image dimensions using bilinear interpolation [45]. Finally, the outputs are fused by taking the maximum value from each scale for every position in the image. Although this ensembling technique does improve performance, it is computationally heavy as multiple CNNs have to be trained independently on different scales of data.

In the paper [25], the authors proposed a method to perform multi-scale image processing without the need to train CNNs parallely. As mentioned in previous sections, atrous convolutions are a way to have a wider field of view over the input without increasing computations. The higher the atrous rate, the larger the receptive field. Hence, in order to incorporate information from varying scales into a network, a given input can be processed parallely in multiple branches using convolutions with different atrous rates. The outputs of these convolutions can be further processed in their respective branches and then concatenated and passed through a 1x1 filter to get the desired number of channels. Clearly, this method, known as *Atrous Spatial Pyramid Pooling* (ASPP), uses significantly less trainable parameters as compared to the *image pyramid* technique. ASPP can also be tweaked with certain modifications. For instance, one can use separable atrous convolutions instead of normal atrous convolutions [40] in the ASPP module.

Finally, atrous convolutions can also be used in a cascaded structure to gather information at multiple scales. The deeper layers or blocks of the network use convolutions with progressively higher atrous rates. *Cascaded atrous modules* was proposed in the DeepLab v3 paper [40] but we do not use them for the sake of simplicity. In our work, we perform ASPP with three different atrous rates, 6, 12 and 18 and the ASPP module is applied on the output of the main Xception network. A diagrammatic representation of ASPP has been provided in Figure 3.3 for the reader’s convenience. The *output stride* (i.e., the ratio of the input dimensions to the output dimensions) of the network is set as 16 as the resolution is halved at every conv layer having a stride of 2. Thus in order to upsample the outputs, a small decoder framework is used the structure of which we elaborate in the next section.
3.1.3 Decoder Module

The entry and exit flows of the original Xception network [43] contain max pooling layers with a stride of 2. Although we haven’t used pooling layers, we still apply a double stride in the separable convolution layers of the entry and exit flows of our network. Each of these reduce the spatial dimensions of the input by half and the existence of multiple such layers in the network creates a significant reduction in the dimensions of the output. To be precise, the input dimensions of the image are 16 times the output dimensions in our network. One can argue that the all the layers should have a stride of 1. However, in that case, the number of learnable parameters increases to the point that with limited GPU memory, it becomes computationally prohibitive to train such networks. Thus, we cannot do away with layers having a stride of 2 and in order to restore the original dimensions of the image, we apply a decoder module.

As discussed in Chapter 2, the simplest way to restore the image dimensions is to use an upsampling technique like bilinear interpolation. However, it has been observed that a more expansive decoder module which uses skip connections from the encoder feature maps, produces sharper segmentation results. In our work, we use the simple decoder architecture proposed in [40] to upsample the feature maps. This decoder
essentially combines the output of the main network (i.e., the ASPP module) with the feature maps produced by the second entry flow of the Xception network. The ASPP module produces 16x downsized feature maps and the second entry flow produces 4x downsized feature maps. The number of channels in the latter is very high and is first reduced through a 1x1 filter. On a separate branch, the ASPP outputs are enlarged by a 4x upsampling layer. These two branches are then concatenated, passed through a 3x3 filter and finally upsampled by a factor of 4 again (thereby restoring the original image dimensions which had been downsized by 16x). A simple pictorial representation of the decoder module is presented in Figure 3.4.

3.2 MC Dropout in DeepLab: Bayesian DeepLab

In the previous sections, we have discussed the components of the DeepLab v3 network and armed with that knowledge, in this section, we present Bayesian DeepLab, the probabilistic version of DeepLab v3 which is able to produce both predictions and uncertainty estimates for the task of semantic segmentation. As explained in Chapter 2, the main idea behind creating a probabilistic network from a deterministic one is to use a Stochastic Regularisation Technique (SRT) which can be modeled as a distribution over the network weights. In our work, we use dropout as the SRT and insert dropout layers in different parts of the DeepLab v3 network. These layers place a Bernoulli distribution over the model weights parameterised by the dropout rate.
During test time, we keep the dropout layers active and run the network multiple times on a given input. This effectively samples points from the predictive distribution and we can treat the mean of these points as the prediction and the variance as the model uncertainty.

In an ideal scenario, a dropout layer should be inserted after every convolutional layer in the network. However, in a large network like Bayesian DeepLab which contains more than 65 convolution layers in just the main framework, insertion of dropout layers after every convolution will regularise the network to an extent that makes training infeasible. Hence, we have chosen to insert dropout layers only after some of the Xception modules in the network. This choice gives rise to multiple probabilistic variants of the Bayesian DeepLab depending on where in the network, the dropout layers are placed. Furthermore, we have also created three scale-based variants of the Bayesian DeepLab: the full-scale model, the medium-scale model and the toy model. The purpose behind the smaller scale models is essentially to enable faster training on low configuration machines as well as to be able to quickly evaluate different architectures. These variants will be discussed in detail in the next section.

We conclude the chapter with a description of the uncertainty metrics which we use for Bayesian DeepLab. Although for a regression problem, the variance of the Monte-Carlo outputs can suffice as a measure of model confidence, the problem of classification lacks this convenience due to categorical output values. Therefore, we use approximations of certain metrics which have their grounds in information theory. We study these metrics and how they capture uncertainty in the network.

### 3.2.1 Model Variants

In this section, we discuss the details of the Bayesian DeepLab architecture and its variants. However, before we get into the variants, a short note on the common structure of the model is in order. Essentially, the backbone framework is similar to DeepLab v3, i.e., the inputs are first passed through an extended Xception network followed by an ASPP module and finally a decoder module to restore the original input dimensions with sharp segmentation. It is worth noting that there are certain differences between the original Xception [43] architecture and the one which we use. The differences are listed as follows:
1. There are no pooling layers or fully connected layers in our network. We use separable convolution filters with a stride of 2 instead of max pooling layers. This helps in dense predictions. Furthermore, following the FCN [7] philosophy, our network is fully convolutional (does not have any fully connected layers) and can segment images of arbitrary sizes.

2. The middle flow in the extended Xception network has 16 modules instead of 8 as described in the original Xception paper [43].

3. An element-wise ReLU non-linearity is applied before all the separable convolution layers. This is slightly different from the original network where ReLU is not applied before separable convolution in the entry flow modules.

In the Bayesian DeepLab variants, we insert dropout layers in different parts of the Xception network with dropout rates set to 0.5, although these rates can be further optimised. The architectural differences between DeepLab v3 and Bayesian DeepLab are as follows:

1. Bayesian DeepLab networks have dropout layers at different points depending on the network variant. These dropout layers are active even during test time. DeepLab v3 has a single dropout layer after the ASPP module but the dropout is only applied during training.

2. Atrous convolutions in cascade (i.e., one after another instead of in parallel) are not applied in our network. The only place where we use atrous convolutions is in the ASPP module. Cascaded atrous modules and ASPP were both designed to detect features at varying scales. For the sake of simplicity, we stick to just ASPP. Furthermore, the ASPP and decoder modules use normal convolution layers instead of separable convolutions.

In this work, we have developed two types of variants. First, we have the scale-based variants of the Bayesian DeepLab network. The full-scaled models have 3 Xception modules in the entry flow, 16 modules in the middle flow and 2 modules in the exit flow of the network. The depth (i.e., number of channels) of each convolution/separable convolution filter in the modules is the same as the original Xception network. The medium-scaled models have a similar architecture as the full-scaled models but the depth in all the layers is reduced by a factor of 8. For instance, the number of channels in the first module of the full-scaled network is 128 and in the medium-scaled network, it is 16. This reduces the total number of learnable parameters by a factor of
8 and enables faster training. Finally, the toy models have just one Xception module each in the entry, middle and exit flows. The toy models don’t even use ASPP or decoders. The purpose of these models is to simply be able to train a network in minutes even without GPUs and see the output of different prediction and uncertainty metrics on a small scale.

The probabilistic variants of Bayesian DeepLab depend on the positions where the dropout layers are placed in the network. It is worth noting that the dropout layers are placed only at the end of certain Xception modules. There is no dropout within any Xception module. The following are the probabilistic variants:

1. **Dropout-Entry-Flow**: A dropout layer is inserted after every Xception module in the entry flow.

2. **Dropout-Exit-Flow**: A dropout layer is inserted after every Xception module in the exit flow.

3. **Dropout-Middle-Flow**: A dropout layer is inserted after every 4 Xception modules in the middle flow. Since there are 16 such modules, there are a total of 4 dropout layers.

4. **Dropout-Entry-Exit-Flow**: A dropout layer is inserted after every Xception module in the entry and exit flows.

5. **Dropout-Entry-Middle-Exit-Flow**: A dropout layer is inserted after every Xception module in the entry and exit flows and after every 4 modules in the middle flow.

A figure of the full-scale Dropout-Entry-Flow variant is illustrated in Figure 3.5. The Xception module parameters (like depth, skip connection type, existence of dropout and stride) of the full-scaled and toy variants are presented in Table 3.1 and Table 3.2 respectively. The parameters for the medium-scaled models are very similar to those of the full-scaled ones except for the depth values (as discussed above). The table for the medium-scaled models has been provided in Appendix B. In the next section, we discuss the metrics to capture pixel-wise uncertainty in Bayesian DeepLab.

26
Figure 3.5: Architecture of the Bayesian DeepLab full-scale Dropout-Entry-Flow variant
Table 3.1: Xception Module structures of the full-scaled probabilistic model variants of Bayesian DeepLab

The columns of the table represent the probabilistic variants of the full-scaled Bayesian DeepLab architecture. The rows correspond to the different Xception modules in the Bayesian DeepLab network in sequence. In each cell, we provide the architectural parameters of the Xception module in the corresponding variant. The Channels parameter is a list providing the depth (i.e., the number of channels) in each of the three separable convolution layers in the module. The Skip parameter represents the type of skip connection in the module. The Conv and Sum operations represent the convolution output and the skip connection. Finally, the Stride parameter indicates the stride of the last separable convolution layer as well as the 1x1 filter in the skip connection if any. A stride of 2 will reduce the spatial dimensions of the input by half. A stride of 1 keeps the dimensions unchanged.

<table>
<thead>
<tr>
<th>Xception Module</th>
<th>Dropout-Entry-Flow</th>
<th>Dropout-Middle-Flow</th>
<th>Dropout-Entry-Middle-Exit-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Flow Module 1</td>
<td>Channels: [1536, 1536, 2048]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Entry Flow Module 2</td>
<td>Channels: [728, 728, 728]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Entry Flow Module 3</td>
<td>Channels: [728, 728, 728]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Middle Flow Modules 1-4</td>
<td>Channels: [728, 128, 728]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Middle Flow Modules 5-8</td>
<td>Channels: [1536, 1536, 2048]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Middle Flow Modules 9-12</td>
<td>Channels: [728, 728, 728]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
<tr>
<td>Middle Flow Modules 15-16</td>
<td>Channels: [728, 728, 728]</td>
<td>Dropout: Yes</td>
<td>Dropout: No</td>
</tr>
</tbody>
</table>

Table 3.2: Xception Module structures of the toy model variants of Bayesian DeepLab

The columns of the table represent the probabilistic variants of the toy Bayesian DeepLab architecture. The rows correspond to the different Xception modules in the toy network in sequence. Each cell of the table says if the corresponding module contains a dropout layer at the end. All the separable convolution layers have a depth of 16. The entry flow skip connection has a 1x1 filter, the middle flow skip connection is a simple sum operation and the exit flow has no skip connection. The stride of the entry flow is 2 and the remaining modules have stride 1.

<table>
<thead>
<tr>
<th>Xception Module</th>
<th>Dropout-Entry-Flow</th>
<th>Dropout-Middle-Flow</th>
<th>Dropout-Middle-Exit-Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Flow Module 1</td>
<td>Dropout: No</td>
<td>Dropout: No</td>
<td>Dropout: Yes</td>
</tr>
<tr>
<td>Middle Flow Module 1</td>
<td>Dropout: No</td>
<td>Dropout: No</td>
<td>Dropout: Yes</td>
</tr>
<tr>
<td>Exit Flow Module 1</td>
<td>Dropout: No</td>
<td>Dropout: No</td>
<td>Dropout: Yes</td>
</tr>
</tbody>
</table>

3.2.2 Uncertainty Metrics

Bayesian Deep Learning depends on SRTs like dropout and multiplicative Gaussian noise to model distributions across network weights. Once trained, samples from the posterior distribution can be drawn by running the network multiple times on the
same input. These samples can be evaluated to get the mean and variance as predictions and model uncertainty respectively. However, there can be multiple types of model uncertainty. Noisy data is a common phenomenon in machine learning and this can lead to an uncertainty known as aleatoric uncertainty. The model can also be uncertain about which parameter set (i.e., the set of network weights) to converge to, if multiple sets of parameters explain the data equally well. This kind of uncertainty is known as model uncertainty or epistemic uncertainty. Both of these contribute to the final predictive uncertainty.

In case of regression problems, since the output is real valued, the variance itself can be used as a metric of uncertainty. However, as we are dealing with the problem of pixel-wise classification, we have to use more sophisticated metrics that can work with categorical outputs. In the work [29], the authors suggested some useful information theoretic metrics which can be used as measures of uncertainty. In our work, we use the predictive entropy and the mutual information metrics. The predictive entropy captures the average information present in a predictive distribution and is given as follows:

$$H[y|x, D_{\text{train}}] = -\sum_c p(y = c|x, D_{\text{train}}) \log p(y = c|x, D_{\text{train}})$$  \(\text{(3.1)}\)

where \(y\) is the output, \(x\) is the input, \(D_{\text{train}}\) is the training data and \(c\) covers the classes. The sum is over all possible classes. The above metric can be approximated by using the outputs obtained from the Monte-Carlo runs. Let’s say there are \(T\) such runs. Then we can replace \(p(y = c|x, D_{\text{train}})\) in equation 3.1 with \(\frac{1}{T} \sum_t p(y = c|x, \hat{w}_t)\), where \(p(y = c|x, \hat{w}_t)\) is the softmax probability of input \(x\) being in class \(c\) and \(\hat{w}_t\) are the model parameters. The approximated predictive entropy is as follows:

$$\hat{H}[y|x, D_{\text{train}}] = -\sum_c \left(\frac{1}{T} \sum_t p(y = c|x, \hat{w}_t)\right) \log \left(\frac{1}{T} \sum_t p(y = c|x, \hat{w}_t)\right).$$  \(\text{(3.2)}\)

The second metric we use is mutual information between the prediction \(y\) and the posterior distribution over parameters \(w\). The mutual information metric is given as follows:

$$I[y, w|x, D_{\text{train}}] = \mathbb{H}[y|x, D_{\text{train}}] - \mathbb{E}_{p(w|D_{\text{train}})}[\mathbb{H}[y|x, w]].$$  \(\text{(3.3)}\)
Mutual information can also be approximated in a similar way to predictive entropy as follows:

\[
\hat{I}[y, w|x, D_{\text{train}}] = -\sum_c \left( \frac{1}{T} \sum_t p(y = c|x, \hat{w}_t) \log \left( \frac{1}{T} \sum_t p(y = c|x, \hat{w}_t) \right) + \frac{1}{T} \sum_{c,t} p(y = c|x, \hat{w}_t) \log p(y = c|x, \hat{w}_t) \right) .
\]  

(3.4)

It is quite interesting to note that predictive entropy and mutual information reflect different types of uncertainty. Let us consider the example which has been discussed in [29]. We have the outputs of multiple stochastic forward passes for a binary classification system classifying an image as a cat or a dog. The three softmax output cases for the class of dog are as follows:

1. All the probability vectors collected are (1,0), (1,0), .... (1,0).
2. All probability vectors have value 0.5, i.e., values collected are (0.5, 0.5), (0.5, 0.5), ..., (0.5, 0.5).
3. Half the vectors are (1, 0) and the other half are (0, 1).

It is quite clear that case 1 signifies high confidence. In cases 1 and 2, the model is confident on its output and produces the same output in all the stochastic runs. In case 3, the model is not confident on its output and produces different values in different runs. This is a case of \textit{model uncertainty}. On the other hand both cases 2 and 3 show \textit{predictive uncertainty}. In case 1, both mutual information and predictive entropy values are 0 which indicates low uncertainty and high confidence. In case 3, both mutual information and predictive entropy return value 0.5 which indicates high uncertainty. However, for case 2 predictive entropy value is 0.5 but mutual information is 0. Thus, we can say that predictive entropy captures predictive uncertainty but mutual information captures model uncertainty (or epistemic uncertainty).

In the Bayesian DeepLab network, we use the mutual information and predictive entropy metrics to capture pixel-wise uncertainty and generate uncertainty maps. In fact, uncertainty maps provides a good visual aid to understand how a model is training. We study uncertainty maps as one of the tools to compare BDL models for semantic segmentation. This, along with several other evaluation metrics will be discussed in the next chapter.
Chapter 4

Evaluation Metrics and Alternative Inference Techniques

The primary purpose of our project is to develop mechanisms for measuring the performance of Bayesian models which have been designed for the problem of semantic segmentation. Having described the architectural features of the Bayesian DeepLab variants, in this chapter, we are ready to discuss the evaluation techniques which we have studied and developed for this purpose. The first part of the chapter deals with metrics which can be used as sanity checks and comparison tools for Bayesian models. In the latter part of the chapter, we describe a few different ways of generating uncertainty values from deep networks (other than vanilla dropout layers). We apply these techniques on the Bayesian DeepLab architecture to compare how they perform versus MC dropout. This study helps in setting up baselines and benchmarks for researchers to compare new BDL models.

4.1 Metrics

This section describes evaluation metrics which can be used to compare the performance of different Bayesian models designed for semantic segmentation. Firstly, we use three metrics which measure prediction performance by comparing the ground-truth labels with the predicted labels. These metrics are quite heavily used in the semantic segmentation literature and can act as a good indicator of how well the model is performing as far as pixelwise prediction is concerned. Secondly, we study metrics which compare model accuracy with model uncertainty. The idea is that the model should be more confident on pixels which it predicts accurately and less confident on ones where it makes mistakes. These metrics thus serve as a sanity check for BDL networks, i.e., they can be used to ensure that the model’s behaviour
in terms of accuracy and uncertainty is expected. Finally, based on the intuition of the inverse relationship between model accuracy and uncertainty, we finally introduce three novel metrics which can serve as an easy and effective comparison method for the performance of Bayesian neural networks designed for semantic segmentation. It should be noted here that all the metrics which are described in this section work on labels, predictions and uncertainty maps which have the same spatial dimensions (i.e., height and width) as the image.

4.1.1 Semantic Segmentation Metrics

In the paper on FCNs [7], the authors proposed a few metrics for measuring the performance of models designed for dense prediction tasks. We use three of their metrics: pixel accuracy, mean (class-wise) accuracy and mean I/U (intersection-over-union).

The pixel accuracy is the total fraction of pixels over all classes which are correctly predicted. Let there be $C$ classes in total. For classes $i$ and $j$, let $n_{ij}$ represent the number of pixels which belong to class $i$ but have been predicted as class $j$. Also, for the sake of notational convenience, let $t_i$ denote the number of pixels which belong to class $i$ according to the ground truth labels. Clearly $t_i = n_{i1} + n_{i2} + ... + n_{iC}$. In that case, the pixel accuracy for a single (label, prediction) pair is given as:

$$\text{pixel\_accuracy} = \frac{\sum_{c=1}^{C} n_{cc}}{\sum_{c=1}^{C} t_c}. \quad (4.1)$$

The mean accuracy is computed by averaging the individual accuracies of each class. For a single class $c$, the accuracy of the class is given as $\frac{n_{cc}}{t_c}$. Then, the mean accuracy is computed as:

$$\text{mean\_accuracy} = \frac{1}{C} \sum_{c=1}^{C} \left( \frac{n_{cc}}{t_c} \right). \quad (4.2)$$

Finally, the mean I/U (intersection-over-union) metric is similar to the mean accuracy in that it is computed separately for each class and then averaged. However, for a class $c$, it also considers the pixels which are misclassified by the model as class $c$ (i.e., the model predicts some other class as $c$) as part of the denominator. The mean I/U is given as:

$$\text{mean\_I/U} = \frac{1}{C} \sum_{c=1}^{C} \left( \frac{n_{cc}}{t_c + \left( \sum_{x \neq c} n_{xc} \right)} \right). \quad (4.3)$$

The computation of each of the above metrics for a simplistic scenario is presented in Figure 4.1 for the reader’s convenience. It should be noted that for a given test
Figure 4.1: Worked out example of evaluation metrics for semantic segmentation: pixel accuracy, mean accuracy, mean I/U

set, the above metrics are first computed for each test image and then the values are averaged over all the test images and this averaged value is reported.

4.1.2 BDL Metrics

In this section, we discuss two classes of metrics which can be used to study the behaviour of BDL models on dense prediction tasks. The first set of metrics serve as a sanity check for the model’s behaviour. They ensure that an inverse relationship exists between the model accuracy and uncertainty. These metrics however are not directly indicative of the model’s performance. We use the inverse accuracy-vs-uncertainty
intuition to propose the second set of metrics which can be easily and effectively used to measure the performance of Bayesian networks on dense prediction problems.

4.1.2.1 Sanity check metrics

The **class-wise accuracy vs mean uncertainty** metric first computes the *pixel accuracy* for each class in a test dataset. Next, it calculates the average uncertainty of all the pixels per class. These values can then be plotted against each other to create a class-wise accuracy vs mean uncertainty scatter plot. The pixel accuracy can be computed globally over all the test images or can also be computed separately for each test image and then averaged. Similarly, for the **class-wise frequency vs mean uncertainty** metric, the total number of pixels in a test set for each class is plotted against the mean uncertainty value for that class (creating the class-wise frequency vs mean uncertainty scatter plot). Both of these metrics are meant to confirm the inverse relationship between the prediction accuracy and the uncertainty of a model. It is expected that the classes which have higher accuracy should be predicted with low uncertainty. Similarly, the classes which occur more in the dataset should have higher confidence values from the model. Therefore, the class-wise accuracy and frequency should be positively correlated.

The **accuracy for top x% most certain pixels** is yet another sanity check for the inverse relationship between accuracy and uncertainty of a model. However, this metric does not work on a class-wise basis. Given a threshold uncertainty, we compute the mean accuracy of only those pixels in the test set which the model has predicted with uncertainty lower than the threshold (i.e., where the model is more confident). Clearly, the accuracy for a set of pixels which have been predicted with greater confidence should be high. Hence, with decrease in the threshold of uncertainty, we expect the prediction accuracy to increase, thereby creating an inverse curve between accuracy and the count of the top x% most certain pixels (x ranging from 10 to 100 say). With this metric however, we can make an additional comment about model performance. As we measure the accuracy of more pixels which the model is uncertain about, the model which still retains a higher overall accuracy is better. It reflects the idea that a model which correctly knows the prediction it is unsure about is still a better model than one which makes mistakes on the predictions it is less confident on. Having said that, we still keep this metric as just a sanity check because it would be unfair to penalise a model on incorrect predictions where the model confidence is low. An illustrative example of the **class-wise accuracy/frequency vs mean uncertainty** and
4.1.2.2 Performance Evaluation Metrics

The metrics which were discussed in the previous section provide insight into the workings of a BDL model and also help in intuitively explaining the model’s outputs. However, we are also interested in comparing one BDL model against another. As the above metrics don’t directly quantify the model performance, in this section we present three metrics to measure the performance of a Bayesian model:

1. \( P(\text{accurate}|\text{certain}) \): The conditional probability that the model is accurate on its output given that it is confident on the same.

2. \( P(\text{uncertain}|\text{inaccurate}) \): The conditional probability that the model is uncertain about its output given that it has made a mistake in its prediction (i.e., is inaccurate).

3. \textbf{Patch Accuracy vs Patch Uncertainty (PAVPU) Accuracy}: A metric which is a measure of similarity between regions of the test images where the model is accurate and confident as well as the regions where the model is inaccurate and uncertain. We define regions as square patches or windows in

The accuracy for top x% most certain pixels for a simple case has been presented in Figure 4.2.
the image with variable patch dimensions. Hence, this metric can also be computed at a pixel level by setting the patch dimensions to 1x1. An ideal model should have a positive regional correlation between the high accuracy and high confidence regions of its outputs.

It is worth noting at this point that we are not evaluating the model performance in the conventional sense of the term. Conventional performance measures deal only with the model predictions compared with the ground-truth labels. Instead we are trying to study the relation between how accurate the model is on a given input vs how uncertain it is on the same input. The basic assumption is that: if a model is confident on its output, it should be accurate on the same. This assumption also implies that if a model is inaccurate on an output, it should also be uncertain about the same output. These two assumptions are directly reflected in the first two conditional probabilities mentioned above where \(P(\text{accurate}|\text{certain})\) is a measure of model accuracy on confident outputs and \(P(\text{uncertain}|\text{inaccurate})\) reflects model uncertainty over inaccurate outputs. Finally, we combine the two good cases (i.e., (accurate, certain) and (inaccurate, uncertain)) into a single metric where we measure the number of patches/windows in an image which are good cases vs the total number of patches in the image.

In order to implement the above ideas, we first choose a patch/window size \(w\) and traverse the predicted labels, actual labels and the uncertainty maps with patches of size \(w \times w\), much like the traversal in a convolution layer but with disjoint receptive fields. Next, we compute the accuracy of each patch obtained from the predicted and actual labels. If this accuracy is above a certain threshold, we mark the patch as “accurate”. Since the accuracy of a patch is always between 0 (none of the pixels in the patch have been correctly predicted) and 1 (predictions on all pixels in the patch are accurate), choosing the threshold of accuracy is relatively simple. Similarly, from the corresponding patch obtained in the uncertainty map, we compute the average uncertainty. If this uncertainty is above a given threshold, we classify the patch as “uncertain”. A small comment on the uncertainty threshold is in order. Since the uncertainty values can theoretically be any real number depending on the uncertainty metric, there are multiple ways of choosing the uncertainty threshold. One simple way would be to find the average uncertainty of all pixels over a test set and use that value as the threshold. Other ways could be to compute the minimum \((mn)\) and maximum \((mx)\) uncertainty values and choose a certain threshold \((t)\) (between 0 and
1) in between. In that case, the uncertainty threshold \((th)\) is computed as:

\[
th = mn + (t(mx - mn)). \tag{4.4}
\]

Once, the entire dimensions of the image have been covered, we end up with two binary maps: the accuracy map and the uncertainty map each having dimensions equal to the image dimensions reduced by a factor of \(w\). Next, we construct a confusion matrix containing the number of patches which are accurate and certain \((n_{ac})\), accurate and uncertain \((n_{au})\), inaccurate and certain \((n_{ic})\) and inaccurate and uncertain \((n_{iu})\). Finally, we can report the following three metrics as follows:

\[
P(\text{accurate}|\text{certain}) = \frac{n_{ac}}{n_{ac} + n_{ic}} \tag{4.5}
\]

\[
P(\text{uncertain}|\text{inaccurate}) = \frac{n_{iu}}{n_{ic} + n_{iu}} \tag{4.6}
\]

\[
\text{PAVPU Accuracy} = \frac{(n_{ac} + n_{iu})}{(n_{ac} + n_{au} + n_{ic} + n_{iu})}. \tag{4.7}
\]

The PAVPU Accuracy metric has been named such because it represents the operation of computing the accuracy (i.e., \(\frac{TP}{TP + TN}\), where TP = True Positive, FP = False Positive, TN = True Negative and FN = False Negative) from a conventional confusion matrix. Clearly, a model with a higher value of the above metrics is a better performer. The pseudocode to compute the metrics has been presented in Algorithm 1. In the algorithm, \(\text{shape}()\) returns the (height, width) pair of a given 2-D array, \(\text{init_zeros}()\) returns a new array of a given shape initialised with zeros at all indices, \(\text{get_window_from_array}()\) gets a window/patch of a given set of dimensions from a 2-D array, \(\text{get_pixel_accuracy}()\) returns the pixel accuracy from a given pair of labels and predictions and \(\text{get_mean_uncertainty}()\) returns the average of a given uncertainty map. The remaining method names in the algorithm are self-explanatory.

It is worth noting here that the values of the above metrics are dependent on three parameters: the accuracy threshold, the uncertainty threshold and the patch/window dimensions. However, for the sake of simplicity, in our experiments we keep the patch dimensions fixed at 4x4 and the accuracy threshold at 0.5. This allows us to observe how the PAVPU Accuracy changes with varying thresholds of uncertainty. As we prefer higher values of the metric, we can also report the peak value of the curve from such plots as well. These plots and other results will be discussed in the next chapter. An illustrative example to compute the three evaluation metrics has been provided in Figure 4.3 for the reader’s convenience. In the next section, we discuss two alternative
techniques of performing approximate inferencing in deep neural networks. We experiment on the inference techniques using the metrics we have developed in this section.

**Algorithm 1: Patch Based Evaluation Metrics**

**Input**: $l$: ground truth label, $p$: predictions, $u$: uncertainty values, $w$: window dimension, $ath$: accuracy threshold, $uth$: uncertainty threshold

**Output**: tuple containing the three patch based evaluation metrics

```plaintext
ht, wd ← shape(l)
red_wd ← wd/w, red_ht ← ht/w
bin_acc_map ← init_zeros((red_ht, red_wd))
bin_unc_map ← init_zeros((red_ht, red_wd))
for h_index ← 0 to (red_ht − 1) do
    for w_index ← 0 to (red_wd − 1) do
        window_label ← get_window_from_array(l, h_index, w_index)
        window_prediction ← get_window_from_array(p, h_index, w_index)
        window_uncertainty ← get_window_from_array(u, h_index, w_index)
        pixel_acc ← get_pixel_accuracy(window_label, window_prediction)
        if pixel_acc > ath then
            bin_acc_map[h_index][w_index] ← 1
        else
            bin_acc_map[h_index][w_index] ← 0
        end
        mean_uncertainty ← get_mean_uncertainty(window_uncertainty)
        if mean_uncertainty > uth then
            bin_unc_map[h_index][w_index] ← 1
        else
            bin_unc_map[h_index][w_index] ← 0
        end
end
nac ← num_accurate_certain(bin_acc_map, bin_unc_map)
nu ← num_accurate_uncertain(bin_acc_map, bin_unc_map)
iic ← num_inaccurate_certain(bin_acc_map, bin_unc_map)
iuin ← num_inaccurate_uncertain(bin_acc_map, bin_unc_map)
p(accurate|certain) ← \left(\frac{nac}{nac + nic}\right)
p(uncertain|inaccurate) ← \left(\frac{niu}{niu + niuc}\right)
pavpu_accuracy ← \left(\frac{nac + nu + niuc + niuin}{nac + nu + niuc + niuin}\right)
return (p(accurate|certain), p(uncertain|inaccurate), pavpu_accuracy)
```
4.2 Alternative Inference Techniques

In this section, we study some alternative methods (apart from MC dropout) of generating uncertainty estimates from a deep neural network. These techniques help us to create benchmarks and baselines against which new models can be compared. There have been quite a few papers [3], [4], [5], [22], [23] etc. which deal with approximate inference techniques in deep networks. However, as we have chosen MC-dropout as our primary method of generating uncertainty values, we choose two techniques, concrete dropout [4] and deep ensembles [5] which have been motivated by the MC-dropout paper [3].

4.2.1 Concrete Dropout

In a previous chapter, we discussed dropout as a relatively convenient approximate inference technique in deep neural networks. However, each dropout layer in a network has a dropout probability $p$ as a hyperparameter. In order for a model to generate well-calibrated uncertainty estimates, the dropout probabilities should ideally be adapted or optimised based on the training data. If we follow conventional approaches of hyperparameter optimisation, we can use grid search over a validation set to find the set of optimal or near-optimal dropout probabilities in the network. However, since our task involves training a large model over multiple GPUs, grid searching
over a validation set becomes a computationally prohibitive operation. Furthermore, the total number of possible dropout configurations (each configuration being a set of probabilities, one for each dropout layer in the network) increases exponentially with rise in number of dropout layers. Under such circumstances, variants like the Dropout-Entry-Middle-Exit-Flow would require significantly higher training time than variants like the Dropout-Entry-Flow.

In the paper [4], the authors suggest a variant of the dropout layer, known as the concrete dropout layer, as a workaround to the problem of optimising dropout probabilities. The concrete dropout layer does not require the dropout probability as a hyperparameter. Rather, it optimises the dropout probability using normal gradient methods which are used to tune model parameters. In order to do this, firstly, an objective function is required. We can find a suitable objective function from the variational interpretation of dropout, where it is seen as an approximating distribution \( q_\phi(\theta) \) to the posterior distribution of model parameters \( \theta \) (recall equation (2.5)).

Let there be a Bayesian neural network with parameters \( \theta = \{ W_l \}_{l=1}^L \) where \( L \) is the number of layers in the neural network and \( \theta \) is the set of all weight matrices. Let \( \phi \) be the set of variational parameters (for the approximating distribution). Then the objective function can be written as follows:

\[
\hat{\mathcal{L}}_{MC}(\phi) = -\frac{1}{M} \sum_{i \in S} \log p(y_i | f^\theta(x_i)) + \frac{1}{N} \text{KL}(q_\phi(\theta) || p(\theta)) \tag{4.8}
\]

where \( \phi \) are the variational parameters to optimise, \( N \) is the number of data points, \( S \) is a random set of \( M \) data points, \( f^\theta(x_i) \) is the network’s output on the input \( x_i \) with weight matrices \( \theta \), \( p(y_i | f^\theta(x_i)) \) is the model’s likelihood and \( p(\theta) \) is the prior distribution over the network weights. The KL term in equation 4.8 is essentially a regularisation term which tries to reduce the deviation between the approximate posterior \( q_\phi(\theta) \) and the prior \( p(\theta) \).

Let us assume that the set of variational parameters \( \phi \) satisfies \( \phi = \{ M_l, p_l \}_{l=1}^L \), where \( M_l \) is the mean weight matrix of the \( l \)th layer and \( p_l \) is the corresponding dropout probability, such that \( q_\phi(\theta) = \prod_l q_{M_l}(W_l) \) and \( q_{M_l}(W_l) = M_l \cdot \text{diag}[\text{Bernoulli}(1 - p_l)^{K_l}] \) for a single random weight matrix \( W_l \) of dimensions \( K_{l+1} \) by \( K_l \). In that case, the KL term can be approximated following [29]:

\[
\text{KL}(q_\phi(\theta) || p(\theta)) = \sum_{l=1}^L \text{KL}(q_{M_l}(W_l) || p(W_l)) \tag{4.9}
\]
\[
\text{KL}(q_M(W) || p(W)) \propto \frac{I^2(1-p)}{2}||M||^2 - K \mathcal{H}(p) \tag{4.10}
\]

where \(K\) is the dimension of the weight matrix \(W\) and \(\mathcal{H}(p)\) is the entropy of a Bernoulli random variable with probability \(p\) given as:

\[
\mathcal{H}(p) = -p \log(p) - (1-p) \log(1-p). \tag{4.11}
\]

The above entropy term can be viewed as a term for dropout regularisation. It is clear that this term is not dependent on the model weights and hence can be ignored when the dropout probability is not optimised (i.e., in the case of fixed dropout probability). However, in this case (where we are explicitly optimising the dropout probability), this term cannot be ignored.

Now, given the objective function, we need to find its derivative with respect to the dropout probability \(p\). This is generally done using estimators like the score function estimator [46] or the pathwise derivative estimator [47]. The score function estimator has very high variance in practice which inhibits the optimisation process. The pathwise derivative estimator makes the assumption that the distribution can be re-parameterised to the form \(g(\theta, \epsilon)\) where \(\theta\) are the distribution’s parameters and \(\epsilon\) is a random variable which is independent of \(\theta\). This reparameterisation is not possible for Bernoulli distributions which we are working with. Hence, instead of using the vanilla dropout’s discrete Bernoulli distribution, we use the relaxed Concrete distribution [48] which is a continuous distribution (and can be used to approximate discrete random variables). Instead of sampling the random variable from a Bernoulli distribution which generates 0s and 1s, we sample from a Concrete distribution with some temperature \(t\) which results in values in the interval \([0, 1]\) but are very skewed towards the edges of the interval (i.e., values are very close to the boundaries 0 and 1). As has been shown in the paper [4], this relaxation allows us to use the pathwise derivative estimator to optimise the dropout probability.

In order to apply concrete dropout to the DeepLab-v3 architecture, we insert two concrete dropout layers in each Xception module (one between the first and second separable convolution layers and the other between the second and third). The reason is that the concrete dropout layers can theoretically tune the dropout probabilities and thus, we do not need to be concerned about the network parameters being over regularised (since technically, the dropout probabilities can be lowered down to 0). Furthermore, each concrete dropout layer returns a regularisation value which needs
Figure 4.4: Typical module of a Bayesian DeepLab variant with concrete dropout layers.

to be passed as the weight and bias regulariser for the next layer (i.e., the next convolution layer). Thus, we do not place a concrete dropout layer after the third separable convolution in an xception module to prevent passing of hyperparameters between consecutive xception modules. This leads to just a single probabilistic variant for the full-scale, medium scale and toy scaled models. The remaining parts of the architecture are exactly the same as discussed in the previous chapter. A single Xception module for the concrete dropout variant has been presented in Figure 4.4. The process of generating predictions and uncertainty values from the concrete dropout variant of Bayesian DeepLab is also the same as the MC dropout variant, i.e., the trained network is run multiple times with dropout for each test input and the mean and variance (predictive entropy and mutual information) of the multiple Monte-Carlo outputs are used as the prediction and uncertainty values respectively.

4.2.2 Deep Ensembles

In this section, we describe yet another inference technique which can be used to generate predictions and uncertainty values from deep neural networks. In the paper [5], the authors propose ensembles of networks as a method of generating uncertainty estimates. The motivation behind the work comes from the MC-dropout paper. We have mentioned before that dropout can be seen as “thinning” the neural network during training time by dropping the outputs of certain neurons. In conventional approaches, dropout is not applied during test time and hence, the prediction produced by the network is an average of the predictions produced by the multiple “thinned”
networks which were trained. In a way, we can consider the final network as an ensemble combination. Following from this idea, the deep ensemble paper [5] proposes training an ensemble of multiple deep networks and using the mean of the individual network outputs as the prediction and the variance measures as uncertainty estimates. It is worth noting here that there is just a single run of the ensemble for a given test input. We do not use multiple Monte-Carlo runs in this case.

There are two main points in the paper which we incorporate in our work. Firstly, the objective function which is optimised should not only maximise the neural network’s performance in prediction but should also result in well-calibrated uncertainty estimates. This can be ensured by using proper scoring rules which score each predictive distribution $p_\theta(y|x)$ ($\theta$ being the network parameters or weights, $y$ is the output and $x$ is the input) based on how close it is to the true distribution $q(y,x)$ on the $(y,x)$ tuples. A higher score indicates a better predictive distribution. Let $S(p_\theta, (y,x))$ denote the score of the distribution $p_\theta(y|x)$. The expected scoring rule can then be defined as:

$$S(p_\theta, q) = \int q(y,x)S(p_\theta, (y,x))dydx.$$ (4.12)

A scoring rule $S(p_\theta, (y,x))$ is proper if and only if $S(p_\theta, q) \leq S(q,q)$ with equality only when $p_\theta(y|x) = q(y|x)$, for all $p_\theta$ and $q$. We can optimise proper scoring rules in deep networks by setting the objective $L(\theta)$ to $-S(p_\theta, q)$. Fortunately, it turns out that most common loss functions which are well-known and heavily used are also proper scoring rules. For instance, in case of maximising the likelihood, the score function can be $S(p_\theta, (y,x)) = \log p_\theta(y|x)$. This leads to a proper scoring rule due to Gibbs’ inequality:

$$S(p_\theta, q) = \mathbb{E}_{q(x)} q(y|x) \log p_\theta(y|x) \leq \mathbb{E}_{q(x)} q(y|x) \log q(y|x).$$ (4.13)

In our case which is a multi-class classification problem, the softmax cross entropy loss objective which we use is equivalent to the log likelihood and hence is also a proper scoring rule.

The second point which we incorporate is using randomisation-based ensembles and not boosting-based ones. In randomisation based ensembles, each network can be trained in parallel and there is no interaction between individual networks. In boosting based approaches, networks are often placed in a sequential order and the output of one network is treated as the input to the next. We train two types of randomisation-based ensembles: the deterministic ensemble and the probabilistic ensemble. The
deterministic ensemble simply has five vanilla DeepLab-v3 networks (with no additional dropout layers in them) trained in parallel. The probabilistic ensemble has five Bayesian DeepLab networks with the Dropout-Middle-Flow variants. We observe the best performance from the Dropout-Middle-Flow variant and hence we train the probabilistic ensemble with this variant. Once trained, the outputs obtained from the individual networks in the ensemble are averaged to produce the prediction. Similarly, the variance measures (like predictive entropy and mutual information) are also computed on these outputs to generate uncertainty estimates. An illustrative figure on how predictions and uncertainties are obtained from an ensemble has been provided in Figure 4.5. In the next chapter, we discuss the experiments we undertake and the results obtained from the different inference techniques when evaluated with the metrics which we have studied.
Chapter 5
Experiments and Results

In this chapter, we present the results obtained by applying the evaluation metrics which we developed in the previous chapter on trained Bayesian DeepLab variants. We conduct tests to compare the three techniques of approximate inferencing in deep networks: MC-dropout [3], concrete dropout [4] and deep ensembles [5]. For all our tests, we use the Cityscapes [27] dataset which we describe in section 5.1. Next, we discuss the first experiment which involves performing sanity checks on the trained Bayesian models to ensure that the model behaviour is expected. Finally, in the second experiment we compare between the three inference techniques using the performance evaluation metrics: $P$(accurate|certain), $P$(uncertain|inaccurate) and $PAVPU$ Accuracy.

5.1 Cityscapes

Cityscapes is one of the most popular datasets for urban scene understanding. It has high-quality pixel annotations with a total of 5000 images collected from street scenes in 50 different cities. These 5000 images are split as: 2975 images in the training set, 500 images in the validation set and 1525 images in the test set. Among the classes, there are a total of 7 super-categories: flat (or ground), human, vehicle, construction, object, nature and sky. Each super-category has multiple classes in it and there are a total of 19 classes. The dimensions of each image are 1024 x 2048 (height x width).

As we have three scale-based variants of Bayesian DeepLab (i.e., full-scale, medium-scale and toy models), we train the different scales on modified versions of the dataset. The full-scale model is trained on the complete set of 2975 training images (each having dimensions 1024 x 2048). In the medium-scale architectures, the number of feature maps is reduced by a factor of 8 compared to their full-scale counterparts. Due to
this significant reduction in the number of parameters, we train the medium-scale
variants on a subset of the full training set (of 2975 images). In particular, we only
include the street scenes of three cities: Bochum, Jena and Ulm. This creates a re-
duced training set of 310 images. Correspondingly, the validation set has 59 images
(pictures from the city of Lindau) and the test set has 58 images (pictures from Lev-
erkusen). For the toy models, we use the same set of images as the medium-scale
models but we reduce the dimensions of the images by a factor of 16. Therefore, we
get images which are 64 x 128 (height x width). The reduction in image dimensions is
done using bilinear-interpolation. The corresponding reduction in label dimensions is
performed by dividing the entire 2d matrix of labels into disjoint 16x16 patches and
for each patch, choosing the majority label (label occurring the maximum number
of times in the patch). The full Cityscapes dataset has a size of approximately 11
GB, the medium scale dataset requires around 1 GB of space and the toy dataset has
only about 5.8 MB. Figure 5.1 contains a few images and corresponding labels of the
Cityscapes dataset.

Figure 5.1: Sample images and corresponding labels from the Cityscapes dataset
5.2 Experimental parameters and a brief analysis of the output

The input parameters to train a Bayesian DeepLab model include:

- a list of atrous rates for the ASPP (Atrous Spatial Pyramid Pooling: Refer to Section 3.1.2) module,
- the output stride which is the ratio of the spatial dimensions of the input to the output,
- a decoder output stride (ratio of spatial dimensions of the input to the decoder output) in case the model architecture uses a decoder,
- crop size for the training images (the images are centrally cropped based on the given crop size),
- a batch size for training, and
- the total number of training iterations.

For the full and medium-scale models, we use the same set of parameters as mentioned in the DeepLab-v3 [26] paper and set [6, 12, 18] as the list of atrous rates, 16 and 4 as the output and decoder output strides respectively, 768 x 768 as the image crop size, 16 as the batch size and train our model for 90000 iterations. The toy models do not have an ASPP module and do not use a decoder either. For these models, we set the output stride to 4, the crop size to 64 x 64, the batch size to 32 (as we are using smaller images) and the number of training iterations to 90000. For all the MC dropout variants, we use a dropout probability of 0.5. This applies to the probabilistic ensemble as well. For the concrete dropout variant, there are two hyperparameters: the weight regulariser which is set to $1e - 8$ and the dropout regulariser which we set to $1/(\text{num\_training\_images} \times \text{image\_dimensions})$. The number of Monte Carlo samples drawn during test time is 20 for both the MC dropout and Concrete dropout variants. Finally, each network is trained on a single NVIDIA Tesla P100-SXM2 GPU. The full-scale models require around 25 hours to train, the medium-scale models take approximately 8 hours and the toy models can be trained in about 30 minutes.

Once trained, we generate prediction and uncertainty maps on the test set and use these to compute the model performance metrics. Before we go to the metrics, a comment on the different types of uncertainties (i.e., mutual information and predictive
Figure 5.2: Images showing the ground-truth label, pixel-wise predictions and pixel-wise uncertainty maps for both predictive entropy and mutual information from a trained Bayesian DeepLab medium-scale model with MC dropout inference.

entropy) is in order. In Figure 5.2, we have provided the ground-truth labels, predictions and uncertainty maps (for both predictive entropy and mutual information) obtained from a trained medium-scale model with the Dropout-Middle-Flow variant (which uses MC dropout as the inference technique). In the uncertainty maps, a darker shade represents higher uncertainty and vice-versa. If we observe the mutual information map, we can see that the regions of the image which correspond to classes having low frequency (like pedestrian) have darker shades indicating higher uncertainty. On the other hand, in the predictive entropy map, we can notice dark shades near the borders of objects where the noise in the data is higher. This supports the explanation [29] that mutual information is a measure of model uncertainty (or epistemic uncertainty) which can be explained away with more data (i.e., if we keep on providing more and more data, the model will eventually become confident even on the classes which have low frequency). However, predictive entropy captures both epistemic and aleatoric uncertainties (together they are known as predictive uncertainty). Since the aleatoric uncertainty is due to noise in the data, it cannot be explained away with more data. Due to this property of predictive entropy (i.e., it captures the predictive uncertainty of the model), in our evaluation experiments, we use predictive entropy as the measure of uncertainty.
5.3 Experiment 1: Sanity Checks

In this section, we present the results of applying the sanity check tools: i) class-wise accuracy vs mean uncertainty, ii) class-wise frequency vs mean uncertainty and iii) pixel accuracy for the top x% most confident pixels on the trained Bayesian networks to ensure that the model behaviour is as expected. Figure 5.3 presents the results of the first two metrics applied on the full-scale Bayesian DeepLab variants (with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble). Firstly, we should note that in all the plots, with decrease in model confidence (or increase in uncertainty), the class accuracy also goes down. Furthermore, the classes which are less frequent have a higher level of uncertainty. This indicates a positive correlation between class accuracy and frequency which is expected. It is also apparent from the plots that some of the classes dominate others in terms of frequency. For instance, the classes Road, Vegetation and Building have the highest frequencies in the dataset and correspondingly have relatively low uncertainty values.

The plots obtained by applying the third metric (i.e., pixel accuracy for the top x% most confident pixels) on the trained Bayesian networks (containing the three scale-based variants) have been presented in Figure 5.4. Again as expected, all the curves show a downtrend which indicates that as we add pixels which the model is less confident about, the overall accuracy goes down. This is true for all kinds of uncertainties and hence, we obtain downtrends for both predictive entropy and mutual information. As we are presenting results from multiple inference techniques, it is important to mention here that the model which corresponds to the MC dropout inference is the Bayesian DeepLab Dropout-Middle-Flow variant. We observe that the middle flow variant consistently outperforms its peers (i.e., the other MC dropout variants like Dropout-Exit-Flow and Dropout-Entry-Flow) and hence we use this variant as the model of choice in our experiments. The results for justifying the performance of the Dropout-Middle-Flow variant have been reported in Appendix C.

5.4 Experiment 2: Performance Evaluation

In this section, we report the performance of the Bayesian DeepLab variants based on two classes of metrics:

- conventional evaluation metrics used in semantic segmentation like pixel accuracy, mean accuracy and mean I/U
• Patch based BDL evaluation metrics: $P(\text{accurate}|\text{certain})$, $P(\text{uncertain}|\text{inaccurate})$ and PAVPU Accuracy.

We want our model to have both a high accuracy in prediction and a strong inverse relation between accuracy and uncertainty. Hence, an ideal model should have high values in all of the above mentioned metrics. The results of the former set of metrics (i.e., pixel accuracy, mean accuracy and mean I/U) for the toy, medium and full-scale models have been reported in Table 5.1. The performance results obtained by using the BDL evaluation metrics have been presented in Table 5.2. It should be noted that the values of $P(\text{accurate}|\text{certain})$, $P(\text{uncertain}|\text{inaccurate})$ and the PAVPU Accuracy (columns 3 - 5) in Table 5.2 have been computed by taking the uncertainty threshold as the average uncertainty value over all pixels in the validation set. The plots of the PAVPU accuracy with varying thresholds of uncertainty are given in Figure 5.5. From these plots, we compute the peak PAVPU accuracy and report it in Table 5.2.

We can draw the following insights from the results:
1. Firstly, let us analyse what happens to the PAVPU Accuracy metric as we vary the uncertainty threshold. When the uncertainty threshold is 0, all the patches are classified as uncertain. Hence, the PAVPU Accuracy reduces to $\frac{n_{iu}}{n_{iu} + n_{au}}$, where $n_{iu}$ is the number of inaccurate and uncertain patches and $n_{au}$ is the number of accurate and uncertain patches. The denominator equals the total number of patches. Hence, the value of the metric becomes the fraction of inaccurate patches. Similarly, when the uncertainty threshold is 100% (i.e., the uncertainty threshold equals the maximum uncertainty value), all the patches are certain. In that case, the metric becomes the fraction of accurate patches. Hence, the sum of the PAVPU Accuracy values at thresholds 0 and 100% should be 1. This can be observed from the plots in Figure 5.5. Furthermore, we can also notice that the metric values obtained at uncertainty threshold of 100% are higher than the ones obtained at 0%. This means that the number of patches which are accurate is higher than the number of patches which are inaccurate. However, the curves are also bell-shaped. This indicates that the best uncertainty threshold to compute the PAVPU accuracy (given that we want to have high PAVPU accuracy) lies not at the edges of the 0-100 interval but somewhere in between.

2. It is worth noting that in the plots of Figure 5.5, the uncertainty values of the different inference techniques are not calibrated (i.e., they need not be in the same
range). Thus, it is better to consider the Peak PAVPU Accuracy for each inference technique separately rather than computing values like Area Under the Curve (AUC) which is typical in ROC curves. We report the peak PAVPU Accuracy (instead of the AUC) as a comparison metric. However, as an alternative, we could also have computed the PAVPU Accuracy for the top x\% most confident pixels and varied x from 10 - 100 (say). In this case, the uncertainty threshold should be fixed (at a value like the mean uncertainty over pixels in the test set). The values obtained from such a plot would be calibrated for every model and one could choose to compute the AUC of such a curve as a measure of model performance as well.

3. It should be noted that there is no significant difference between the PAVPU Accuracy value computed by taking the mean uncertainty over all pixels as the threshold and the peak PAVPU Accuracy value computed from the plots (i.e., the fifth and sixth columns in Table 5.2). In most cases, the latter is slightly higher in magnitude but the difference is quite insignificant given that the metric value has a relatively large range. This observation indicates that the mean uncertainty value is a decent threshold to compute the PAVPU Accuracy metric.

4. We observe that for the toy models, the order of performance is: \textit{Probabilistic Ensemble} > \textit{MC Dropout} \approx \textit{Concrete Dropout} > \textit{Deterministic Ensemble}. For the medium-scale models, the order is: \textit{Concrete Dropout} > \textit{Probabilistic Ensemble} > \textit{MC Dropout} > \textit{Deterministic Ensemble}. Finally, for the full-scale models, we have: \textit{Probabilistic Ensemble} \approx \textit{MC Dropout} > \textit{Deterministic Ensemble} > \textit{Concrete Dropout}. Here \textit{Model A} \succ \textit{Model B} means that model A performs better than model B based on the BDL evaluation metrics. \textit{Model A} \approx \textit{Model B} indicates that both models perform almost equally well. It is interesting to note that the there is a significant drop in the performance of the Concrete Dropout model in the full scale networks. A possible reason behind this drop can be that the model is heavily regularised as there are dropout layers in front of every separable convolution. Furthermore, as mentioned in the concrete dropout paper [4], larger models tend to push the dropout probability to 0.5, the highest value it can obtain. This regularisation may have caused a noticeable drop in the performance of the concrete dropout model. Therefore, a possible solution can be to reduce the number of concrete dropout layers as we increase the scale of the model.

5. Finally, we notice that the toy models perform quite poorly as compared to the medium-scale and full-scale models when evaluated with the conventional semantic
segmentation metrics (given in Table 5.1). However, they do not perform so poorly when evaluated with the BDL metrics (given in Table 5.2). The reason is that the BDL metrics do not evaluate a model based on how accurate its predictions are but by how certain it is on its accurate predictions and how uncertain it is on its inaccurate predictions. The values of the BDL metrics obtained for the toy models indicate that although they make bad predictions, they are still decent Bayesian models as they are quite uncertain on those bad predictions.

In a nutshell, we have laid out benchmarks comparing the inference techniques MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble using the BDL metrics which we proposed in Chapter 4. These benchmarks can in turn be used for comparing other BDL models. In the next chapter, we present our conclusions about the work and also discuss future research possibilities on the same.

<table>
<thead>
<tr>
<th>Scale of model</th>
<th>Approximate Inference Technique</th>
<th>Pixel Accuracy</th>
<th>Mean Accuracy</th>
<th>Mean I/U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy</td>
<td>MC Dropout</td>
<td>53.02</td>
<td>28.88</td>
<td>20.79</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td>53.63</td>
<td>29.43</td>
<td>21.07</td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>52.64</td>
<td>29.09</td>
<td>20.93</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td><strong>56.81</strong></td>
<td><strong>31.16</strong></td>
<td><strong>22.39</strong></td>
</tr>
<tr>
<td>Medium</td>
<td>MC Dropout</td>
<td>84.03</td>
<td>68.31</td>
<td>59.50</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td><strong>90.28</strong></td>
<td><strong>68.93</strong></td>
<td><strong>61.23</strong></td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>80.56</td>
<td>66.08</td>
<td>58.12</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td>87.96</td>
<td>69.19</td>
<td>60.64</td>
</tr>
<tr>
<td>Full</td>
<td>MC Dropout</td>
<td>94.60</td>
<td>82.25</td>
<td>75.63</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td>89.13</td>
<td>75.53</td>
<td>69.91</td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>94.39</td>
<td>81.06</td>
<td>75.95</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td><strong>95.44</strong></td>
<td><strong>86.05</strong></td>
<td><strong>77.05</strong></td>
</tr>
</tbody>
</table>

Table 5.1: Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with multiple inference techniques using conventional semantic segmentation metrics: pixel accuracy, mean accuracy and mean I/U (expressed as percentages).

The values shown in bold correspond to the maximum values of the metrics obtained. Each column has three values shown in bold, one for each of the scale based variants (full-scale, medium-scale and toy).
Figure 5.3: Plots of classwise accuracy vs uncertainty and classwise frequency vs uncertainty for full scale-models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques.
Figure 5.4: Plots of accuracy for the top x% most confident pixels (x varying from 5 to 100 in steps of 5) for the toy, medium-scale and full-scale models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques. There are two plots for each scale, one corresponding to predictive entropy and the other for mutual information.
Figure 5.5: Plots of the PAVPU accuracy for varying thresholds of predictive entropy. The figure includes plots for the toy, medium-scale and full-scale models with MC dropout, concrete dropout, probabilistic ensemble and deterministic ensemble inference techniques.
Table 5.2: Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with multiple inference techniques using BDL performance evaluation metrics: $P(\text{accurate}\mid\text{certain})$, $P(\text{uncertain}\mid\text{inaccurate})$ and PAVPU Accuracy.

The values shown in bold correspond to the maximum values of the metrics obtained. Each column has three values shown in bold, one for each of the scale based variants (full-scale, medium-scale and toy). The columns corresponding to the $P(\text{accurate}\mid\text{certain})$, $P(\text{uncertain}\mid\text{inaccurate})$ and PAVPU Accuracy contain values which have been computed by taking the uncertainty threshold as the mean uncertainty value of all pixels in the validation set. The values in the final column (i.e., Peak PAVPU Accuracy) have been obtained by looking at the plot of PAVPU Accuracy for different uncertainty thresholds. We choose the peak value that the PAVPU Accuracy attains in the plot.

<table>
<thead>
<tr>
<th>Scale of model</th>
<th>Approximate Inference Technique</th>
<th>$P(\text{accurate}\mid\text{certain})$</th>
<th>$P(\text{uncertain}\mid\text{inaccurate})$</th>
<th>PAVPU Accuracy</th>
<th>Peak PAVPU Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy</td>
<td>MC Dropout</td>
<td>0.8034</td>
<td>0.7526</td>
<td>0.6980</td>
<td>0.7044</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td>0.8044</td>
<td>0.7568</td>
<td>0.6938</td>
<td>0.7014</td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>0.7956</td>
<td>0.7686</td>
<td>0.6917</td>
<td>0.6966</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td>0.8225</td>
<td>0.8081</td>
<td>0.7041</td>
<td>0.7103</td>
</tr>
<tr>
<td>Medium</td>
<td>MC Dropout</td>
<td>0.8244</td>
<td>0.7968</td>
<td>0.6907</td>
<td>0.6966</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td>0.8267</td>
<td>0.8097</td>
<td>0.6961</td>
<td>0.7129</td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>0.8074</td>
<td>0.8069</td>
<td>0.6993</td>
<td>0.6966</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td>0.8526</td>
<td>0.8035</td>
<td>0.6871</td>
<td>0.7008</td>
</tr>
<tr>
<td>Full</td>
<td>MC Dropout</td>
<td>0.8212</td>
<td>0.8151</td>
<td>0.6902</td>
<td>0.7142</td>
</tr>
<tr>
<td></td>
<td>Concrete Dropout</td>
<td>0.8287</td>
<td>0.8194</td>
<td>0.6983</td>
<td>0.7155</td>
</tr>
<tr>
<td></td>
<td>Deterministic Ensemble</td>
<td>0.8092</td>
<td>0.7932</td>
<td>0.6985</td>
<td>0.7121</td>
</tr>
<tr>
<td></td>
<td>Probabilistic Ensemble</td>
<td>0.8419</td>
<td>0.8015</td>
<td>0.7012</td>
<td>0.7038</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

In this chapter, we summarize the contribution of the work and analyse how we did given the project goals. We also propose possible extensions of the work which can be considered for future research.

6.1 Summary and Contributions

In Chapters 3 and 4, we modified the well-known DeepLab-v3 [26] architecture in order to enable the network to produce both dense predictions and pixel-wise uncertainty estimates. We call the modified architecture *Bayesian DeepLab*. We incorporated three approximate inference techniques into the network: i) MC dropout [3], ii) concrete dropout [4] and iii) deep ensembles (both probabilistic and deterministic ensembles) [5]. Furthermore, we have also presented five probabilistic variants of Bayesian DeepLab with MC dropout depending on where in the network we place the dropout layers (i.e., Dropout-Entry-Flow, Dropout-Exit-Flow, Dropout-Middle-Flow, Dropout-Entry-Exit-Flow and Dropout-Entry-Middle-Exit-Flow).

In Chapter 4, we have studied three tools (class-wise accuracy vs mean uncertainty, class-wise frequency vs mean uncertainty and plots of accuracy for the top x% most confident pixels) which enable users to perform sanity checks on trained Bayesian networks ensuring that there exists an inverse relation between model accuracy and uncertainty. Furthermore, we have also utilised the intuition of this inverse relation to propose three metrics ($P$(accurate|certain), $P$(uncertain|inaccurate) and PAVPU Accuracy) which can be used to quantify the performance of a Bayesian model in terms of its accuracy and uncertainty.

Finally, in our experiments, we have applied the above tools and metrics on Bayesian
DeepLab variants with the approximate inference techniques mentioned above and presented the results obtained in Chapter 5. We observe that our models perform quite well in terms of the conventional semantic segmentation metrics (with the best performing model achieving a mean I/U of 77.05% as compared to 77.23% which was reported in the DeepLab-v3 paper [26] on the Cityscapes dataset). Furthermore, it is interesting to observe that the models which perform the best in terms of the BDL evaluation metrics also tend to have higher accuracies. In Chapter 5, we present benchmarks which researchers can use to evaluate and compare other BDL models. Thus, we have successfully achieved the project goals which we listed in Chapter 1.

6.2 Future Research

The field of Bayesian Deep Learning is gaining more and more popularity with the development of scalable and efficient ways of generating uncertainty measures from deep neural networks. With this work, we expect that researchers will have an easier time evaluating and comparing their models with the state-of-the-art. Having said that, there are multiple possible extensions of this work which can be carried out in future research endeavours. Some of them are as follows:

1. **Metrics based on downstream tasks:** In this work, we have used the Cityscapes dataset for all our experiments. We have evaluated models based on their performance in segmenting urban street scenes which can be applied to autonomous vehicles. Most of the decisions in these vehicles are made at a level higher than just the task of semantic segmentation. Hence, we can define metrics based on these decisions and evaluate the BDL models on the correctness of the decisions made by the vehicle. A decision can be defined as a simple binary event. For instance, “Should the car stop?” can be a binary event representing a decision. The car should stop if it sees a pedestrian in front. Based on the dense predictions and uncertainty estimates obtained from the semantic segmentation module, we can say if there is a pedestrian in front of the car. Finally, we can compute a confusion matrix representing the true positives, true negatives, false positives and false negatives (based on where the car should have stopped and where it actually stopped). Furthermore, we can draw an ROC curve from the matrix by varying the threshold of uncertainty where we decide if a pedestrian is in front of the car or not. Since this is just for a single binary event of “Should the car stop?”, we can define several other binary events and come up with corresponding ROC curves to judge the performance of an autonomous vehicle. It is worth noting here that these metrics are completely specific to not just semantic
segmentation but urban street scene understanding and autonomous driving. The metrics which we have developed in this thesis are more general as in they are applicable in any semantic segmentation scenario.

2. **Drawback of the PAVPU Accuracy metric:** Although the PAVPU metric computes the fraction of good case patches i.e., patches which are (accurate, certain) or (inaccurate, uncertain) compared with the total number of patches, there is a scenario where the PAVPU Accuracy will provide a high value even when the model is not a good one. If all the patches are uncertain and inaccurate, the value of the PAVPU Accuracy is 1. Although it is desired that the model should be uncertain about inaccurate patches, we do not want all the patches to be inaccurate in the first place. Modifications to correct this drawback can include having just the number of accurate and certain patches in the numerator (see equation 4.7) or to multiply the fraction of accurate patches to the PAVPU Accuracy value.

3. **Enriching the benchmarks:** Finally, in this work we have considered the approximate inference techniques which have been motivated by the MC dropout paper [3]. In the recent past, there have been a few other scalable ways of approximate inferencing in deep networks like Bayes by Backprop [22], Deep Kernel Learning [23] and Scalable Bayesian Optimisation [49]. The set of benchmarks which we have laid out can be enriched with these inference techniques as well.

In conclusion, we hope that our work will provide an easy and effective way for researchers to evaluate and compare deep Bayesian models.
Appendix A

Using the BDL Benchmark Tool: Simple API Tutorial

The code for this project has two primary sections: i) the BDL benchmark tool which is responsible for evaluating a given BDL model on the Cityscapes dataset and ii) the semantic segmentation code which contains training code for the different variants of Bayesian DeepLab as well as test code which uses the BDL benchmark tool to evaluate the trained models. The BDL benchmark tool is agnostic to deep learning frameworks and can be used with multiple frameworks like Tensorflow and PyTorch. An illustrative figure explaining how the code works has been provided in Figure A.1. The numbers on the arrows indicate the steps involved in running the entire evaluation system. The steps are as follows:

1. *Train model:* The BDL model (i.e., deep neural network with an inference technique that can generate both predictions and uncertainty values) is trained on the Cityscapes data (or other data if the user wishes to see performance on out-of-distribution points). The trained model is saved as a checkpoint.

2. *Load trained model:* The trained model is loaded into memory by the test module and the model weights are restored from the checkpoint.

3. *Call BDL Benchmark APIs:* The test module calls APIs exposed by the BDL Benchmark tool specifying the metrics which need to be computed.

4. *Load Cityscapes Data:* The BDL Benchmark tool starts by creating an iterator over the Cityscapes data. Since the model has already been trained this iterator is over the test data. However, the iterator can also be created over the training set or the validation set if the user chooses.
5. **Evaluate model**: The benchmark tool uses the metrics module to evaluate the trained model using the specified metrics. The results are collected in a dictionary.

6. **Report results to caller**: The benchmark tool reports the results back to the caller (the test module in this case).

7. **Create tables and plots**: Using the dictionary of computed metrics, the test module calls methods required to plot graphs and create tables based on how the results need to be reported.

8. **View results**: The result presentation module plots the required metrics and they are finally presented. The plotted graphs can be used to compare different BDL models.

In the next section, we describe the APIs exposed by the BDL benchmark tool for the convenience of the reader.

### A.1 BDL Benchmark Tool

The main purpose of the BDL benchmark tool is to expose framework-agnostic APIs which can help users evaluate their own BDL models designed for semantic segmentation. The APIs are defined in the tool have been described below.
A.1.1 Iterator APIs

There are two APIs `iterate_train()` and `iterate_validation()` which return normal python iterators over the training and validation sets of the Cityscapes dataset. The path of the Cityscapes data is taken as input. We have intentionally not defined any `iterate_test()` API because we do not want any external user to have direct access to the test set. The only access to the test set should be inside the benchmark tool itself. The code for creating an iterator over the Cityscapes dataset is present in the `data_iterator.py` file which contains the method `get_data_iterator()`. The method takes the split (training/test/validation) of the dataset to create the iterator over and the path of the Cityscapes data. The code for the iterator APIs is given below.

```python
def iterate_train(self):
    return get_data_iterator(split='train', path=self.data_path)

def iterate_validation(self):
    return get_data_iterator(split='val', path=self.data_path)
```

A.1.2 Metrics APIs

The metrics APIs are used to get the values of different evaluation metrics for BDL models. There are two APIs: i) the `metrics_deterministic()` API and the `metrics_stochastic()` API. The first one deals with all the metrics which can be computed from just the ground-truth labels and the corresponding predicted labels. These metrics are not concerned with the uncertainty values. To be precise, these are the three semantic segmentation metrics, pixel accuracy, mean accuracy and mean I/U defined in Chapter 4. The second API deals with the second class of metrics which use labels, predictions and uncertainty values as inputs. These metrics are all the BDL metrics which we discussed in Chapter 4. The implementations of the metrics are in the `deterministic_metrics.py` and `stochastic_metrics.py` files (inside the `/BDL_benchmarks/segmentation/report` folder). We expose the metric computation APIs to enable users to separately compute certain metrics as part of the training/cross-validation process. The code for the `metrics_deterministic` API is presented below for the reader to understand the input and output for the metrics APIs.
The input/output type for the \textit{metrics\_stochastic} API is almost the same, the only difference being that the latter takes the uncertainty values as an input parameter as well.

\begin{verbatim}
METRIC_FUNCTION_MAP = {
    segmentation_metrics.PIXEL_ACCURACY: evaluate.pixel_accuracy,
    segmentation_metrics.MEAN_ACCURACY: evaluate.mean_accuracy,
    segmentation_metrics.MEAN_IOU: evaluate.mean_iou,
    segmentation_metrics.CLASS_ACCURACY_MEAN_UNCERTAINTY: evaluate.classwise_accuracy_frequency_uncertainty,
    segmentation_metrics.CLASS_FREQUENCY_MEAN_UNCERTAINTY: evaluate.classwise_accuracy_frequency_uncertainty,
    segmentation_metrics.ACCURACY_VS_THRESHOLDED_UNCERTAINTY: evaluate.accuracy_for_threshold_uncertainty,
    segmentation_metrics.PATCH_BASED_METRICS: evaluate.patch_based_metrics,
    segmentation_metrics.ACCURACY_VS_THRESHOLDED_UNCERTAINTY_PLOT: evaluate.accuracy_for_threshold_uncertainty_plot,
    segmentation_metrics.PATCH_BASED_METRICS_PLOT: evaluate.patch_based_metrics_multiple_params
}

@staticmethod
def metrics_deterministic(labels, predictions, metric=None):
    if (metric is None):
        raise ValueError('Metric name cannot be None')

    return METRIC_FUNCTION_MAP[metric](labels, predictions)
\end{verbatim}

\subsection{A.1.3 The \textit{generate\_report}() API}

The most important API of the BDL benchmark tool is the \textit{generate\_report()} API. It lets users evaluate their own BDL models with specified metrics. Before invoking the API, a BDL model must first be defined and trained. Next, the user has to define three functions:
1. **The Load Function:** The *load function* is used to load the trained model into memory. The loading process can be performed from a checkpoint (in case of big models) or it can even be an empty function for smaller models which do not need to be stored separately into a checkpoint (i.e., for models which have very low training times).

2. **The Estimator Function:** The *estimator function* is called by the *generate_report()* API in order to get predictions and uncertainty values from the loaded model. The estimator is provided a single image as input and the expected result is a tuple containing the predictions from the model as the first element and the uncertainty estimates as the second element. It is worth noting that the process of generating predictions and uncertainties may be different for different BDL models and inference techniques. For instance, in the case of MC-dropout the process involves multiple Monte Carlo runs of the trained model on the same input. However, for deep ensembles, the process is to get the output from each trained network in the ensemble and then compute their average and variance measures.

3. **The Destroy Function:** Once the process of getting predictions and uncertainty estimates is complete, it is good practice to remove the model from memory to prevent unnecessary consumption of memory space by the model which is no longer required. In order to do this, the user is required to define a *destroy function* which clears the memory of the loaded model.

In addition, the *generate_report* API takes a few other parameters like a list of metrics which have to be computed for the model, whether the outputs from the model (i.e., the predictions and uncertainty values) should be saved as images or not etc. The API returns a dictionary containing the values of all the metrics which have been computed on the model. It should be noted here that with the way the API is defined, the users are required to follow a certain pattern of coding in their models. However, in return, the APIs remain completely agnostic of any kind of deep learning framework (like tensorflow or pyTorch). This feature increases the applicability of these APIs. The code for the *generate_report* API is given below.

```python
def generate_report(self, estimator, load_function=_default_load_function,}
```
load_function_params = {},
destroy_function = _default_destroy_function,
destroy_function_params = {},
metrics = [],
additional_params = {},
dataset_split = 'test',
save_model_outputs = False,
save_path = None):

    # Check if estimator is None or not
    if estimator is None:
        raise ValueError("estimator() function cannot be None.")

    # Get iterator on specified split of data
    data_iterator = get_data_iterator(split=dataset_split,
                                       path=self.data_path)

    # Call the loader function to load the model
    load_function_output = load_function(
                                      load_function_params)

    # Iterate over dataset and call estimator to get predictions and uncertainties
    labels = []
predictions = []
uncertainty_values = []
len_iter = data_iterator.length()
counter = 1
while (True):
    try:
        (image, label) = data_iterator.next()
        image = np.expand_dims(image, axis=0)
        prediction, uncertainties = estimator(image,
                                      load_function_output=load_function_output)
        prediction = np.squeeze(np.argmax(prediction,
                                       axis=3))
        predictions.append(prediction)
        uncertainty_values.append(uncertainties)
        labels.append(label)
sys.stdout.write("\r Processed %d/%d images ...
..." % (counter, len_iter))
sys.stdout.flush()
counter += 1
except StopIteration:
    break

print()
if save_model_outputs:
    self._save_model_outputs(save_path, labels, predictions, uncertainty_values)

# Get the required metrics
print('Computing evaluation metrics: ...')
res = self._get_evaluation_metrics(labels, predictions, uncertainty_values, metrics, additional_params)
print('Evaluation metrics computed!')

# Remove model from memory
destroy_function_params[LOAD_FUNCTION_OUTPUT] = load_function_output
destroy_function(destroy_function_params)

return res
Appendix B

Medium-scale Bayesian DeepLab variants

In the table on the next page, we detail out the model architecture of the medium-scaled Bayesian DeepLab variants which use the MC dropout inference technique.
Table B.1: Xception Module structures of the medium-scaled probabilistic model variants of Bayesian DeepLab

The columns of the table represent the probabilistic variants of the medium-scaled Bayesian DeepLab architecture. The rows correspond to the different Xception modules in the Bayesian DeepLab network in sequence. In each cell, we provide the architectural parameters of the Xception module in the corresponding variant. The \textit{Channels} parameter is a list providing the depth (i.e., the number of channels) in each of the three separable convolution layers in the module. The \textit{Skip} parameter represents the type of skip connection in the module. The \textit{Conv} skip connection means that there is a 1x1 filter in the skip connection having the same depth as the last separable convolution layer. The \textit{Sum} skip connection means that there is no additional operation in the skip connection and it just contains the input to the module. The \textit{None} value means that there is no skip connection. The \textit{Dropout} parameter is a boolean value representing if there is a dropout layer at the end of the module (i.e., after the addition of the last separable convolution output and the skip connection). Finally, the \textit{Stride} parameter indicates the stride of the last separable convolution layer as well as the 1x1 filter in the skip connection if any. A stride of 2 will reduce the spatial dimensions of the input by half. A stride of 1 keeps the dimensions unchanged.
Appendix C

Results of the Bayesian DeepLab variants using MC dropout inference

This appendix is a continuation of the Experiments chapter (Chapter 5) and contains results obtained by comparing the different Bayesian DeepLab variants using MC dropout (i.e., the Dropout-Entry-Flow, Dropout-Exit-Flow, Dropout-Middle-Flow, Dropout-Entry-Exit-Flow, and Dropout-Entry-Middle-Exit-Flow variants). Firstly, Figure C.1 presents the sanity check results on the variants where the models are expected to have decreasing accuracy values when tested on an increasing number of uncertain pixels. Next, Table C.1 gives the results of evaluating the predictions of the models using conventional semantic segmentation metrics like pixel accuracy, mean accuracy and mean I/U. Finally, in Table C.2 and Figure C.2, we present the results of evaluating the variants using the performance evaluation metrics, $P(\text{accurate}|\text{certain})$, $P(\text{uncertain}|\text{inaccurate})$ and PAVPU Accuracy. The presentation of the results here is similar to the ones given in Chapter 5. The primary purpose of the appendix is to justify the use of the Dropout-Middle-Flow variant as the representative for the MC dropout inference technique. Clearly, the results indicate that the Dropout-Middle-Flow variant consistently outperforms its peers on all of the metrics for the different scales of Bayesian DeepLab networks.
Figure C.1: Plots of accuracy for the top x% most confident pixels (x varying from 5 to 100 in steps of 5) for the toy, medium-scale and full-scale models using the various Bayesian DeepLab MC dropout variants. There are two plots for each scale, one corresponding to predictive entropy and the other for mutual information.
Figure C.2: Plots of the PAVPU accuracy for varying thresholds of predictive entropy. The figure includes plots for the toy, medium-scale and full-scale models for different Bayesian DeepLab MC dropout variants.
Table C.1: Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with MC dropout inferencing using conventional semantic segmentation metrics: pixel accuracy, mean accuracy and mean I/U (expressed as percentages).

The values shown in bold correspond to the maximum values of the metrics obtained. Each column has three values shown in bold, one for each of the scale based variants (full-scale, medium-scale and toy).

Table C.2: Evaluation results obtained from Bayesian DeepLab variants (full-scale, medium-scale and toy models) with MC dropout inferencing using BDL performance evaluation metrics: $P(\text{accurate}|\text{certain})$, $P(\text{uncertain}|\text{inaccurate})$ and PAVPU Accuracy.

The values shown in bold correspond to the maximum values of the metrics obtained. Each column has three values shown in bold, one for each of the scale based variants (full-scale, medium-scale and toy). The columns corresponding to the $P(\text{accurate}|\text{certain})$, $P(\text{uncertain}|\text{inaccurate})$ and PAVPU Accuracy contain values which have been computed by taking the uncertainty threshold as the mean uncertainty value of all pixels in the validation set. The values in the final column (i.e., Peak PAVPU Accuracy) have been obtained by looking at the plot of PAVPU Accuracy for different uncertainty thresholds. We choose the peak value that the PAVPU Accuracy attains in the plot.
Bibliography


