Towards Probabilistic Reasoning for Autonomous Vehicles

A

Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment

of the requirements for the degree

Master of Science

by

Ryan Nicholas McCampbell

August 2020

APPROVAL SHEET

This

Thesis

is submitted in partial fulfillment of the requirements
for the degree of

Master of Science

Author: Ryan Nicholas McCampbell

This Thesis has been read and approved by the examing committee:

Advisor:	Madhur Behl
Advisor:	
Committee Member:	Yanjun Qi
Committee Member:	Vicente Ordóñez Román
Committee Member:	
Committee Member:	
Committee Member:	
Committee Member:	

Accepted for the School of Engineering and Applied Science:

[illegible]

Craig H. Benson, School of Engineering and Applied Science

August 2020

Acknowledgements

I would like to thank my advisor Prof. Madhur Behl for his endless patience and gentle nudging during these long months of quarantine while I slagged along listlessly, hoping that something would eventually come of this work. Without his guidance I don't think I ever would have finished my Masters degree.

I would also like to thank my family, for keeping me company and providing me a place to live and food to eat all this time. If not for you I would be lonely, starving and homeless.

Abstract

Autonomous cyber-physical systems such as self-driving cars are increasingly becoming dependent on AI enabled methods for their perception, planning, and control tasks. Unfortunately, deep learning algorithms have been proven to be unreliable in presence of incomplete, imprecise, or contradictory data and adversarial attacks that exploit critical design flaws leading to untrustworthy results. Managing uncertainty is possibly the most important step towards safe autonomous systems. Modeling an autonomous vehicle's unfamiliarity for a given dynamic scenario enables appropriate subsequent decisions to be made under such uncertainties.

We propose to develop a framework to characterize and quantify the uncertainty in the perception stage of an autonomous vehicle's computation loop. Using Bayesian learning, we can quantify the confidence the AV has in its scene understanding outputs. Using this framework, we can also detect when the autonomous vehicle is operating outside of its operational design domain (ODD). Mistakes by lower-level AI components can propagate up the decision-making process and lead to devastating results. In such modular autonomous systems, we can use probabilistic reasoning in low-level components and make safe, and reliable high-level decisions given this uncertainty information.

In this thesis, we first provide motivation for the use of Bayesian methods in autonomous vehicles, followed by some background on Bayesian networks in machine learning. Then we explore two case studies. The first is a simplified look at the capabilities of Bayesian neural networks on the basic MNIST image detection dataset. Then we explore a problem more relevant for autonomous vehicles, semantic segmentation of driving scenes, and examine the benefits of Bayesian neural networks for this task. We find that Bayesian neural networks can provide more reliable measures of confidence than standard softmax outputs and can enable us to detect inputs that fall outside of our training domain.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade much progress has been made in the use of artificial intelligence to automate tasks that previously could only be performed by humans. This has many benefits: not only can we improve efficiency and reduce human burden in some tasks by enabling computers to supplement the work of humans, but we can also use autonomous systems to perform tasks which are too difficult or dangerous for some humans to perform. Driving is one function that could result in major improvements for humanity if it can be automated on a large scale: improving traffic flow, reducing accidents, affording people who are disabled or otherwise unable to drive better mobility and freedom, and simply increasing convenience for people who have long commutes and don't want to spend large amounts of time driving.

But driving is a difficult and complex task. Even for humans, we have state-mandated driving training courses and minimum hours of learning required before we are allowed to drive unsupervised. There are many concurrent sub-tasks which a human or computer must master in order to drive effectively. First, there is perception: the environment surrounding a car must be thoroughly scanned and all obstacles or items of interest identified, including the road and its markings, other cars on the road, road signs and signals, pedestrians walking along or crossing the road, and bicyclists and other atypical objects on the road. These must not only be identified but localized in the three-dimensional environment surrounding the car. Then a plan must be generated, taking into account all objects in the surroundings and the desired destination to generate a driving route. Finally the vehicle must be controlled to implement this plan, which requires precise feedback and understanding of physics to ensure the car does not deviate from its path. All of these tasks must be performed concurrently in real time to ensure safe and correct driving. There are many challenges to implementing each of these tasks in a self-driving car, but most importantly, each one is safety-critical, meaning any error could potentially result in human injury or loss of life. Therefore an autonomous vehicle system must be particularly cautious and aware of its own limitations, and never take an action that cannot be guaranteed to be safe with a high degree of confidence. The perception or scene understanding stack is particularly important because if the vehicle fails to properly identify obstacles in its way, then the planning and control stages have no hope of avoiding them, so this component must be scrutinized more than any other.

Deep learning has made impressive strides in the field of computer vision, particularly with convolutional neural networks (CNNs). These networks are incredibly powerful and flexible for classifying and detecting objects in images, which makes them the preferred choice for implementing perception stacks in self-driving cars. Tasks performed by neural networks can include object detection (both for obstacles like cars in the road and for signs/signals), lane detection, drivable surface detection, and semantic segmentation - which is like object detection but on a per-pixel level, giving an exact shape for each object. Several existing open source self driving car platforms use neural networks for some or most of their perception algorithms, some examples being Baidu's Apollo [1] and Comma.ai's openpilot [8], which both use custom neural networks for perception, and Autoware.AI [19], which includes state-of-the-art YOLO and SSD models for object detection.

1.1 Limitations of Deterministic CNNs

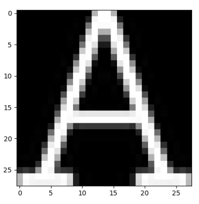
The downside to deep neural networks is that they are mostly black boxes when it comes to analysis. It is very difficult to gain intuitive understanding or make predictions about the behavior of a neural network, let alone provide strict guarantees. A neural network may provide the correct classification 99.9% of the time for pictures of cars, cats and dogs or whatever you prefer, but there are always some edge cases where it fails to predict correctly.

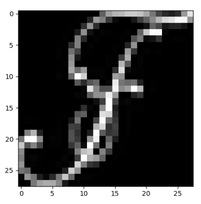
This comes in part from the limitations of the supervised training process: the model has not seen every possible scenario during its training stage, so we cannot know how it will act when it sees something it has limited experience with. There is also simply error inherent in the fact that the real world cannot be perfectly divided into a neat set of finite objects and classes everywhere. What should the network report if the lane markings are faded, or the weather is obscuring part of its vision, or there are objects that don't quite fit into the category of "car" or "truck"? These are severe problems when we are implementing something that requires strong safety guarantees.

What we ultimately would like is some method of determining how reliable our model's predictions about the environment are. We need a model that can output not only a function of its input but also a measure of confidence or uncertainty in its results.

But don't neural networks give us a confidence measure? Certainly, most neural networks produce their final results via a sigmoid or softmax function (for binary or multi-class classification). These functions assign a value to each result between zero and one, such that all classes' scores sum to one. So these outputs can indeed be intuitively considered as probabilities for each class being the correct class, and we can say that our uncertainty is simply how far away our chosen class score is from 1.

Unfortunately, it is not that simple. An artificial neural network is designed to learn an arbitrary function from input vectors to output vectors, and the softmax function's purpose is merely to "squash" the output vector into the range of zero to one so that it can be compared to a one-hot ground truth vector. This means that for examples within its training domain a neural network should learn to output scores close to zero or one, but for outliers there is little guarantee about what the network will output, and in fact it may simply learn to output highly "confident" scores for *any* input example (this is an example of overfitting). As a brief example of a case where softmax outputs cannot be treated rigorously, here are some example outputs of an MNIST [26] digit classifier on inputs that don't fit into any of its defined categories:


(a) Label: 4, 100% conf


(b) Label: 8, 84.5% conf

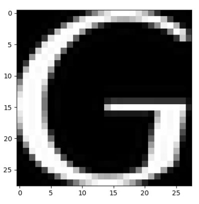

(c) Label: 3, 94.5% conf

Figure 1.1: Examples of MNIST classifier predictions on English letters

As can be seen, although none of the examples are actually digits, the model outputs a high confidence score with its classification, even as high as 100% for the letter A as 4. We will revisit this later in chapter 5.

While misclassifying a handwritten digit is not a safety-critical failure, the CNNs used for scene understanding and perception for self driving cars rely on similar neural network architectures and training practices and therefore could suffer form similar shortcomings. Consequentially, we would like to include a more rigorous measure of confidence or uncertainty in our model results, which is the focus of this work.

1.2 Probabilistic Reasoning in Neural Networks

We ideally would like to be able to generate a *probability distribution* of all possible outputs from a model. This means that we can generate both a mean and standard deviation for numerical results of a model, and we can hopefully get a more reliable measure of how likely a given result is correct. This not only gives us our confidence score but lets us see what other results had high probability, which is useful for tasks like image classification/segmentation where we might want to know all of the likely candidates besides the final classification.

Since a distribution can be an arbitrary abstract function, in practice we can generate a sample of a large number of stochastic model outputs as our distribution, and derive numerical statistics from this sample. This means there must be some randomness built into the model. This randomness could be provided in a static manner via random perturbations to the network and/or inputs, multiple independently trained networks (an ensemble), or mechanisms like dropout. A more dynamic and comprehensive approach is to learn random distributions for model parameters, which yields a Bayesian neural network (BNN). This means for example each weight of a neural network layer could be represented as a normal distribution $\mathcal{N}(\mu, \sigma)$ with μ and σ learned independently. More complex multivariate distributions could also be used to capture interdependence between weights. In this work we explore the implications of Bayesian neural networks for autonomous driving tasks.

1.3 Challenges for Bayesian Neural Networks

The curse of dimensionality: There are many challenges to implementing and using Bayesian neural networks, especially for real-time applications like autonomous driving. One of the major problems is the curse of dimensionality - that is, the fact that models become increasingly difficult to train as the number of model parameters increase and become subject to overfitting. Making neural network weights Bayesian by adding sigma terms to each parameter greatly increases the network size, making it both more space- and time-consuming and more difficult to converge to an optimal solution. One way to address this problem is by only making parts of the network Bayesian, and keeping the rest of the network deterministic - which can then be thought of as only a ”feature extractor” followed by a fully Bayesian network.

Computation time: Another challenge is the computation time needed for inference in Bayesian neural networks. In order to obtain a reasonable estimate of the true distribution of the result, we have to sample at least in the tens to hundreds of model passes for a single input. Each pass includes sampling the distributions of all model weights followed by a forward pass through the network. Minibatching can allow reuse of a single model sample for many forward passes, but this might not be viable for real-time applications. Thus the inference time can be increased by hundreds. The inference time can be improved again by keeping the initial layers of the network deterministic, thereby requiring only one pass for this part of the network followed by several shorter passes. A way to address this further is to use a kind of pruning: if we can determine which model weights have lower variance or contribute less to the overall randomness, we can replace them with deterministic weights and then avoid having to sample them each time. This doesn't speed up the forward data pass however.

Interpreting uncertainty: A final more abstract problem is how to interpret and use the Bayesian model outputs to make better AI's for driving and other tasks. This requires defining various metrics and heuristics to decide how to act on probabilistic results. There are multiple ways to interpret the "confidence" probabilistic results, in terms of output variance as well as average scores across multiple passes. At some point it is necessary to define arbitrary thresholds and rules for when to consider the uncertainty too high and thus requiring some extraneous behavior or human intervention.

1.4 Probabilistic Reasoning for Self-Driving Cars

For our specific case of self-driving cars, there are many tasks that Bayesian networks could be applied to. For lane detection we could compute many different approximate paths for each detected lane marker and use these to determine how safe a particular trajectory is. For object detection we could compute many bounding boxes for an object to see how confident in its position or distance we are, or for objects that only appear spuriously, how likely they are actually there in the first place.

Probabilistic scene understanding: Semantic segmentation is like object detection but can provide more fine-grained pixel-level information so is easily adaptable to many tasks. The goal is to label every pixel in an input image with a corresponding class label. An example of the semantic segmentation task is shown in Figure 1.2

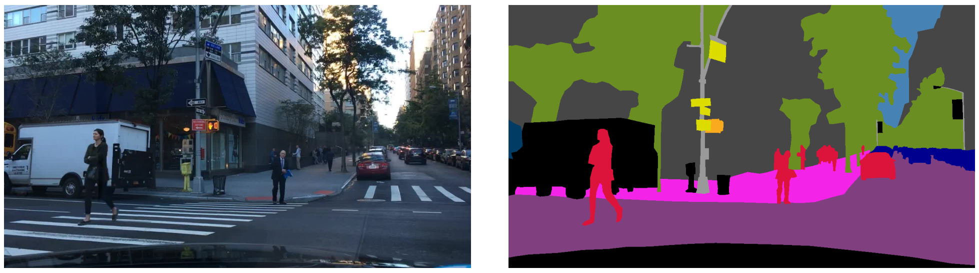


Figure 1.2: Example input and labels for semantic segmentation

This task is easily adaptable to Bayesian neural networks because the outputs are simple numerical results per-pixel, as opposed to models outputting discrete geometric features which require confidence thresholds and duplicate suppression. Moreover this is an essential component of many autonomous perception stacks, and other tasks can build off of it by using pixel clustering algorithms or other techniques to detect objects rather than using a whole separate model. We therefore focus on semantic segmentation in this work.

Operational Design Domain violations: An independent use for Bayesian neural networks in autonomous vehicles regardless of their primary task is detection of Operational Design Domain (ODD) violations. The ODD defines the set of operating conditions that an autonomous vehicle is designed to safely operate in [32]. We can assume that the set of conditions seen during training for a neural network constitutes part of the ODD; therefore when the model outputs high variance results for a large part of the perception input image this is a good signal that the ODD is being violated as the car is encountering a scenario it has not been thoroughly trained on. This can be a useful signal to tell a driver to take control of the vehicle or stop the vehicle altogether.

1.5 Aleatoric vs Epistemic Uncertainty

One final thing that is useful to note is that uncertainty is not uniform; it can be divided into different types. *Epistemic uncertainty*, also called *model uncertainty*, is the uncertainty that results from a lack of knowledge, i.e., lack of training data in parts of the input domain. While neural networks perform very good at interpolation tasks, it is adequate to say that they cannot extrapolate well. In other words, neural networks can only deal with things similar to what they have seen before.

On the other hand, there is also uncertainty due to potential intrinsic randomness of the real data generating process. This is called *aleatoric uncertainty.* It includes things like measurement and process noise. Aleatoric uncertainty is mostly independent of the model, meaning a different model or even a human wouldn't necessarily perform better, because the model is simply not given enough information to produce a reliable response.

We can further distinguish between *homoscedastic* and *heteroscedastic* aleatoric noise. Noise is called homoscedastic if it follows the same distribution indifferent of the input values. Heteroscedastic noise, on the other hand, depends on the input and can, thus, change in variance or even distribution across the input domain.

1.6 Outline

This dissertation is organized as follows. First we present a summary of related work in Chapter 2. Then we provide a background on Bayesian theory for machine learning in Chapter 3, followed by a brief tutorial on probabilistic programming and Bayesian network implementation with the Pyro python library in Chapter 4. We then present the two main studies conducted in this work. Chapter 5 presents an exploration of Bayesian neural networks for a simplified task, MNIST digit classification, in which we focus on the potential of BNNs for generic computer vision without worrying about the complications of driving tasks. The second study in Chapter 6 involves the more real-world application of semantic segmentation for scene understanding. In this we look deeper into how we can use Bayesian models to augment normal segmentation with uncertainty features for self-driving cars. Finally we conclude with a discussion of the results and potential future research directions in Chapter 7.

Chapter 2

Related Work

In this section we will discuss the prior work relevant to this research. First we present an overview of Bayesian neural networks and other kinds of stochastic neural network models. Then we provide relevant background on computer vision and scene understanding, leading up to our case study in semantic segmentation. Finally we discuss some limited work that tries to combine Bayesian learning with convolutional neural networks.

2.1 Bayesian Neural Networks

One of the first attempts to apply Bayesian methods to neural networks with back propagation was by Buntine and Weigend [5] in 1991. They discuss various methods of approximating uncertainties for network weights and incorporating prior probabilities, with applications for network pruning and estimating model uncertainty. MacKay [30] explored similar applications of Bayesian uncertainty for neural networks, showing that model evidence is correlated to generalization error and could be used for model selection and comparison.

Hinton and van Camp [16] used a Bayesian model for network weights as a form of regularization. They showed that finding a *Minimum Description Length* form of a neural network, based on information theory, was equivalent to minimizing the distance of the posterior weight distribution from a given prior distribution. They used a diagonal Gaussian to represent the network weight posterior distribution and constructed a loss function based on KL divergence from the prior. This was the first application of variational inference to NNs.

Neal [31] in his thesis suggested an alternative approach to Bayesian NNs using Hamiltonian Monte Carlo techniques. Rather than using an explicit approximating distribution for the network weights, this uses a dynamical simulation (in an analogy to physics) to generate Markov chains to approximate the distribution. This requires fewer assumptions about the distribution, and can incorporate any prior distribution, but does not scale to larger data sizes.

These early techniques were generally difficult to scale to large network architectures and data sizes. They all suffer from intractable integrals that must be solved in order to find the optimal probability distribution. Later works attempted to make these problems more tractable.

More recently, in 2011 Graves [13] used data sub-sampling to make variational inference more scalable, by drawing weight samples for each data sample instead of applying sampled weights over the entire dataset. The author used a diagonal Gaussian distribution similar to Hinton and Keeping. This work was further improved by Blundell et al. [3] who used a reparameterization of the log likelihood that leads to unbiased gradient estimates for back-propagation. They also extended the method to non-Gaussian priors. Their algorithm is known as *Bayes by Backprop*.

A further improvement to the Bayes by Backprop algorithm is by Kingma et al [21], who introduced the *local reparameterization trick*. Their innovation is that instead of randomly sampling the model weights and multiplying them with the inputs to get the activations, they instead sample from the activations directly. To do this they first calculate the posterior distribution of the activation as a function of the weight posterior, which given that the weight posterior is a diagonal Gaussian will also be a Gaussian. This means they can merely compute the μ and σ of the activations from those of the weights and the input values. This reduces the sampling necessary and results in lower variance of the gradients. They call the result *variational dropout* as an generalization of Gaussian dropout, described below.

Our work in this thesis is primarily based on the methods by Graves and Blundell et al. Their algorithms are generic enough that they are implemented in the open source Pyro [2] library for probabilistic machine learning, which is the implementation we use. We chose not to explore the local reparameterization method of Kingma et al. due to its increased complexity, especially for CNNs; although there has been at least one attempt to apply this to CNNs, which we will discuss later.

2.2 Other Methods for Uncertainty

There have been several other methods proposed that incorporate uncertainty into neural networks.

One of the most well-known is *dropout*, introduced by Geoffrey Hinton et al. in 2012 [15]. It was proposed as a regularization method to reduce overfitting, particularly the "co-adaptation" of individual neurons to fit the training data. By randomly omitting some proportion of the neurons for each training sample (setting the activations to zero), individual neurons can learn more robust features that are less dependent on the overall configuration of the network and can better withstand noise.

Dropout was originally applied only during training. However, Gal et al. [11] proposed in 2016 the use of dropout as a Bayesian approximation, known as Monte Carlo dropout. In this approach randomly zeroing neuron outputs is also performed at test time. This allows a distribution of outputs to be sampled and uncertainty obtained. Gal showed that this approach compared favorably with variational inference-based methods in terms of both accuracy and uncertainty estimates.

Gal later expanded on the use of dropout and other stochastic regularization techniques (SRTs) for Bayesian inference in his PhD dissertation [9]. SRT is defined as any form of regularization that injects stochastic noise into the model. He shows that for some choice of approximating distribution, variational inference yields an equivalent optimization problem to dropout. He further shows that other SRTs such as multiplicative Gaussian noise [38] can be recovered from different choices of variational distributions, and that any approximating distribution and prior satisfying some constraint corresponds to an SRT with a similar training procedure to dropout. He then shows how to use these to obtain model uncertainty.

One more method for generating uncertainty is using ensembles of networks. An ensemble means that several models are trained on the same data with different random initializations. Lakshminarayanan et al. [24] developed a method for using ensembles to derive uncertainty comparable to Bayesian networks. One limitation of this approach is that it requires memory proportional to the number of ensemble networks trained to store the parameters.

While these alternative methods of approximate Bayesian network representation are interesting and can provide performance benefits, they do have their limitations compared to the variational Bayesian approach. For instance, Shridhar [37] noted that they the MC dropout technique doesn't allow prior knowledge of the distribution to be incorporated. Thus we chose not to follow this route.

2.3 Perception and Semantic Segmentation

Yan LeCun [27] was the first to use back-propagation to learn convolution kernel coefficients in a neural network for digit recognition, demonstrating the promise of Convolutional Neural Networks (CNNs) for computer vision. In 1998 he improved on this with the model known as LeNet-5 [25], also introducing the now-standard MNIST dataset. More than a decade later, CNNs accelerated by GPUs first entered into mainstream machine learning with AlexNet [22], which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. From there deep learning for computer vision advanced rapidly, with the state-of-the-art for image classification being set by Microsoft's deep residual networks (ResNets) [14], which won the ILSVRC 2015. ResNets avoid the problem of very deep models being harder to train by adding skip connections between layers, allowing later layers direct access to earlier inputs which makes training more stable.

Object detection is a slightly more recent field. Object detection involves computing bounding boxes and class labels for one or more distinct objects in an image. The first major use of CNNs to perform object detection was R-CNN [12], which used a non-learning-based region proposal algorithm to generate bounding boxes, followed by a convolutional network to derive rich features from the identified subregion of the image, and a linear SVM to produce the final object classifications. This was followed up by Faster R-CNN [35], which improved on this by using convolutional layers for the region proposal network, followed by a Region of Interest (RoI) pooling layer which slices the convolutional feature map based on the generated bounding boxes, and finally a fully connected component that runs on each region of the feature map to produce the final detections/classifications. Later work including the YOLO [34] and SSD [28] models improved on detection speed even more by eliminating the multiple stages in favor of a single pass from convolutions to outputs, where the output is mapped to a grid and each grid point generates a bounding box regression and class labels along with confidence scores. These models are on par with image classification models in terms of both speed and accuracy.

Semantic segmentation is a slightly simpler task than object detection, although potentially expensive, as there are no bounding boxes, but a classification vector has to be computed at each pixel. Early methods like Ciresan et al. [7] applied a traditional CNN over a region centered at each pixel corresponding to the output, requiring in many expensive model evaluations. Long et al. [29] obtained better results with a fully convolutional network (FCN) which made use of an upsampling (deconvolution) layer to produce high-resolution outputs. The U-Net [36] improved on the FCN with a symmetric architecture consisting of a sequence of downsampling convolutions followed by a sequence of upsampling convolutions, concatenated with the corresponding feature maps from the downsampling part. Pyramid Scene Parsing Network (PSPNet) [42] uses a different approach without deconvolutions by upscaling and concatenating convolution outputs of different scales before a final convolution including all scaled feature maps, known as a pyramid pooling module (PPM). It also uses dilated convolutions in the downsampling part to increase the receptive field. Finally, UPerNet [40] combines this PPM with a feature pyramid network (FPN) which combines different upscaled feature maps with corresponding scaled feature maps from the downsampling (encoder) stage.

2.4 Bayesian Perception

Applications of Bayesian machine learning to computer vision is a relatively new topic, but there are some existing works. Gal et al. [10] used a Monte Carlo dropout approximation to Bayesian inference for CNNs, which he also expanded upon in his above mentioned thesis [9]. Kendall and Gal [20] later examined the implications of modeling both *aleatoric* and *epistemic* uncertainties in computer vision. Shridhar et al. [37] recently introduced an efficient method for Bayesian Convolutions using the local reparameterization trick mentioned above to transfer uncertainty in the weights into uncertainty in the activations. He did this by computing two separate convolutions for each CNN layer, one for the means and one for the standard deviations, which are then used to sample the resulting activations. [20]

Kampffmeyer et al. [18] has applied the MC dropout Bayesian approximation to the task of semantic segmentation. This is the first work we know of to attempt this. They obtained uncertainty scores by computing standard deviation of softmax scores across multiple model passes with dropout, and averaged these across all classes.

While researchers have implemented Bayesian approaches to image classification, as well as MC dropout for semantic segmentation, the use of BNNs with variational inference for semantic segmentation has not to our knowledge been addressed, nor has there been an analysis of coherent representations for segmentation uncertainty. This is where the contribution of your work lies.

Chapter 3

Bayesian Theory for Machine Learning

Before discussing the implementation and applications of Bayesian neural networks in this work, we will first review the theory behind Bayesian models and variational inference.

3.1 Bayesian Modelling

In general, in machine learning we have a dataset $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_n\}$ and are trying to learn a function $y = f_\omega(x)$ with some parameters (or weights) ω. In the ordinary frequentist approach, we want to learn a point estimate representing the maximum likelihood estimation or MLE for ω. In the Bayesian approach, on the other hand, we are trying to learn a probability distribution for ω. To do this need to define two more functions, the *prior* distribution for $p(\omega)$ and the model likelihood distribution $p(y|x, \omega)$. The prior essentially defines our initial guess for what the distribution of our model parameters should be, without considering the training data. The likelihood function defines the relationship between inputs and outputs in a probabilistic interpretation. For instance, for a classification task, this can be defined with a softmax function:

$$p(y = c|x, \omega) = \frac{\exp(f_{\omega,c}(x))}{\sum_{c'} \exp(f_{\omega,c'}(x))} \tag{3.1}$$

What we want to derive is the *posterior* probability distribution of ω, which is the probability of ω after taking the observed data into account. This can be solved for using Bayes' theorem:

$$p(\omega|X, Y) = \frac{p(Y|X, \omega)p(\omega)}{p(Y|X)} \tag{3.2}$$

If we can solve this then we can perform model inference to predict y^* given x^* by taking an expectation over the weight distribution:

$$p(y^*|x^*, X, Y) = \int p(y^*|x^*, \omega)p(\omega|X, Y)d\omega \tag{3.3}$$

If we can sample from the posterior weight distribution, than we can estimate this integral easily by the Monte Carlo method, i.e. averaging the likelihood over many weight samples.

The difficulty in evaluating equation 3.2 is the unknown denominator, the *model evidence*, also called *marginal likelihood*. This is the probability that the dataset would be observed for *any* model parameters, given our model definition. We can rewrite this by marginalising the likelihood over ω, as follows:

$$p(\omega|X, Y) = \frac{p(Y|X, \omega)p(\omega)}{\int p(Y|X, \omega)p(\omega)d\omega} \tag{3.4}$$

Unfortunately in the general case this integral is intractable, because the search space for ω is too large. One way to alleviate this is to use an approximation to the true posterior distribution. This is the approach of variational inference.

3.2 Variational Inference

The basic idea of variational inference is to replace an arbitrary posterior distribution over unobserved variables conditioned on observed data $p(\omega|X)$, which cannot be estimated directly, with an approximating or *variational* distribution $q_\theta(\omega)$, which can be expressed in closed form with parameters θ. (**Note:** for simplicity in this section I will use just X to represent the dataset, omitting Y). In order to obtain an optimal approximation we want to minimize some measure of distance between $q_\theta(\omega)$ and $p(\omega|X)$. The most convenient distance measure between distributions is the Kullback-Leibler (KL divergence) [23], which in general is defined as:

$$D_{KL}(P||Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx \tag{3.5}$$

KL divergence is 0 when two probability distributions are identical, and increasingly positive the farther apart the distributions are. In our variational inference case we have

$$D_{KL}(q_\theta(\omega)||p(\omega|X)) = \int q_\theta(\omega) \log \frac{q_\theta(\omega)}{p(\omega|X)} d\omega \tag{3.6}$$

We can't compute this directly since this still depends on knowing the true posterior p. However, we can rearrange this a bit:

$$\begin{aligned} D_{KL}(q_\theta(\omega)||p(\omega|X)) &= \int q_\theta(\omega) \log \frac{q_\theta(\omega)p(X)}{p(X,\omega)} d\omega \\ &= \int q_\theta(\omega)(\log q_\theta(\omega) + \log p(X) - \log p(X,\omega))d\omega \\ &= \log p(X) + \int q_\theta(\omega)(\log q_\theta(\omega) - \log p(X,\omega))d\omega \end{aligned} \tag{3.7}$$

$$\log p(X) - D_{KL}(q_\theta(\omega)||p(\omega|X)) = \mathbb{E}_{q_\theta(\omega)}[\log p(X,\omega) - \log q_\theta(\omega)] = \text{ELBO} \tag{3.8}$$

This gives us what is known as the *Evidence Lower Bound* (ELBO), since the KL divergence is non-negative so this represents a lower bound on the evidence term $\log p(X)$. More importantly, maximizing this term is equivalent to minimizing the KL divergence since $\log p(X)$ is constant. It also turns out that this formula is actually tractable: it can be evaluated by Monte Carlo sampling $\omega \sim q_\theta(\omega)$, rather than requiring integrading over all ω.

We can also see some intuition for the ELBO loss function with a little more rearranging:

$$\begin{aligned} \text{ELBO} &= \mathbb{E}_{q_\theta(\omega)}[\log p(X|\omega) + \log p(\omega) - \log q_\theta(\omega)] \\ &= \mathbb{E}_{q_\theta(\omega)}[\log p(X|\omega)] - D_{KL}(q_\theta(\omega)||p(\omega)) \end{aligned} \tag{3.9}$$

In this form, we can see that there are two key components to this loss function: the first term is trying to *maximize* the expected log likelihood of the model, meaning we want the training data to be assigned high probabilities by the model; the second term is trying to minimize the distance between the variational distribution and the *prior* distribution $p(\omega)$.

3.3 Bayesian Backpropagation

Since we want to *maximize* the ELBO, we must negate it to use it as our loss function. During training, for each batch we must Monte Carlo sample the weights and compute forward passes through the model. In theory we must take many samples of the weights to accurately compute the ELBO, but in practice for stochastic gradient descent we only need rough unbiased gradient estimates so a single sample per batch turns out to work well enough.

One final problem however is that we need to compute the gradient of an expectation to optimize the ELBO loss. This turns out to be easy enough if we can reparameterize our random variables so they no longer depend on the parameters *theta*.

Suppose we have an arbitrary function $f(\omega)$ and we want to compute gradients of the expectation over q:

$$\nabla_\theta \mathbb{E}_{q_\theta(\omega)}[f(\omega)] \tag{3.10}$$

If we can reparameterize this such that the expectation is not dependent on ω like so:

$$\mathbb{E}_{q_\theta(\omega)}[f(\omega)] = \mathbb{E}_{q(\epsilon)}[f(g_\theta(\epsilon))] \tag{3.11}$$

In which $q(\epsilon)$ is a fixed distribution, such as a standard normal, and g_theta is a function that transforms it to the distribution q_θ; in the Gaussian case by scaling and shifting ϵ by the μ and σ. Then we can move the gradient inside the expectation:

$$\nabla_\theta \mathbb{E}_{q(\epsilon)}[f(g_\theta(\omega))] = \mathbb{E}_{q(\epsilon)}[\nabla_\theta f(g_\theta(\omega))] \tag{3.12}$$

This means that we can get unbiased gradient estimates by simply performing our Monte Carlo sampling over the reparameterized distribution (i.e. the standard normal) and scaling and shifting the weights by our parameters.

3.4 Bayesian Neural Networks

Now we will briefly illustrate how this can all be applied to Bayesian feed-forward networks and convolutional neural networks.

A standard feed-forward layer of a neural network has the structure:

$$f(x) = \alpha(Wx + b) \tag{3.13}$$

where W and b are the weight and bias parameters and α is a nonlinear activation function; for the final layer this may be a sigmoid or softmax output for classification. To apply Bayesian weights to this we simply decide on a prior $p(W, b)$ and a variational distribution $q_\theta(W, b)$, and then we can Monte Carlo sample $W, b \sim q_\theta$ to compute forward and backward passes through the model, using the negative ELBO as the loss. During inference we can generate many samples of the model weights and use them to compute a distribution of outputs and obtain both the mean or median result and the model variance for a given input.

This can be applied to convolutional layers as well with no changes to the overall procedure. All we need to do is change the model definition:

$$f(x) = \alpha(W \circledast x + b) \tag{3.14}$$

where $\circledast$ represents convolution of a set of 2D input image channels x by a set of feature kernels W.

Chapter 4

Probabilistic Deep Learning with Pyro

Pyro [2] is a Python library that works together with PyTorch to enable construction and training of probabilistic models for machine learning. The basic functionality it contains allows us to define models as joint probability distributions of latent and observed random variables as well as learnable parameters. The probabilities of any random variables sampled inside the model can be tracked, and random samples can be substituted with predetermined values to implement conditioning. Because Pyro is based on PyTorch tensors, gradients can be tracked automatically and it can easily be implemented into deep learning models.

4.1 Pyro Basics

Pyro's most basic primitive is the `sample` function. The main effect of this is to sample a random distribution. However it also enables Pyro to keep track of samples of a named random variable, as well as substitute other values for it. This is a powerful way of implementing joint probability distributions. For example if we have the following model, with y dependent on x:

```
def model():
    x = pyro.sample('x', pyro.distributions.Normal(0, 1))
    y = pyro.sample('y', pyro.distributions.Normal(x, 1))
    return x, y
```

This function will sample from the joint distribution of x and y. We can condition the model on x using `pyro.condition`:

```
cond_model = pyro.condition(model, {'x': 3})
```

Calling the new conditioned function will sample from the conditional distribution $p(y|x = 3)$.

We can also pass the `obs` keyword argument to a `sample` call to pass observed data, which has the same effect as `condition` if we know the value inside the model. This may seem useless since we could just replace the sample call with the observation, but the key is that Pyro still records the sample call and its probability, which means it can compute the *model likelihood* with respect to the observed data.

Pyro also allows random samples to be recorded and reused with the `trace` and `replay` functions. `trace` runs a stochastic model and remembers the values of all sampled variables as well as their probability density functions so their log-probabilities can be accumulated. `replay` runs a model with a trace from a previous model run and substitutes all variable samples with their values from the original run. These functions are not necessary to use directly, but are essential to the workings of Pyro's variational inference algorithm, explained below.

4.2 Stochastic Variational Inference

Pyro's most powerful feature is its Stochastic Variational Inference (SVI) algorithm. This allows us to learn models of latent variable from observed data using the techniques discussed in Chapter 3. To perform variational inference, we must first define a model function and a *guide* function. The model function should take in observed data as a parameter and sample both latent variables and the observed variable. As an example[1]:

```
def model(data):
    loc = pyro.sample("loc", dist.Normal(0., 1.))
    with pyro.plate("data", len(data), dim=-1):
        pyro.sample("obs", dist.Normal(loc, 1.), obs=data)
```

This models *loc* with a prior distribution. If we want to learn the posterior distribution $p(loc|data)$ given the observed data, we can use a *variational* distribution $q_\theta(loc)$ to approximate it, as we learned in the last chapter. In Pyro this is called a *guide* function. This looks like the following:

[1]From Pyro's documentation, `http://pyro.ai/examples/minipyro.html`. We can ignore the plate function; it is an implementation detail to perform broadcasting from the sample distribution to the data size

```
def guide(data):
    guide_loc = pyro.param("guide_loc", torch.tensor(0.))
    guide_scale = ops.exp(pyro.param("guide_scale_log",
        torch.tensor(0.)))
    pyro.sample("loc", dist.Normal(guide_loc, guide_scale))
```

This uses another Pyro primitive, `param`, to define learnable parameters, which are simply PyTorch tensors that can have gradients computed; the passed arguments are the initial values. These are the θ parameters that we want to learn to approximate the posterior distribution of *loc*.

To learn this we use Pyro's `SVI` class, which takes in the model and guide functions, a optimizer algorithm (like Adam), and a loss function, typically `Trace_ELBO` which computes the ELBO function for all latent variables and observed data. Then we call `svi.step(data)` in our training loop with a batch of data each time to compute the loss and update our parameters.

Internally, what this does is compute a `trace` of the guide function, which samples the latent variables from the variational distribution; then uses `replay` to run the model function conditioned on the values that were generated from the guide, along with the observed data; and finally accumulates all of the log-probability terms from each sample and computes a single-sample Monte Carlo approximation of the ELBO loss. Then all of the gradients can be computed and the Pyro parameters updated using PyTorch's standard autograd functionality. Note that for gradients to be correctly computed all random variables that depend on parameters must be reparameterizable as discussed in 3.3.

4.3 Probabilistic Neural Networks

Pyro also includes some convenient functionality for integrating variational inference into neural networks. The `PyroModule` class allows us to wrap PyTorch modules and have all of their parameters registered as Pyro params. The `PyroSample` class allows neural network parameters to be replaced by `sample` calls when accessed. This allows a Bayesian neural network to be quickly built by wrapping normal neural network components.

Pyro also provides automatic guide functions that can use a standard distribution for all learnable variables, for instance `AutoDiagonalNormal`, which uses an independent normal distribution for all variables. We use this guide function for our models in this work.

Chapter 5

MNIST Experiments

The first case study conducted is an application of Bayesian neural networks to simple image classification using the MNIST[26] handwritten digit dataset, one of the most common computer vision benchmarks. The MNIST classification task is to take an image of a handwritten digit and classify it as one of the ten digit labels 0-9. We have two goals for this study: first we want to demonstrate the limitations of a standard non-Bayesian neural network when it comes to expressing uncertainty. Second, we want to demonstrate how Bayesian neural networks can be used to improve the situation by filtering out misclassified examples and examples that don't fit into any class.

5.1 Deterministic CNN

For our first set of experiments we used an off-the-shelf CNN model trained on MNIST, taken from the PyTorch examples [33]. The structure of the model is shown in Figure 5.1. It consists of two convolutional layers, each with a kernel size of 5 and hidden dimensions of 20 and 50, each followed by max-pooling layers, then followed by two fully-connected layers with a hidden dimension of 500 and a final 10-way softmax. The ReLU activation function is used for all intermediate layers.

5.1.1 Model Analysis

The model was trained on the MNIST training set of 60,000 images for 10 epochs. It achieves a test set accuracy of 99.0%. This leaves little room for improvement in terms of raw accuracy; however, we are interested in the model confidence. In Figure 5.2 we plot the distribution of model confidence scores (i.e. the highest softmax outputs) for the correctly classified examples

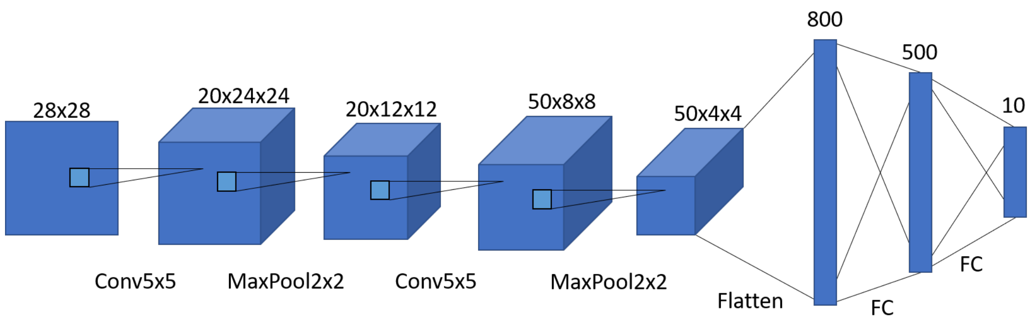


Figure 5.1: Basic MNIST CNN diagram

along with the incorrectly classified ones. We also show some examples of misclassified digits and their confidence levels (Figure 5.3).

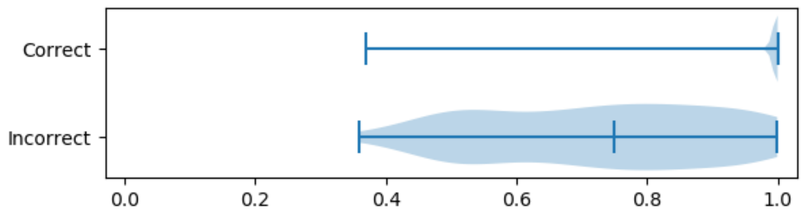


Figure 5.2: Distribution of model confidence for correctly and incorrectly classified MNIST examples

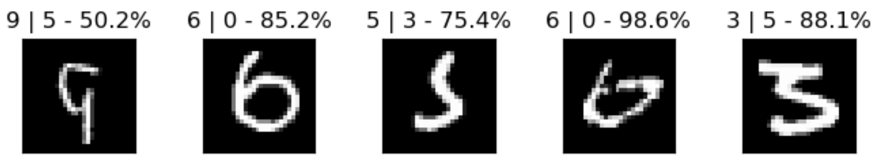


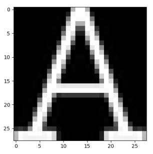
Figure 5.3: Examples of misclassified MNIST examples. They are labeled with ground truth, prediction and confidence from left to right. Some examples have misleadingly high confidences.

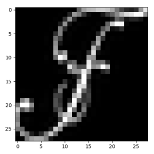
The median score for the incorrect examples is 74.9%; 25% of the scores are above 88% and 5% of them are above 99%. Most of the correct examples are above 99% confidence. If we wanted to use the model confidence to discard mispredicted examples, then there would necessarily be a tradeoff between how many *good* examples we are willing to discard and how many *bad* examples we are willing to allow (precision vs. recall). If we wanted to avoid 95% of misclassifications for example, we would have to discard more than 8% of samples.

As we can see, while the model confidence is generally lower for these "bad" examples it can sometimes be very high. What we really want to see though is how the model responds to clearly bad input, i.e. images that are not even digits. For this we conducted experiments with a separate smaller dataset of English letters, as well as some uniformly random noise images.

The letters dataset, known as not-MNIST, was obtained from the article *Making Your Neural Network Say "I Don't Know" — Bayesian NNs using Pyro and PyTorch*[6] (original dataset from [4]). It consists of 459 examples of the English letters from A-J in various fonts, in the same shape and prepared the same way as MNIST. This provides a reasonable set of images that superficially resemble digits, at least enough to fool an algorithm that has never seen anything other than a digit before, while still being distinctly not digits. Thus it seems like a good test for how our model handles out-of-distribution data.

Our naïve expectation is that on inputs that don't fit within the training domain, our model would give roughly equal scores of 10% for each of the 10 classes, indicating that all classes are equally unlikely. But in reality it gives scores much higher than 10%, ranging from the mid 40%'s all the way to 100%. We showed some examples back in Chapter 1, which we show again here (Figure 5.4):


(a) Label: 4, 100%


(b) Label: 8, 84.5%

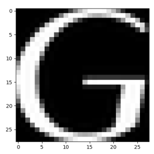

(c) Label: 3, 94.5%

Figure 5.4: Examples of MNIST classifier predictions on English letters

The fact that the A is interpreted as a four is somewhat understandable—in fact most of the A's are interpreted as 4's with high probability—but the other two are more surprising. Regardless, we would like to be able to distinguish between an actual digit and something that is clearly not a digit at all.

Even more surprising is that purely random noise is also classified quite confidently. Strangely, almost all the uniform random samples we generated were classified as an 8 with probability around 80% (Figure 5.5)

A possible explanation of this is that the difference in detail between individual random images is too high frequency for the network to detect, so

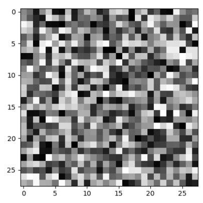


Figure 5.5: A randomly generated image
Classification: 8: 88.8%

it sees them all as vaguely grey squares, and thus computes the same label for all of them. Still, the returned probability is much higher than we would expect.

To quantify the problem numerically, on the English letters dataset, the median model confidence across all samples was 83%. This is very much higher than the expected average confidence of 10%. Thus the model seems to have a false confidence in its own predictions. To demonstrate this further we show histograms of the model confidences on each dataset in Figure 5.6:

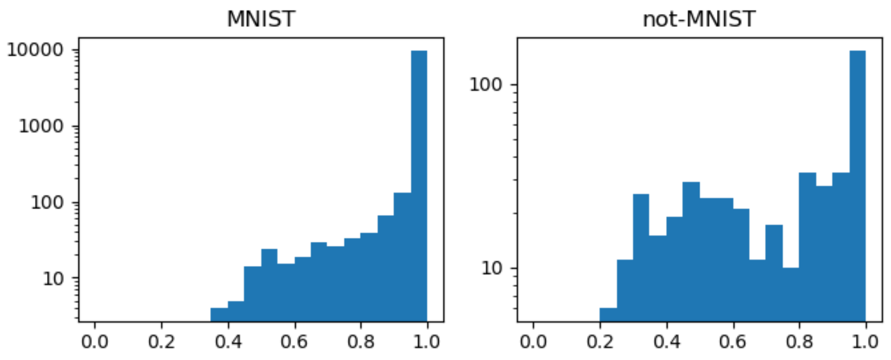


Figure 5.6: Distribution of model confidence on the MNIST and not-MNIST datasets, on a log scale

As can be seen, a large number of examples in the not-MNIST dataset are given confidences very close to 1; in fact 25% of the scores are above 98%. This makes these examples practically indistinguishable from the actual digit examples by the model.

5.1.2 Discussion

This points to a fundamental problem with deterministic networks which we would like to address. Because the network has only been trained on a finite set of data, it isn't necessarily able to generalize to data outside of its training domain. However it will still produce *some* result no matter what input we give it, and this result may be completely arbitrary depending on how similar it is to what the network has seen before, so the model confidence score can't be relied on in this case. If we could instead generate a distribution of outputs, then the variance of examples outside of the training domain should be expected to be larger than that of points within the training region, because it is less constrained by what the model has learned. Thus this variance can be interpreted as an uncertainty measure for the model.

5.2 Bayesian Neural Networks

To test this intuition, we trained two Bayesian neural networks on the MNIST dataset to compare with the deterministic model. Their architectures are shown in Figures 5.7 and 5.8. As a starting point we adapted code from this [6] article on Bayesian NNs. The first model architecture is extremely simple, a Bayesian version of a standard feed-forward network with two fully-connected layers and a softmax output. Each weight matrix is modeled as a diagonal (component-wise independent) normal distribution in terms of μ and σ, which are learned using variational inference. At test time, a number of instances of the network are sampled from the learned posterior distribution and applied to the test inputs. The highest-scoring label after averaging the scores across all model samples is chosen as the prediction.

The second model architecture is a modified version of the original deterministic CNN, with Bayesian posteriors placed over some of the weights. Instead of making the network fully Bayesian, we only replace the two final fully-connected layers with Bayesian weights, again with diagonal normal distributions. This is because adding more Bayesian parameters to a model makes it harder to train. In this way we can think of the Bayesian CNN as a deterministic feature extractor followed by a fully Bayesian feed-forward network. In order to train this we don't start the variational inference from scratch; instead we first train the deterministic version of the network (or reuse the original weights) and then train the Bayesian weights with variational inference as a sort of fine-tuning step, where the non-Bayesian weights can still update but they have a reasonable initial value. We also tried training a version with all Bayesian parameters, including the convolution and linear layers, but it was unable to converge to an optimal solution.

One evident property of these networks is that they have a less stable training process than deterministic networks. That is it may require restarting the training process multiple times before the network converges to a

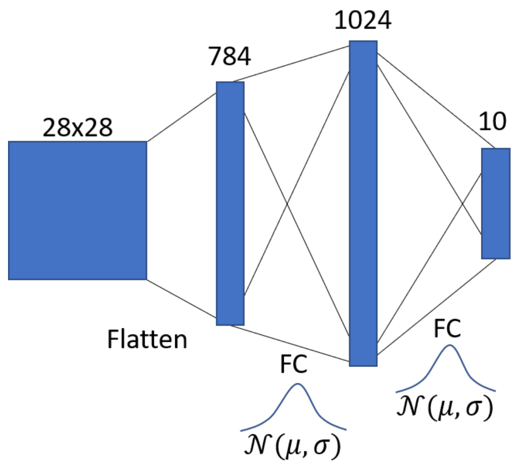


Figure 5.7: Architecture of basic Bayesian neural network. It contains 814,090 Bayesian weights (including bias terms) consisting of μ and σ.

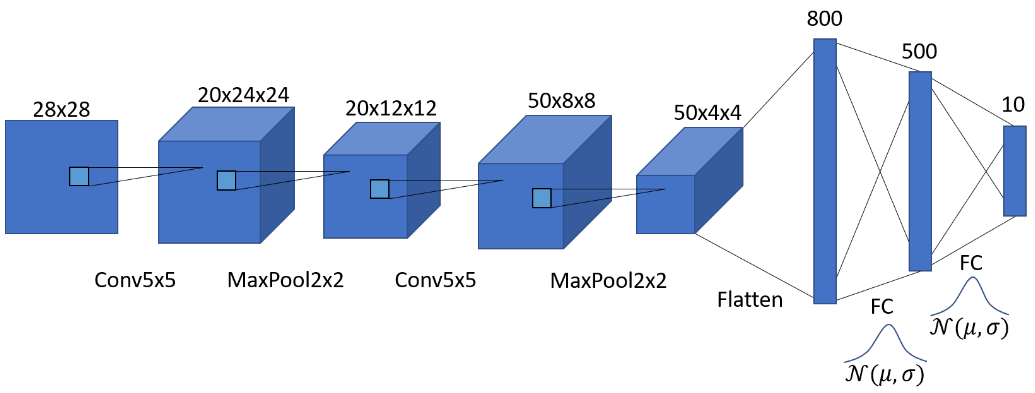


Figure 5.8: Architecture of Bayesian convolutional neural network. The convolution layers are unchanged from the deterministic model trained before, but the fully-connected layers are replaced with Bayesian weights as above. It contains 405,510 Bayesian weights/bias terms consisting of μ and σ.

reasonable locally optimum solution. Moreover even when it does converge it may lead to significantly different quality models between different training attempts. The results we show are based on the best models we selected during trained.

5.2.1 Refusing to predict

Besides computing average scores to pick the best class, our model can also refuse to predict. There are multiple ways we can interpret the stochastic model outputs in a way that allows us to discard uncertain results. The method we pick is based on a form of majority vote. We choose two thresholds, the *score threshold* s and the *majority threshold* m. The model outputs a separate softmax vector for each sample pass. The rule for classification is that in order to be classified with a given label, at least m percent of the softmax scores for that class have to be greater than s. In other words we use the score threshold to determine how many positive votes there are for the class, and then we require the number of votes to pass the majority threshold. If the condition is not satisfied we mark the example as discarded.

5.2.2 Evaluation and Analysis

We evaluate our Bayesian models in three ways. First we report the raw accuracy of the models on MNIST, taking the maximum average softmax score as the predicted class. Next we report the number of discarded samples and accuracy on the remaining MNIST data after applying our threshold rule. Finally we evaluate the model on a mixed dataset containing an equal number of examples from MNIST and not-MNIST. On this set we simply test whether the example is discarded or not; we use this to identify MNIST from not-MNIST, and count a correctly accepted MNIST example as a true positive, and a correctly discarded not-MNIST example as a true negative, etc. (we don't actually consider whether the digit class was correct, only whether it chose to classify at all). We then report precision, recall and F1 scores for this task.

The results of our experiments on both models are shown in Table 5.1, along with the deterministic CNN accuracy.

Some general observations to be drawn from this table are:

- Although our accuracy before discarding examples is slightly reduced compared to the plain CNN, the Bayesian models can produce even higher accuracy after discarding bad examples

- We can achieve over 90% precision and recall when distinguishing digit inputs from non-digits. This is promising.
- While the digit vs. non-digit selection scores are not significantly different between the two BNN models, the Bayesian CNN has higher overall accuracy than the fully connected BNN.

	CNN	Bayesian NN			Bayesian CNN		
# Samples		10	100	1000	10	100	1000
Majority Threshold		55%	63%	63%	50%	58%	58%
Raw Accuracy	99%	96%	96%	96%	97%	98%	98%
% Discarded		8%	7%	7%	11%	7%	7%
Accuracy w/ Discard		98%	99%	99%	100%	100%	100%
Precision*		90%	93%	93%	88%	92%	91%
Recall*		91%	92%	92%	88%	92%	92%
F1*		90%	92%	92%	88%	92%	92%

Table 5.1: Results of experiments on MNIST and not-MNIST datasets. Majority thresholds are tuned based on MNIST-not-MNIST precision and recall. *On the MNIST-not-MNIST selection task

Varying number of samples

In the table we show the results with varying number of model samples, from 10 to 1000, for each model. We found that both accuracy and selection ability slightly improve as the number of samples is increased, but the difference between 100 and 1000 samples is negligible. Even with only 10 samples the raw accuracy is barely reduced.

Threshold tuning

As mentioned we have two thresholds that can be tuned to trade off between false positives and false negatives. In practice we observed that the trained model almost always outputs scores close to 0 or 1, so the score threshold is somewhat arbitrary; we set it to 20%. As for the majority threshold, we tuned different thresholds for each model and different numbers of samples in order to obtain optimal precision/recall values in the MNIST-not-MNIST selection task on the balanced dataset. We can also tune it lower to reduce the number of discarded results in the MNIST set at the expense of a slight amount of accuracy; in practice this threshold should be based on the expected proportion of good and bad examples in the test data.

Prior weights and initial parameter values

There are some hyperparameters we can tune during the model selection process; one simple one is the prior weight scale. All of the weights and biases are given a prior distribution of a zero-centered normal; the scale of the normal distribution can be tuned to produce more stable training. If the prior scale is too high or too low the model may not converge. By trial and error we determined that the non-CNN Bayesian network works best with a prior scale of 10, and the Bayesian CNN we left with a prior scale of 1. We can also change the mean of the priors to be equal to the previously learned deterministic weight values, instead of 0; however this might inadvertently bias the Bayesian model to be too similar to our deterministic model, which we may not want; in practice it seems to have little effect.

One more thing we can change is the initialization of the μ and σ parameters in the variational distribution (guide) function; by default they are randomized, but we might set the initial μ parameter equal to the deterministic weight values instead, which shouldn't have the same biasing problem as changing the priors, since the loss function is not changed; nevertheless this seems to have no observable effect either.

Distribution of model confidence

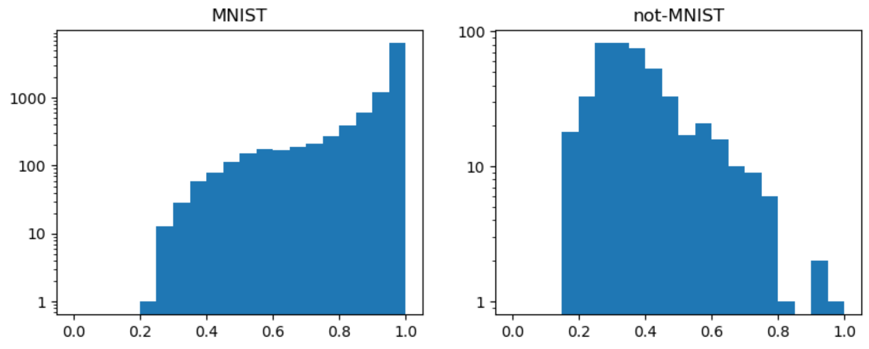


Figure 5.9: Distribution of average model confidence on each dataset, according to Bayesian CNN

Above we showed that the deterministic model's confidence scores tended to cluster around 1 even for non-MNIST images. We plot the same information for the Bayesian CNN in Figure 5.9, with the confidence scores for each sample being averaged together, and show that it is much closer to our expectation for the non-MNIST dataset.

Computational cost analysis

We also performed a timing analysis of each model based on the number of samples. The results are shown in Table 5.2, and in Figure 5.10.

	CNN	BNN						B-CNN					
Samples	1	1	10	50	100	500	1000	1	10	50	100	500	1000
Time (ms)/Input	0.085	0.06	0.11	0.32	0.58	2.7	5.4	0.065	0.13	0.45	0.86	4	8.8

Table 5.2: Inference time of each model by number of samples

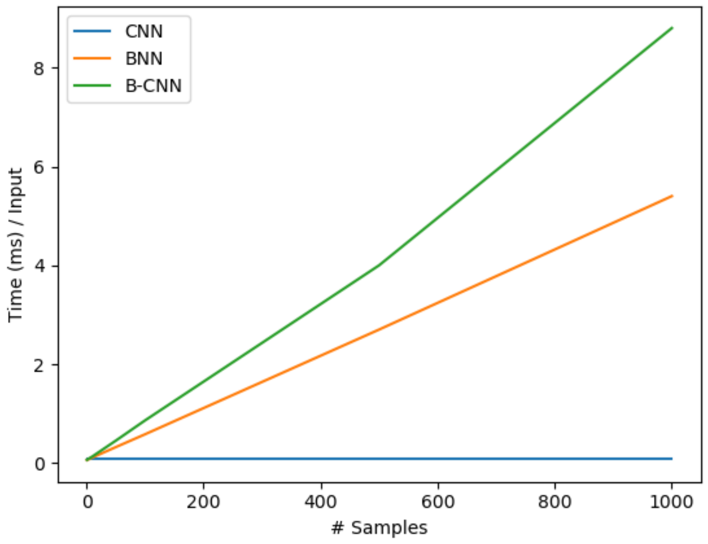


Figure 5.10: Inference time of each model by number of samples

Pretty intuitively, the inference time is linear in the number of samples taken. Also the Bayesian CNN is slower than the simpler feed-forward network.

We also looked at the time breakdown between the model sampling part and the forward propagation part of inference, for two minibatch sizes (128 and 256). The results are in Table 5.3. It is important to note that while the sampling phase takes the majority of the time in the Pyro/PyTorch implementation, this is not optimized, and the relationship would likely be more balanced if it were the model were implemented in optimized C or C++. One observation to be made is that we can reduce the proportion of time required for sampling by increasing the minibatch size, so that more instances can be processed for the same amount of random sampling. This however might not be possible in real-time applications like driving, where you generally want the inference to happen as soon as the data is received.

	BNN		B-CNN	
Batch Size	128	256	128	256
Sampling	620 (90%)	620 (84%)	869 (74%)	869 (61%)
Forward Pass	66 (10%)	119 (16%)	300 (26%)	561 (39%)
Total	686	739	1169	1430

Table 5.3: Time breakdown for sampling and forward pass, in μs

Bayesian parameter pruning

One thing that we can potentially do to reduce inference time, particularly in the sampling part, is to use parameter pruning. Ordinarily neural network pruning would involve completely removing some subset of weight values. However in this case we propose just pruning the stochastic part of the weights, i.e. the σ's. We can do this by sorting the σ's in each parameter tensor and setting the smallest ones to 0. Then during parameter sampling, we can generate a Gaussian distribution only for the non-zero σ components of the weight, and simply substitute the mean for the pruned weights. (Note that some authors [13] [3] have also suggested using the Bayesian weight distributions to prune entire weights, setting the mean to 0 as well, by testing if they are statistically close to 0. We do not implement this.) Table 5.4 shows the accuracy and precision/recall statistics of the model after pruning X% of the Bayesian weights.

This table shows that the model accuracy is not effected at all by the stochastic weights, and they can be safely replaced with their mean values if we only want an accurate deterministic model. However we can still get some use from the Bayesian component of the model even with pruning, for example by pruning 50% of the weights on the Bayesian CNN we can still get 75% precision on separating non-MNIST instances from MNIST, which means we are correctly discarding 2/3 of the bad images.

We completed a final timing analysis showing how the inference time can be reduced by pruning, shown in Figure 5.11. Note that these are completely

	% Pruned	0%	10%	30%	50%	70%	90%	100%
BNN	Accuracy	96%	96%	96%	96%	96%	96%	96%
	Accuracy w/ Discard	99%	97%	96%	96%	96%	96%	96%
	Precision	93%	88%	81%	76%	69%	58%	50%
	Recall	92%	98%	99%	100%	100%	100%	100%
B-CNN	Accuracy	98%	98%	98%	98%	98%	98%	98%
	Accuracy w/ Discard	99%	99%	99%	99%	98%	98%	98%
	Precision	92%	88%	80%	75%	70%	62%	50%
	Recall	92%	95%	97%	97%	98%	99%	100%

Table 5.4: The MNIST accuracy and MNIST-non-MNIST precision/recall for different amounts of pruning

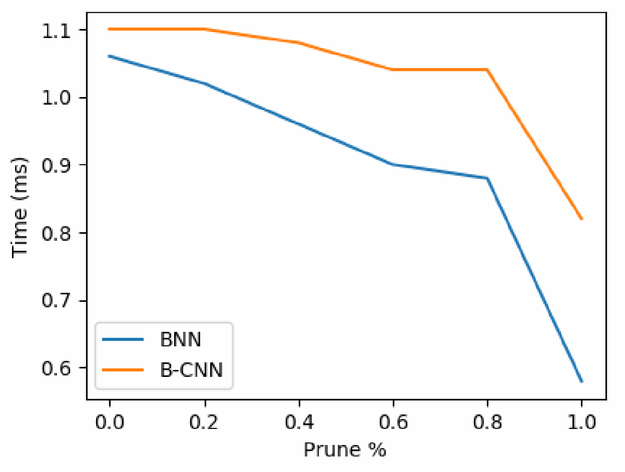


Figure 5.11: Inference time (in ms) of BNN models vs pruning %

different times than the previous chart because we are bypassing the Pyro infrastructure and doing the sampling manually along with some preprocessing, so it is highly unoptimized. Although the time saved in this implementation is not very significant until we prune 100% of the model, a more optimized low-level version should be able to save significant time as well as space due to fewer parameters requiring storage.

5.3 Discussion

We demonstrated with this experiment that Bayesian neural networks can successfully model uncertainty and allow for fairly reliably discarding low-confidence results. This isn't perfect of course, as we still necessarily have some false negatives where we are overly cautious in discarding an image, as well as some false positive images that are classified even though the should probably be discarded; we have to accept a tradeoff. We also learned that useful Bayesian uncertainty can be obtained from a convolutional neural network, even though only the non-convolutional layers were made Bayesian; this means that we don't need to make an entire network Bayesian to get probabilistic results.

The lessons learned from this study were helpful in formulating our second study on semantic segmentation. We can generalize the problem of computing uncertainty for a multi-class classification output to computing pixel-wise uncertainty where each pixel is a multi-class classification. This is the subject of the next chapter.

Chapter 6

Semantic Segmentation Experiments

In our second study we seek to provide proof-of-concept that we can obtain practical benefits from Bayesian neural networks on real-world applications; specifically for self-driving cars. The focus for this study is Bayesian networks for semantic segmentation. We picked the semantic segmentation task for two reasons:

1. It is relevant for autonomous driving
2. It is simpler to deal with than other tasks related to driving, such as object detection or lane detection

Common perception tasks for self-driving cars often output discrete geometric objects; this the case for object detection which outputs bounding boxes, and for lane detection which outputs line segments or other path descriptions. These could in theory be posed in a stochastic framework, with the implication that many potential bounding boxes could be derived for an object, or many possible paths for a lane. One complication that this presents is that the models have a variable number of outputs; there can be zero to arbitrarily many objects in a scene, and the same for lane paths. This generally implies some kind of filtering or "suppression" algorithm which reduces a large fixed number of outputs to a few. Additionally even for a single object detection, formulating a loss function is more complex than many machine learning problems as there is both a regression component (the bounding box points) and a classification component (for the object classes). We would like to avoid these complicating factors for our Bayesian neural network experiments but leave them as a challenge for future work.

Semantic segmentation on the other hand is essentially pixel-wise classification. This means the model only needs to output a fixed sized tensor with a softmax vector for each pixel. A lot of what we used for the MNIST classification model can thus be applied to this task with some minor adjustments.

6.1 Dataset Description

The dataset we use for this task is the Berkeley DeepDrive dataset (BDD100K) [41]. This comes with many different kinds of labeled and unlabeled data, including raw video footage, object detections, semantic/instance segmentation, drivable surface markings and lane markings. For our purposes we are interested only in the semantic segmentation set. This consists of 10,000 pairs of input images and target images, with class labels represented as grayscale values. An example image and target from the dataset is shown in Figure 6.1.

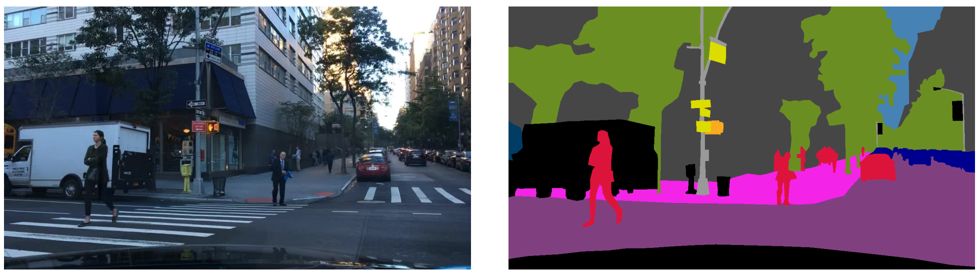


Figure 6.1: Example input image and segmentation labels (colorized) from BDD100K dataset

The dataset contains 19 categories, listed in Table 6.1:

Road	Pole	Sky	Bus
Sidewalk	Light	Person	Train
Building	Sign	Rider	Motorcycle
Wall	Vegetation	Car	Bicycle
Fence	Terrain	Truck	

Table 6.1: Berkeley DeepDrive semantic segmentation categories

6.2 Base Model

The base model we selected to perform our Bayesian modifications was taken from MIT's CSAIL Computer Vision Git repo [39][43]. Their code provides a number of different semantic segmentation model architectures split into encoder-decoder pairs. For the encoders they provide ResNet18, 50 and 101 [14] as well as MobileNet [17], modified to use dilated convolutions to increase their receptive field. MobileNet is a convolutional network optimized for mobile and embedded platforms by simplifying the convolution layers, separating them into a depth-wise convolution followed by a pointwise (1x1) convolution. For decoders they provide a simple single convolution module (C1), a Pyramid Pooling Module (PPM) [42], and a UPerNet module [40]. They provide pre-trained weights for all these networks on the MIT ADE20K scene parsing dataset. We modified the code to support training and evaluation on the BDD100K dataset.

We trained two of their model architectures deterministically, using the pre-trained weights as a starting point for transfer learning. The first is the combination of dilated ResNet50 + PPM, which is their default model. The second is MobileNetV2 + C1. The accuracy and per-class Intersection over Union (IoU) obtained on these models are shown in Table 6.2

We chose to adapt the MobileNetV2 + C1 model for our Bayesian NN experiments. While it is less powerful than the first model, it has a much smaller and simpler structure, making it both faster to train and test and simpler to convert to Bayesian, and the reduction in accuracy is low enough for our purposes. It also may be a better choice to use for autonomous driving since it is designed for more constrained hardware. (Note however that in a real self-driving car we would probably need a more accurate model.)

Before discussing the Bayesian model, here is a more detailed description of the deterministic model architecture. The encoder part of the model, as mentioned, is a modified MobileNetV2. This model contains a series of bottleneck layers with residual (skip) connections between them, similar to ResNet. Within each bottleneck block there is an initial 1x1 (pointwise) convolution, followed by a 3x3 depthwise convolution, and another pointwise convolution. The depthwise convolution, which maps each input feature map to only one output feature map, has a number of parameters linear in the input/output dimension instead of quadratic like an ordinary convolution. The depthwise followed by pointwise convolution thus results in a cheaper approximation of a single multi-channel convolution layer.

The decoder part of the model consists only of a 3x3 convolution, a batch normalization, a ReLU activation function, and a final 1x1 pointwise convolution. Note that a pointwise convolution is the same as a fully-connected layer that is applied separately to each pixel with the same weights. Finally the output is passed through a softmax for classification.

During evaluation the model scales a given input image to multiple different resolutions and passes each through the model one at a time, then bilinear

IoU	ResNet50 + PPM	MobileNetV2 + C1
Road	0.938	0.932
Sidewalk	0.636	0.618
Building	0.849	0.828
Wall	0.340	0.281
Fence	0.510	0.413
Pole	0.361	0.323
Light	0.286	0.354
Sign	0.509	0.427
Vegetation	0.863	0.845
Terrain	0.469	0.457
Sky	0.947	0.933
Person	0.615	0.563
Rider	0.328	0.188
Car	0.893	0.878
Truck	0.497	0.383
Bus	0.744	0.374
Train	0.000	0.000
Motorcycle	0.456	0.388
Bicycle	0.446	0.375
Mean IoU	0.563	0.503
Accuracy	92.8%	91.7%

Table 6.2: IoU's for each class and total accuracy for the two models

filters them back to a common size and averages their softmax scores before computing the final label at each pixel.

Note that the decoder does not have any up-scaling convolutions. This means that since the encoder reduces the feature resolution by a factor of 8, the final segmentation image is also reduced by a factor of 8, before being bilinear filtered to match the original size again. This is not a huge problem since generally we don't need particularly high resolution to detect objects.

6.3 Bayesian Model

For our Bayesian semantic segmentation model, we took an approach similar to the MNIST model in that we didn't convert the entire network to Bayesian, but only the last few layers. In this case we left the entire MobileNet encoder module fixed, treating it as a black-box deterministic feature

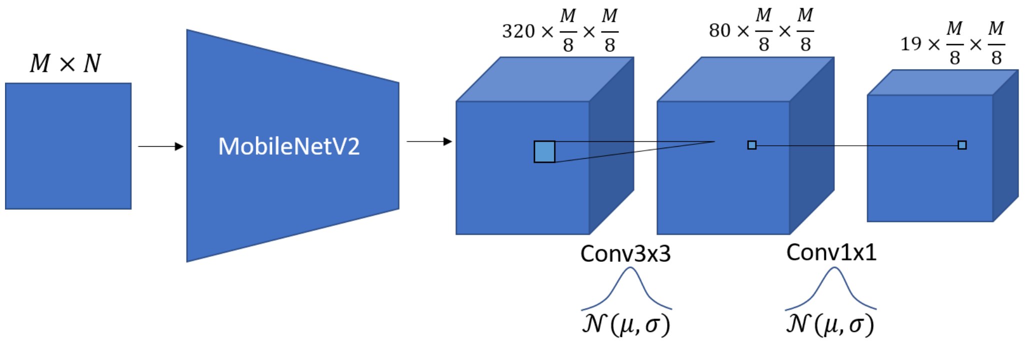

$M \times N$
MobileNetV2
$320 \times \frac{M}{8} \times \frac{M}{8}$
$80 \times \frac{M}{8} \times \frac{M}{8}$
$19 \times \frac{M}{8} \times \frac{M}{8}$
Conv3x3
Conv1x1
$\mathcal{N}(\mu, \sigma)$
$\mathcal{N}(\mu, \sigma)$

Figure 6.2: The Bayesian Semantic Segmentation model. The structure is exactly copied from the deterministic CSAIL model, but the convolutions in the decoder are replaced by Bayesian weights. This allows a distribution of softmax vectors to be output for each pixel. Note that the segmentation resolution is divided by 8. There are a total of 231,939 Bayesian weights.

extractor. We froze its weights, copied from the deterministic model, while training the Bayesian model. For the C1 decoder module we made both layers Bayesian. Note that unlike for the MNIST model, this includes Bayesian convolution kernels, although the 1x1 Bayesian convolution is again equivalent to a fully-connected layer applied on each pixel. The model architecture is in Figure 6.2. We trained the model with Stochastic Variational Inference for 10 epochs.

6.3.1 Bayesian Module Accuracy

For simple inference we take a sample of 100 model outputs and average the softmax scores at each pixel before taking the argmax, similar to what we did for MNIST. The resulting accuracy and IoU scores are shown in Table 6.3. Interestingly, the overall accuracy is a tiny bit better than the equivalent deterministic model, although the average IoU is worse.

Class	IoU	Class	IoU
Road	0.931	Person	0.462
Sidewalk	0.587	Rider	0.000
Building	0.824	Car	0.884
Wall	0.274	Truck	0.456
Fence	0.407	Bus	0.616
Pole	0.308	Train	0.000
Light	0.187	Motorcycle	0.000
Sign	0.343	Bicycle	0.003
Vegetation	0.839	Mean IoU	0.446
Terrain	0.419	Accuracy	91.8%
Sky	0.936		

Table 6.3: IoU's for each class and total accuracy for the Bayesian model

6.3.2 Identifying Error with Bayesian Uncertainty

Now we know that we can use our Bayesian network like an ordinary network and get decent accuracy, but what we really want is to use it to compute *uncertainty*. In our MNIST experiment, the only thing we could really do with uncertainty was determine whether or not to classify the image. But with semantic segmentation we can get pixel-level uncertainty.

There are multiple ways we can define this. One way is to use the variance/standard deviation of the outputs. If we take 100 samples of our model, then for each pixel of output, we have 100 softmax vectors each with 19 class probabilities. If we look only at the class we chose for that pixel based on the maximum average score, then we can measure the standard deviation of softmax probabilities for that class. If the model gives very inconsistent softmax scores for the given class then it is probably not trustworthy. Another option is to simply average the softmax scores and take $1 - p$ where p is the maximum softmax score/confidence. These two measures should be correlated because a high variance in softmax scores will tend to pull the average away from 1. Alternatively we could use a voting-based method by counting the number of scores that pass some threshold, as we did for MNIST; but this isn't substantially different from averaging the scores and discards some information, so we didn't try this.

We can also use the softmax scores from the non-Bayesian network to attempt to show uncertainty. As mentioned in the previous chapter, deterministic softmax scores tend to be over-confident and generate close to binary values, so they aren't as reliable as the Bayesian derived uncertainty values.

To demonstrate the results of uncertainty on high-error inputs, we collected a subset of the BDD dataset consisting of the 100 images attaining the lowest accuracy from the deterministic model. We call this the "bad" dataset. We also collected a "good" dataset of the 100 highest accuracy images. We can look at some of the bad examples to see how our uncertainty metrics can identify regions of high error; see Figures 6.3 and 6.4. Figure 6.5 shows a "good" image with very low error for comparison.

The first example we can see has a lot of noise due to rain covering the car's windshield; we can say the uncertainty here is *aleatoric* since it the image is inherently harder to predict. We can see that both deterministic and Bayesian networks are able to predict the two cars, but otherwise produce a lot of random spurious results in the lower left and upper right. This means we should expect to see high uncertainty in the uncertainty maps; which is the case for all three maps. However we can see that the Bayesian uncertainty maps seem to have more uncertainty concentrated in these areas.

The second image is a good example of epistemic noise. This is a case where a human could clearly identify that there is a large truck taking up most of the space in the image. However the model likely had limited training data with large flat uniform-colored trucks at that close distance; hence it somewhat reasonably thinks that part of the truck is a building or wall. The uncertainty outputs neatly reflects this; we can clearly see that the truck is given a high uncertainty by all three metrics, especially the Bayesian ones, reflecting the fact that the model hasn't learned to converge towards a definite answer for this input.

The third input is simply a good demonstration of how uncertainty works when there is very little error. All three uncertainty maps are close to zero for the most part, reflecting high confidence; although there are lines of uncertainty around the object borders in all three maps.

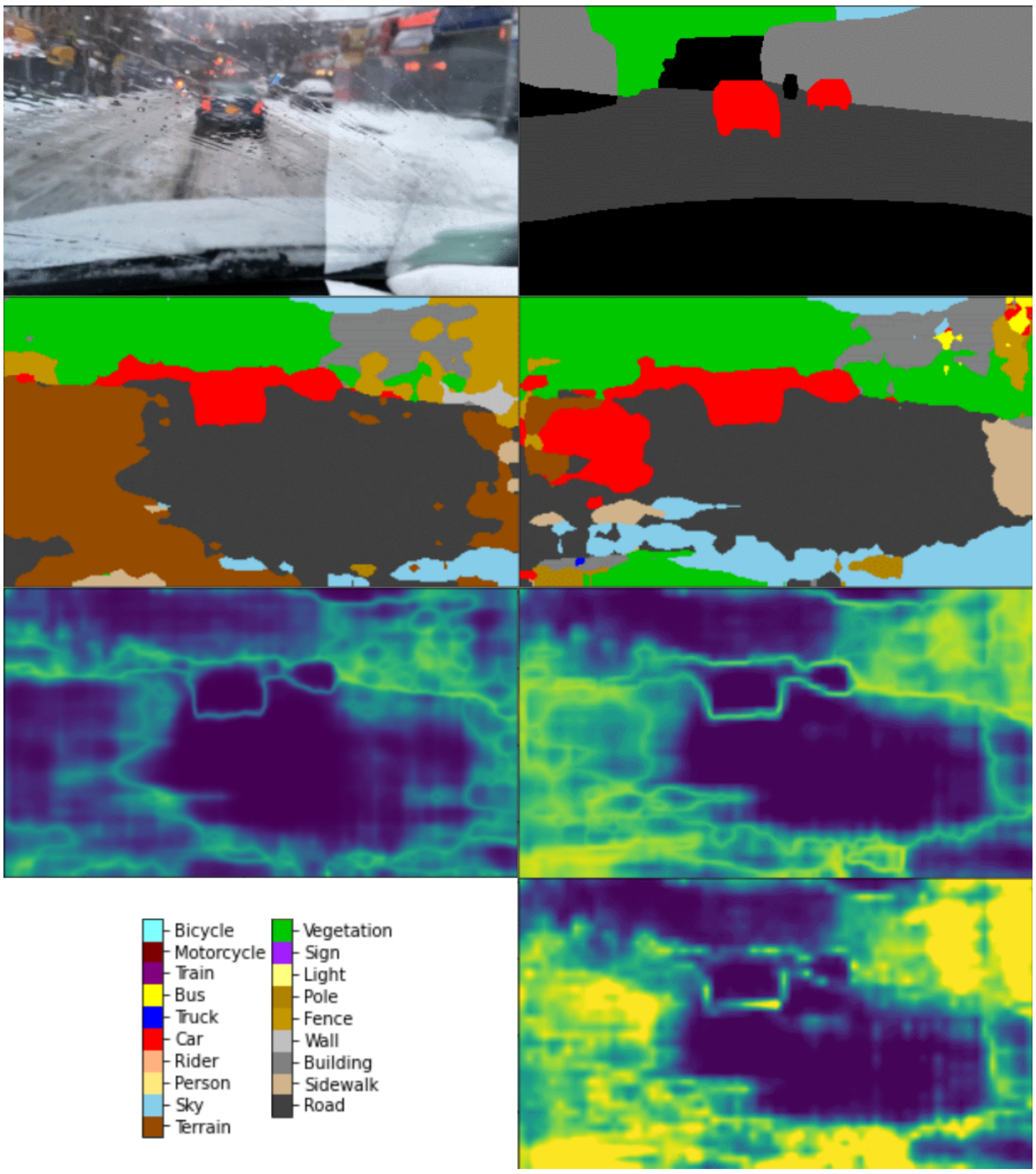


Figure 6.3: Example image with large amounts of input noise and high error. From left to right, top to bottom: input image, ground truth labels; deterministic model segmentation, Bayesian model segmentation; Deterministic softmax uncertainty, Bayesian softmax uncertainty; Bayesian standard deviation uncertainty

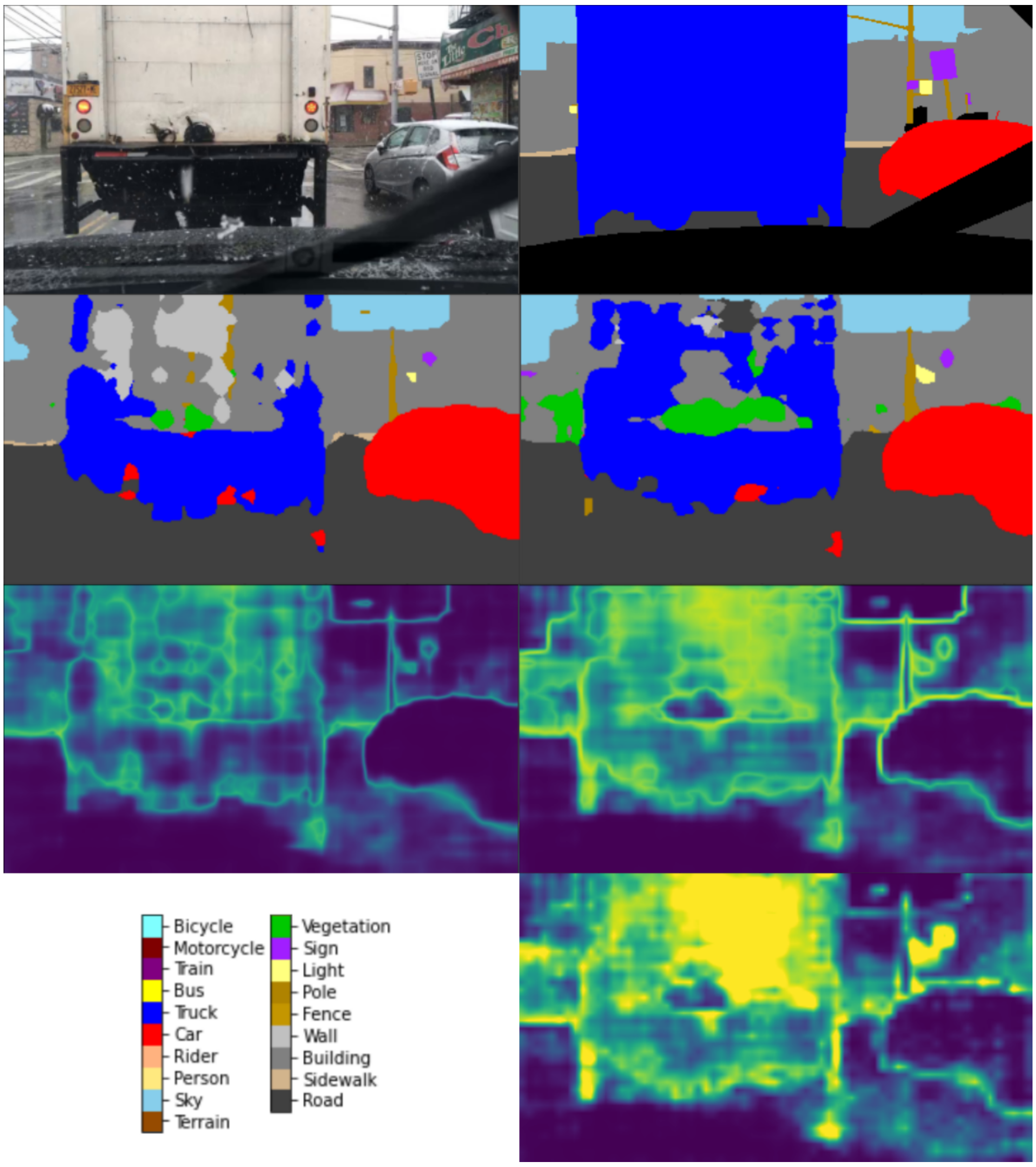


Figure 6.4: Another example with high segmentation error. From left to right, top to bottom: input image, ground truth labels; deterministic model segmentation, Bayesian model segmentation; Deterministic softmax uncertainty, Bayesian softmax uncertainty; Bayesian standard deviation uncertainty

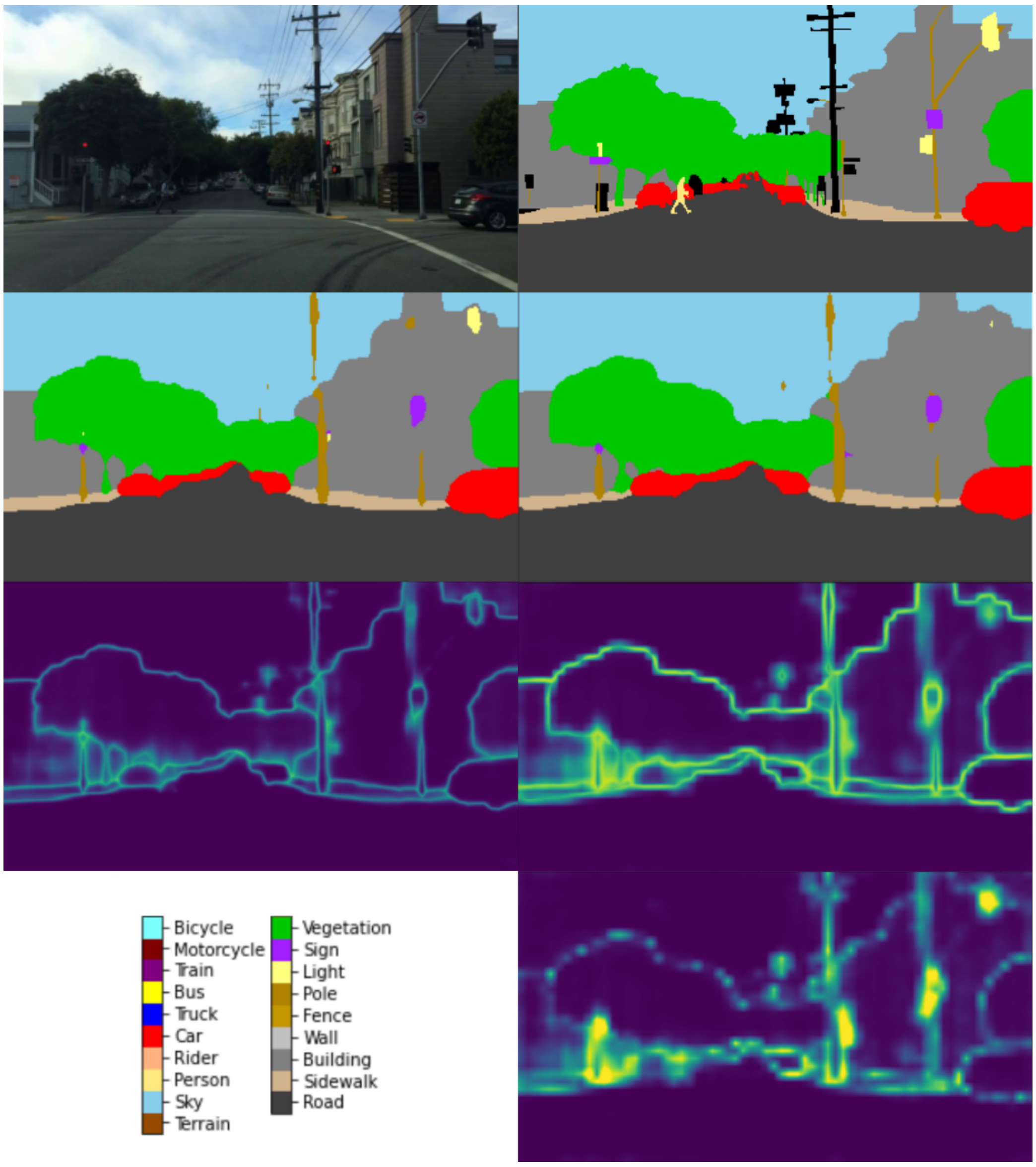


Figure 6.5: A low-error example, showing the very sparse uncertainty maps. From left to right, top to bottom: input image, ground truth labels; deterministic model segmentation, Bayesian model segmentation; Deterministic softmax uncertainty, Bayesian softmax uncertainty; Bayesian standard deviation uncertainty

6.3.3 Comparison of Uncertainty metrics

All three uncertainty metrics are clearly associated with higher error. We can see that the deterministic softmax-based uncertainty is much sparser than the other ones and yields high uncertainty mainly at object boundaries. This is consistent with the previous assertion that softmax tends to be biased towards high confidence. The Bayesian softmax uncertainty also tends to be highest around object edges. This is probably because the softmax scores are linearly interpolated when the image is upscaled, which means at any boundary between two different classes the softmax values will have to rapidly change and there will be a point where they are both less than .5. On the other hand, it makes sense to have high uncertainty at object boundaries since there are multiple classes competing for nearby pixels. The Bayesian softmax uncertainty is consistently higher than the deterministic uncertainty, which implies that it is less biased due to the added randomness.

The standard deviation-based uncertainty can't be directly compared with the softmax based ones since it is on a different scale (standard deviation verses probability). However we can determine statistically how well each uncertainty score predicts segmentation error. We computed the Pearson correlation coefficient between the uncertainty scores and the binary per-pixel errors (1 if the pixel is misclassified, 0 otherwise) for each image in the "bad" dataset and the "good" dataset, then averaged them over all images.

Bayesian Std Dev	Bayesian Softmax	Deterministic Softmax
.173	.213	.140

Table 6.4: Correlation coefficient between pixel error and uncertainty scores

We can see from Table 6.4 that both Bayesian measures of uncertainty are more correlated with error than normal softmax, although not by a huge threshold. Interestingly the Bayesian softmax seems to be a better predictor of error than the standard deviation.

6.3.4 Framework for ODD Violation

We've so far looked at the per-pixel uncertainty, but we can also consider aggregating the uncertainty over the whole image. This can be useful for detecting Operational Design Domain (ODD) violations. The simplest way to do this is simply to average the uncertainty over the whole image. If the uncertainty is above a certain threshold a self-driving car could choose to take some exceptional action like relinquishing control to the user. It could also be useful to have this information recorded so for example if a self-driving car gets in an accident, technicians could determine whether it was driving under abnormal conditions at the time by looking at the on-board logs.

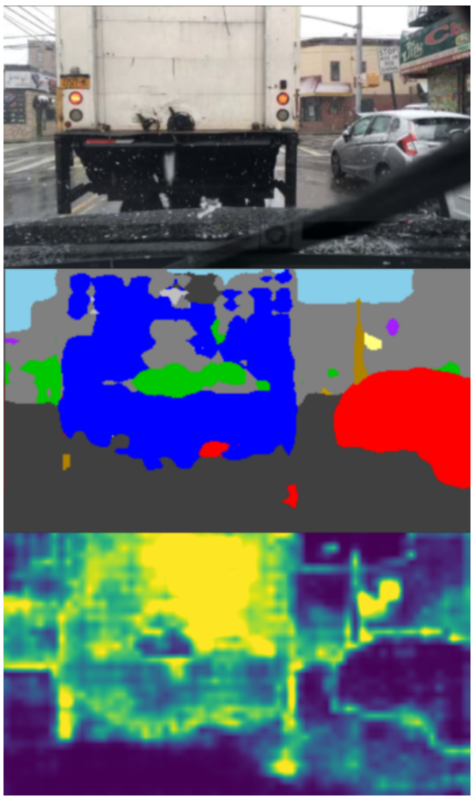
We can add additional features to an ODD violation check, for instance instead of weighting all pixel uncertainty equally, we could weight classes by "safety criticality".

We implemented two functions for the purpose of aggregating uncertainty and determining ODD violations, one that computes the total uncertainty by averaging the standard deviation over all pixels, and one that is the same but it only counts pixels with a class we consider safety-critical, with all other classes weighted 0. For this purpose the safety critical classes are determined to be Car, Truck, Bus, Person, Rider, Bicycle, and Motorcycle: basically all the objects that need to be avoided on a road.

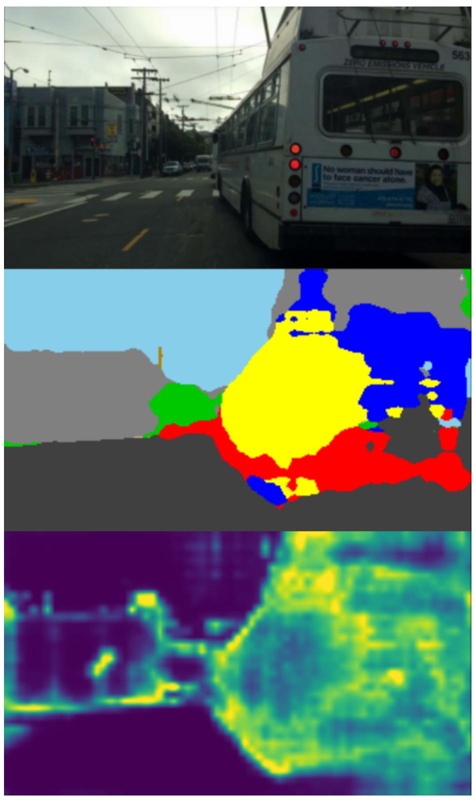
When run on the good and bad datasets, we found that the total uncertainty metric for the good images averaged around .12, while for bad images averaged .16. The safety-critical uncertainty averaged around .04 and .07 in each set respectively.

As an example, Figure 6.6 shows two images from the bad set that have higher than average critical uncertainties at around .12 (we already saw the first one above). These examples illustrate the purpose of the critical uncertainty score, because we can see they both include a large nearby vehicle with a lot of conflicting classifications, which would be a serious problem for a self-driving car.

These uncertainty scores are not precise enough to be able to reliably distinguish between high-error and low-error scenarios, but they are still useful as a tool to help the car determine how confident it is that its predictions are safe and reliable.



(a) Critical uncertainty: .119



(b) Critical uncertainty: .122

Figure 6.6: Example images with high safety-critical uncertainty score, along with their segmentation and standard-deviation uncertainty maps.

Chapter 7

Discussion and Conclusion

We implemented Bayesian neural networks and applied them to two tasks. The first task, MNIST classification, is partially recreating existing work, and the MNIST dataset is not particularly novel or state-of-the-art. However we demonstrated a meaningful proof of concept that we can use Bayesian neural networks to identify inputs with high uncertainty or that fall outside of our training domain, and thus improve our model reliability at the expense of some inputs that we refuse to classify.

The second task is, as far as we are aware, a completely novel innovation. The application of Bayesian Convolutional Neural Networks to the task of semantic segmentation for self-driving cars allowed us to derive per-pixel uncertainty metrics that intuitively mirror the sections of the input image that have high misclassification error. This can be seen in some cases to be related to high noise in the image (*aleatoric* uncertainty), such as rain; in other cases it is due to limitations of the model's discrimination ability (*epistemic* uncertainty), such as between a large white truck back and a wall, or an oddly-shaped bus and a car or truck. In either case this uncertainty is a problem that we need to address for autonomous driving. We further addressed the problem of detecting ODD violations by aggregating uncertainty over an image, and how this can be important in the decision making process of a self-driving car, for instance it may decide to pass control to a human.

There are some limitations that our work did not address which could be expanded on in future work. One improvement could be to try applying Bayesian weights to a larger part of the segmentation model, and see whether equal or better results are obtained; ablation studies could be done to determine specifically what parts of the model should be made Bayesian.

Another option that we explored with MNIST but not with the segmentation model was Bayesian weight pruning. The same experiment could be performed on the segmentation model to see if reasonable uncertainty can still be obtained with fewer Bayesian weights. Additionally there are other hyperparameters and factors we tuned on the MNIST model that we did not change on the segmentation model; for example we could try changing the prior distribution for the weights, which in our trained model was set to $\mathcal{N}(0, 1)$. Increasing or decreasing the prior scale parameter could change the distribution of our uncertainty scores.

A broader task for future work would be to apply this premise to other autonomous driving tasks. For instance as briefly mentioned we could apply Bayesian uncertainty to object detection and obtain not only distributions of class scores, but distributions of bounding box locations which may be helpful in identifying localization error. It could also be applied to lane detection to generate many possible paths which could all be ingested by the planning algorithm. Bayesian uncertainty could even be applied to the planning stage itself for end-to-end learning algorithms; this would allow the car to predict many different possible paths and select between them. There are many possible applications for learnable uncertainty in safety-critical cyber-physical systems that should be explored.

Bibliography

[1] Baidu. *Apollo*. 2020. URL: `https://github.com/ApolloAuto/apollo`.

[2] Eli Bingham et al. "Pyro: Deep Universal Probabilistic Programming". In: *Journal of Machine Learning Research* (2018).

[3] Charles Blundell et al. *Weight Uncertainty in Neural Networks*. 2015. arXiv: `1505.05424 [stat.ML]`.

[4] Yaroslav Bulatov. *notMNIST dataset*. 2011. URL: `http://yaroslavvb.blogspot.com/2011/09/notmnist-dataset.html`.

[5] Wray L Buntine and Andreas S Weigend. "Bayesian back-propagation". In: *Complex systems* 5.6 (1991), pp. 603–643.

[6] Paras Chopra. *Making Your Neural Network Say "I Don't Know" — Bayesian NNs using Pyro and PyTorch*. Nov. 2018. URL: `https://towardsdatascience.com/making-your-neural-network-say-i-dont-know-bayesian-nns-using-pyro-and-pytorch-b1c24e6ab8cd`.

[7] Dan Ciresan et al. "Deep neural networks segment neuronal membranes in electron microscopy images". In: *Advances in neural information processing systems*. 2012, pp. 2843–2851.

[8] Comma.ai. *openpilot*. 2020. URL: `https://github.com/commaai/openpilot`.

[9] Yarin Gal. "Uncertainty in Deep Learning". PhD thesis. University of Cambridge, 2016.

[10] Yarin Gal and Zoubin Ghahramani. "Bayesian convolutional neural networks with Bernoulli approximate variational inference". In: *arXiv preprint arXiv:1506.02158* (2015).

[11] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning". In: *ICML*. 2016.

[12] Ross Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2014, pp. 580–587.

[13] Alex Graves. “Practical Variational Inference for Neural Networks”. In: *Advances in Neural Information Processing Systems 24.* Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 2348–2356. URL: `http://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks.pdf`.

[14] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[15] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *CoRR* abs/1207.0580 (2012). arXiv: `1207.0580`. URL: `http://arxiv.org/abs/1207.0580`.

[16] Geoffrey E Hinton and Drew Van Camp. “Keeping the neural networks simple by minimizing the description length of the weights”. In: *Proceedings of the sixth annual conference on Computational learning theory.* ACM. 1993, pp. 5–13.

[17] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[18] Michael Kampffmeyer, Arnt-Borre Salberg, and Robert Jenssen. “Semantic segmentation of small objects and modeling of uncertainty in urban remote sensing images using deep convolutional neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops.* 2016, pp. 1–9.

[19] Shinpei Kato et al. “Autoware on board: Enabling autonomous vehicles with embedded systems”. In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS).* IEEE. 2018, pp. 287–296.

[20] Alex Kendall and Yarin Gal. “What uncertainties do we need in bayesian deep learning for computer vision?” In: *Advances in neural information processing systems.* 2017, pp. 5574–5584.

[21] Diederik P. Kingma, Tim Salimans, and Max Welling. *Variational Dropout and the Local Reparameterization Trick.* 2015. arXiv: `1506.02557 [stat.ML]`.

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

[23] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *Ann. Math. Statist.* 22.1 (Mar. 1951), pp. 79–86. DOI: `10.1214/aoms/1177729694`. URL: `https://doi.org/10.1214/aoms/1177729694`.

[24] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. “Simple and scalable predictive uncertainty estimation using deep ensembles”. In: *Advances in neural information processing systems*. 2017, pp. 6402–6413.

[25] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[26] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST handwritten digit database”. In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[27] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[28] Wei Liu et al. “Ssd: Single shot multibox detector”. In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[29] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[30] David J C Mackay. “A Practical Bayesian Framework for Backprop Networks”. In: 1991.

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

[32] NHTSA. *Federal Automated Vehicles Policy: Accelerating the Next Revolution In Roadway Safety*. Sept. 2016.

[33] PyTorch. *Basic MNIST Example*. Jan. 16, 2019. URL: `https://github.com/pytorch/examples/blob/master/mnist/main.py`.

[34] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[35] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[37] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. “A Comprehensive guide to Bayesian Convolutional Neural Network with Variational Inference”. In: *arXiv preprint arXiv:1901.02731* (2019).

[38] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[39] MIT CSAIL Computer Vision. *Semantic Segmentation on MIT ADE20K dataset in PyTorch*. Apr. 13, 2020. URL: `https://github.com/CSAILVision/semantic-segmentation-pytorch/`.

[40] Tete Xiao et al. “Unified perceptual parsing for scene understanding”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 418–434.

[41] Fisher Yu et al. “BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[42] Hengshuang Zhao et al. “Pyramid Scene Parsing Network”. In: *CVPR*. 2017.

[43] Bolei Zhou et al. “Semantic understanding of scenes through the ade20k dataset”. In: *International Journal on Computer Vision* (2018).