Uncertainty in Recurrent Neural Network with Dropout

Kha L. H. Nguyen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 25.6.2020

Thesis supervisor:

Prof. Pekka Marttinen

Thesis advisors:

D.Sc. (Tech.) Antti Ajanki

PhD.Sc. Tianyu Cui

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Kha L. H. Nguyen

Title: Uncertainty in Recurrent Neural Network with Dropout

Date: 25.6.2020 Language: English Number of pages: 7+68

Department of Computer, Communication and Information Sciences

Professorship: Machine Learning, Data Science and Artificial Intelligence

Supervisor: Prof. Pekka Marttinen

Advisors: D.Sc. (Tech.) Antti Ajanki, PhD.Sc. Tianyu Cui

Recurrent Neural Network is a powerful tool for processing temporal data. However, assessing prediction uncertainty from recurrent models has proven challenging. This thesis attempts to evaluate the validity of uncertainty from recurrent models using dropout.
Traditional neural network focuses on optimising data likelihood; in order to obtain model and predictive uncertainty, we need to, instead, optimise model posterior. Model posterior is usually intractable, thus we employ various dropout based approach, in the form of variational Bayesian Monte Carlo, to estimate the learning objective. This technique is applied to existing recurrent neural network benchmarks MIMIC-III [9]. The thesis shows that Monte Carlo dropout [6] applied to recurrent neural network [4] can give comparable performance to the current state of the art methods, and meaningful uncertainty of predictions.

Keywords: Uncertainty, Variance, Approximation, Deep learning, Recurrent, Neural network, Bayesian, Probabilistic, Monte-Carlo, Dropout, Variational Bayesian

Preface

I want to thank Professor Pekka Marttinen and my advisors Antti Ajanki and Tianyu Cui for their guidance and support in my thesis work. In particular, I'd like to thank Prof. Pekka Marttinen for providing me with this interesting thesis topic, and reading materials which assisted me in building the solution. I thank Tianyu Cui for his assistance with my PyTorch code, and for helping me understand the nature of uncertainty. I thank Antti Ajanki for assisting me in planning this thesis, reviewing the first draft, and providing suggestions to reorganize its contents. I thank Heidi Wahl for proof-reading my thesis and providing suggestions to make my thesis more fluent, and Miika Aspiala for validating my content, and providing me feedback to reorganize my thesis. I would like to express my gratitude towards my parents, and my friends and family for their emotional support. Lastly, I would like to thank the Aalto Triton team for their support in running my experiments, and to Aalto University, for these last three years of instruction which made thesis possible.

Otaniemi, 25.6.2020

Kha L. H. Nguyen

Contents

Symbols and abbreviations

Symbols

ω	A set of parameters in neural network model to optimise
$\mathbf{M}$	Variational matrix
$\mathbf{W}$	Weight matrix, an element of parameter set ω
$\mathbf{I}_K$	Identity matrix of size $K \times K$
$\mathbf{x}$	A training input vector
$\mathbf{x}^*$	An input vector for prediction
$\hat{y}$	An output from neural network model
y	A training label/output
y^*	A predictive label/output
p	A dropout rate
$\mathbf{z}$	A dropout mask vector
$\mathbf{p}$	A softmax probability vector; equivalent to y for classification tasks
$\mathcal{D}$	A set of training data, including inputs and labels/outputs
K	Number of dimensions
L	Number of network layers
T	Number of Monte Carlo samples
$\mathcal{N}$	A Gaussian distribution
Ber	A Bernoulli distribution
ϵ	A sample from the standard normal distribution $\mathcal{N}(0, 1)$

Operators

$\mathbf{x} \circ \mathbf{u}$	Element-wise multiplication between vector $\mathbf{x}$ and $\mathbf{u}$
$\mathbf{xW}$	Matrix multiplication between vector $\mathbf{x}$ and matrix $\mathbf{W}$
$diag(\cdot)$	A function to turn a vector into a diagonal matrix
$\sigma(\cdot)$	An activation function
$\log(\cdot)$	Natural logarithm function
$sigmoid(\cdot)$	A sigmoid function; usually a logistic function
$p(\cdot)$	Probability distribution function
$q(\cdot)$	Approximated probability distribution function
$\sim$	A sample from distribution function
$E_p[\cdot]$	An expectation over distribution p
$Var_p[\cdot]$	A variance over distribution p
$\sum_i$	sum over index i
$\mathcal{H}(\cdot)$	An entropy function

Abbreviations

CNN	Convolutional Neural Network
KL	Kullback-Leibler
LSTM	Long short-term memory
MC	Monte Carlo
MSE	Mean square error
NN	Neural Network
RNN	Recurrent Neural Network
VB	Variational Bayesian
s.t.	Such that
e.g.	Exempli gratia ("for the sake of an example")
i.e.	Id est ("it is")
w.r.t.	With respect to

Chapter 1

Introduction

Uncertainty quantification is desirable in critical decision making such as clinical diagnosis and prediction. This thesis attempts to quantify predictive uncertainty from recurrent neural network (RNN), specifically long-short-term-memory (LSTM) network, by using Monte Carlo (MC) dropout, a novel technique proposed by Yarin Gal [6]. This technique can be applied to other types of RNN, and non-RNN network, such as feedforward, convolution, or attention networks (see appendix B).

MC dropout is built upon Bayesian statistics. Bayes' theorem gives us the tool to express belief at the presence of some condition, thus Bayesian inference adds two (2) additional properties to a regular model: 1) belief of the parameters, or prior, which usually results in weight regularizing terms in the objective function, and 2) posterior uncertainty. Variational Bayesian (VB) is used to optimise an otherwise intractable (difficult or impossible to express analytically in formulae) posterior; the use of dropout (in both training an inference) is equivalent to approximating posterior with a tractable neural network (NN) weight distribution 2.2, which we will further explain.

The work presented in this thesis is compared against *Multitask learning and benchmarking with clinical time series data* [9], a benchmark in clinical time series data. The tasks in this benchmark includes: in-hospital-mortality (IHM), length-of-stay (LOS), decompensation (DEC), phenotyping (PH), and a combination of the four (4) (Multitask). These tasks are formulated as classification; application to regression is cover in appendix C.

1.1 Uncertainty

Before we embark on quantifying uncertainty on neural network, lets understand what uncertainty really signifies. In a restaurant, when presented with a menu of just a few items, we do not have much difficulty in picking a dish. When we are presented with a large menu with many items, spanning across multiple pages, it would take us longer to pick a dish, and even after picking a dish, we are still *uncertain* if we picked the one we would like the most, least without prior knowledge of this restaurant. What we have just experienced is an effect of uncertainty, a judgement of how good of a choice we make, or the *level of confidence* in our choice. In analogy, a menu is a random variable, and a sample from this random variable is a dish we pick. Note that a sample does not have uncertainty. A random variable (under a distribution) has uncertainty. Therefore, when we say we want to find uncertainty of a prediction in NN, we treat NN outputs (or predictions) as random variables of a distribution. Uncertainty manifests as many different possible outcomes under the same condition.

Like humans, NN models can make predictions. But like in the case for human predictions, predictions by NN models always come with a level of uncertainty, determined by knowledge, experience, and available data. In traditional NN models, we mostly output prediction, but do not output uncertainty, or confidence level. In a regression task, we do not produce uncertainty to quantify how reliable a prediction is. In a classification task, we mistakenly treat the softmax probability output vector as confidence level. Softmax outputs can be interpreted as parameters for a Categorical distribution, but not its variance (which is $p(1-p)$). Softmax outputs can also be understood as bringing pre-softmax output into a range that humans can interpret, e.g. if a prediction that an image contains a cat is 6.43, can we determine if that is a cat or not? Softmax gives us two anchor points, 0.0 and 1.0, to aid our interpretation of the results. Another counter example that softmax output should not be treated as confident level is, suppose we have a pre-softmax output of 100.00, for some input far from training data inputs, softmax output is then very close to 1.0, indicating high confidence for out-of-distribution input. This large discrepancy between pre-softmax and softmax outputs prompts us that we need to introduce prediction uncertainty to existing NN models.

How do we introduce uncertainty to NN? Recall that NN is a special form of the linear regression model (See section 2.1). The linear model builds on/comes with Bayesian probability techniques: Bayesian regression and Bayesian logistic regression. Borrowing from the linear model, this work introduces Bayesian probabilistic to NN, as a means to acquire uncertainty of prediction. Lets review basic Bayesian regression.

$$\begin{aligned} p(\omega|\mathcal{D}) &= \frac{p(\mathcal{D}|\omega)p(\omega)}{p(D)} \\ p(\omega|\mathbf{Y},\mathbf{X}) &= \frac{p(\mathbf{Y}|\omega,\mathbf{X})p(\omega)}{p(\mathbf{Y}|\mathbf{X})}, \end{aligned} \tag{1.1}$$

in which:

- ω represents the parameters we want to optimise
- $D = \mathbf{X}, \mathbf{Y}$ represents training data, where $\mathbf{X}$ are inputs, and $\mathbf{Y}$ are outputs.

In linear models, ω represents linear weights and biases. In NN, ω represents network weights and biases. The expression $p(\omega|\mathcal{D})$ is called posterior, or probability of ω after we have observed training data; here we want to find ω that maximizes the posterior given the observed data.
Traditionally, most NN maximize the likelihood term $p(\mathcal{D}|\omega)$ (or, equivalently, minimize the negative likelihood) because it is simpler[1]. This likelihood, however, does

[1] depending on the nature of training data

not express variance in prediction, but rather only variance in training data. What if training data is noisy? ω will try to capture noise as well, resulting in what we call overfitting (noises become representatives for otherwise unseen cases). One approach to avoid overfitting is to restrict parameters ω into a family (from a distribution). This is expressed as $p(\omega)$, and describes a prior belief that parameters ω belong to family $p(\omega)$.

If we pick a standard Gaussian prior, for example, it results in L2 (squared magnitude of weight matrix) regularizing terms in the objective function. The normalizing term $p(\mathbf{Y}|\mathbf{X})$ is not a function of ω, and can therefore be excluded from the optimization objective. Notice that Bayesian approach does not automatically guarantee good generalization. If we pick a suitable prior for ω, we can avoid both overfitting and underfitting. If we pick a prior far from the unknown family of suitable parameter distributions, we suffer overfitting, or underfitting[2].

We are interested in **predictive uncertainty**. Naturally, we can find it from a predictive distribution

$$\begin{aligned} p(y^*|\mathbf{x}^*, \mathcal{D}) &= \int_\omega p(y^*, \omega|\mathbf{x}^*, \mathcal{D}) d\omega \\ &= \int_\omega p(y^*|\omega, \mathbf{x}^*) p(\omega|\mathcal{D}) d\omega, \end{aligned} \tag{1.2}$$

with variance, or uncertainty expression, from definition as:

$$Var_{p(y^*|\mathbf{x}^*,\mathcal{D})}[y^*] = E[(y^*)^T(y^*)] - E[y^*]^T E[y^*]. \tag{1.3}$$

Looking at the predictive distribution (1.2), we can get an idea of where the uncertainties [1] come from:

- Noisy input data from $p(y^*|\omega, \mathbf{x}^*)$, or *aleatoric uncertainty*
- Uncertainty from model parameters conditioned on training data $p(\omega|\mathcal{D})$, or *epistemic uncertainty*

A challenge in determining uncertainty is that the posterior distribution is intractable for neural network; thus we shall address this challenge by Variational Bayesian[3] approximation later on in section 3.1.

1.2 Proposal to quantify uncertainty in RNN

Uncertainty has been quantified in non-recurrent NN, specifically convolutional neural network (CNN), by Yarin Gal et al. [15] using a stochastic process. Gal et al. also proposed the same stochastic process as *a regularising technique for RNN* [4]. The work in this thesis combines these two (2) major papers to quantify uncertainty in

[2]Note that overfitting can happen for other reasons, such as unbalanced, or biased training data, in which case, cross-validation is also required

[3]Another Bayesian concept, not to be confused with Bayesian modeling

RNN; the objective function, the uncertainty computation, and stochastic model schema are derived.

Various works (see chapter 2) are utilised to customize uncertainty quantification for regression and classification tasks. Some mathematical derivations, although given from previous works, are provided in this paper in greater details and with explanations in order to justify the mathematical expression of uncertainty.

The stochastic LSTM layer, one of the building blocks for stochastic NN models, was implemented in both Keras and PyTorch (see appendix D and E).

The uncertainty quantification solution was implemented in both Keras and PyTorch (see appendix F).

1.3 Potential applications

Uncertainty quantification is important when the cost of making a wrong prediction is high, usually associated with human's safety or investment.

Autonomous vehicles such as self-driving cars can defer control to other decision making systems, including humans, if they are uncertain about their actions.

The benchmark in this work is clinical tasks using clinical time series data, thus an obvious application where uncertainty is critical is in **automated medical diagnosis**. We are interested in how reliable a diagnosis is, and in which factors are corrupting diagnosis, in order to decide if personnel intervention is necessary.

More applications are mentioned by Yarin Gal in his PhD thesis [2].

1.4 Thesis structure

We begin by identifying uncertainty, which is the predictive variance, mathematically. Then we find a way to approximate this uncertainty via a stochastic process. After identifying how to calculate uncertainty, we apply the derived approach to RNN models in common regression and classification tasks. We focus mostly on classification because the benchmark from Hrayr et al. [9] are classification tasks. Uncertainty for regression tasks will also be presented, although no signification benchmarks are provided.

Chapter 2

Related works

The following works, by other researchers, build the foundation for this thesis. Firstly, deep learning is introduced to expose the parameters. Then, prior on the parameters is proposed; the prior manifests as dropout. Finally, uncertainty is quantified as variance; from which we find a mathematical estimator for.

2.1 Deep learning

We can consider linear regression a special case of deep learning neural network, with multi-dimensional input, a single layer, and a single output. See figure 2.1. Mathematically, this is expressed as:

$$\begin{aligned} y &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \textit{where} \quad & \mathbf{x} \in \mathbf{R}^K, y \in \mathbf{R}, \mathbf{W} \in \mathbf{R}^{K\times 1}. \end{aligned} \tag{2.1}$$

The goal of a linear regression problem is to find $\mathbf{W}$ and $\mathbf{b}$ such as it minimizes the mean squared error (MSE) object function; a special case of negative log likelihood:

$$\frac{1}{N}\sum_{i=1}^{N} ||y_i - (\mathbf{W}\mathbf{x_i} + \mathbf{b})||. \tag{2.2}$$

In order to extend this concept to deep learning, we can do two (2) things:

- Allow multi-dimensional output, i.e. $\mathbf{y} \in \mathbf{R}^D$.
- Have more than one (1) linear transformation, i.e. there are many linear layers such that the output of the previous layer becomes the input for the next layer. See the following equations 2.3.

$$\begin{aligned} \mathbf{y} &= \mathbf{W_n}\mathbf{x_n} + \mathbf{b_n} \\ \mathbf{x_n} &= \mathbf{W_{n-1}}\mathbf{x_{n-1}} + \mathbf{b_{n-1}} \\ \mathbf{x_{n-1}} &= \mathbf{W_{n-2}}\mathbf{x_{n-2}} + \mathbf{b_{n-2}} \\ & \dots \\ \mathbf{x_1} &= \mathbf{W_0}\mathbf{x} + \mathbf{b_0}. \end{aligned} \tag{2.3}$$

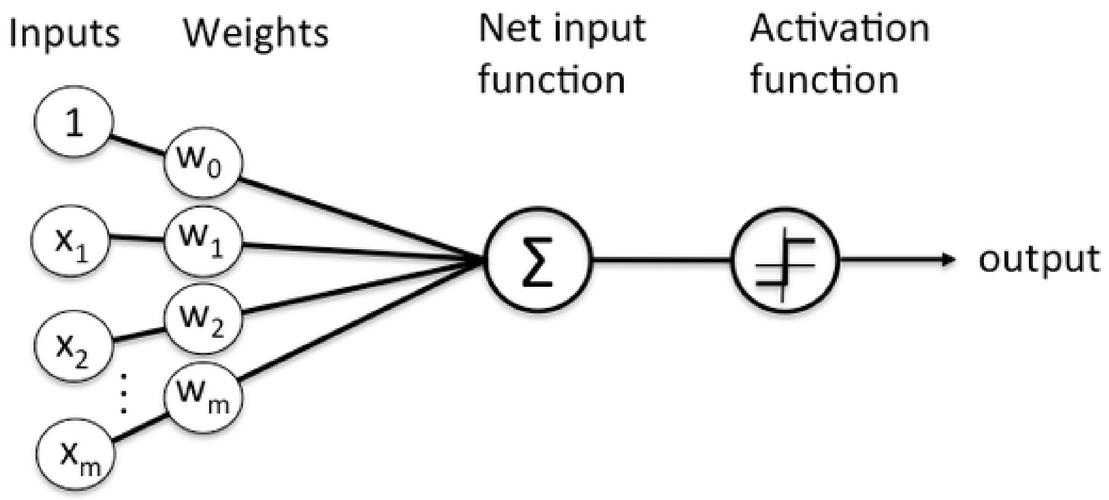

Figure 2.1: This is a linear regression model with an activation function at output. All input features are multiplied by corresponding weights, and summed up to produce an output.

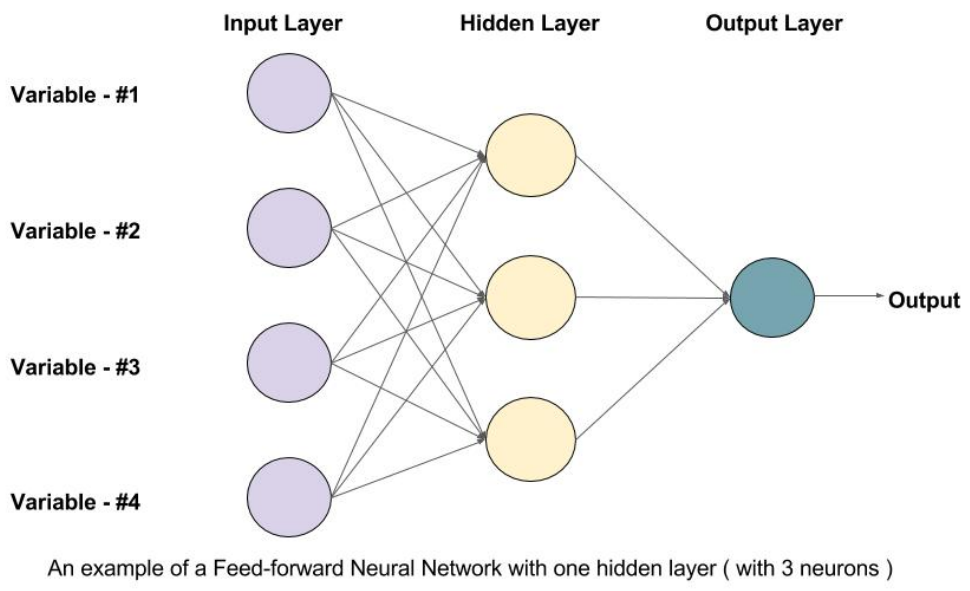

Figure 2.2: Feedforward Neural Network, Adapted from [*Learn OpenCV, Understanding Feedforward Neural Networks*]. A simple neural network can be constructed as a collection of many linear regression models, where the outputs are fed to the next model, so called *feedforward*.

However, if we only have a stack of linear transformations, then all these transformations will collapse into a single transformation $\mathbf{y} = \mathbf{W}_*\mathbf{x} + \mathbf{b}_*$. This still constitutes simple linear regression. By using activation functions (σ) at each linear transformation we introduce non-linearity to the model. We want non-linearity to capture non-linear relationships between inputs and outputs. There are different types of activation functions, e.g. ReLU, tanh, sigmoid, linear, softmax, and so on. The updated transformation is expressed in equation 2.4.

$$\begin{aligned}
\mathbf{y} &= \sigma_n(\mathbf{W_n x_n} + \mathbf{b_n}) \\
\mathbf{x_n} &= \sigma_n(\mathbf{W_{n-1} x_{n-1}} + \mathbf{b_{n-1}}) \\
\mathbf{x_{n-1}} &= \sigma_n(\mathbf{W_{n-2} x_{n-2}} + \mathbf{b_{n-2}}) \\
\dots \\
\mathbf{x_1} &= \sigma_n(\mathbf{W_0 x} + \mathbf{b_0}).
\end{aligned} \tag{2.4}$$

With this construct, we have built a simple deep learning model, aka feed-forward NN; see figure 2.2. The optimization objective is still the same, to minimize the same loss function. In practice, we use tricks such as *stochastic gradient descend* (SGD) to optimise parameters without flooding computer memory.

In classification tasks, we reformulate the model to have a softmax output, or sigmoid output for binary classification. Additionally, we change the objective function from MSE to cross-entropy $\frac{1}{N}\sum_i \sum_k p_{i,k} \log(\frac{1}{q_{i,k}})$, where p is true class, and q is predicted class probability.

Deep learning has evolved to be more complex than a simple feed-forward. In this work, we explore RNN, and specifically LSTM.

2.2 Binary dropout

Binary dropout is an act of randomly disabling some fraction of neurons in a NN layer. Mathematically, it is equivalent to setting some input dimensions to zero. It is commonly used to combat overfitting during training because it prevents models from learning to rely on a subset of input features. Randomly setting those features to zeros makes the models consider other otherwise less dominant features. Binary dropout is, conventionally, not used in inference, thus making inference models deterministic. In this work, we explore how binary dropout has a different implication; in addition to simply preventing overfitting, binary dropout is equivalent to approximating the NN weight posterior distribution as a Gaussian mixture model with one (1) component fixed at zero mean (see [5]):

$$\mathbf{W} \sim p\mathcal{N}(0, \sigma^2 \mathbf{I}_K) + (1-p)\mathcal{N}(\mathbf{M}, \sigma^2 \mathbf{I}_K). \tag{2.5}$$

In equation 2.5, a sample $\mathbf{W}$ has p chance that it is sampled from the first zero-mean Gaussian distribution; i.e. $\mathbf{W}$ is dropped out, and $(1-p)$ chance that it is sampled from the regular model weights $\mathbf{M}$.

In training phase, deactivation is applied to inputs of a layer. Coincidentally, applying random binary dropout has the same effect as randomly sampling NN kernel weights from a weight distributions (see section 3.2.1). It is useful to think of the NN not as collection of fixed weights, but as a collections of random variable weights. With this reasoning, we want to apply dropout both at training and inference time. Yarin Gal proposed Binary dropout for RNN [4], in which sampled weights are used across for all time steps.

2.3 Gaussian dropout

Gaussian dropout, and a generalized version called Variational Dropout by Kingma et al. [16], is an alternative to binary dropout, in which instead of deactivating some neurons, we scale down some features and scale up some other features. It has been shown that this dropout has the same performance but faster convergence than binary dropout. Gaussian dropout also has the benefit of being continuous (Gaussian distribution vs. Bernoulli distribution), thus making the mathematical optimization process simpler and easier to reason about. Gaussian dropout is equivalent to approximating NN weight posterior distribution as Gaussian with dropout variance

$$\mathbf{W} \sim \mathcal{N}(\mathbf{M}, \alpha \mathbf{I}_K), \tag{2.6}$$

where $\alpha = \frac{p}{1-p}$ is a dropout parameter. While Gaussian dropout seems more ideal, for the reasons mentioned above, and some debate [12] claiming that Gaussian dropout is Bayesian, this work uses binary dropout.

2.4 Concrete dropout

Concrete dropout [7] is an approximation of Bernoulli/binary dropout. Concrete dropout applies dropout masks sampled from a *ConCrete* distribution. A sample from a Bernoulli distribution is either 0 or 1. A sample from a Concrete distribution is either very close to 0, or very close to 1; within $(0, 1)$. The difference between a Concrete and a Bernoulli distribution is that in Concrete distribution, the dropout parameter is part of the sample, and the sample is continuous, whereas a sample from a Bernoulli distribution is discrete, hence the name **Con**tinuous-dis**crete**. We want to replace the conventional Bernoulli dropout with Concrete dropout because we want to also optimise dropout parameters; dropout parameters have to be a direct part of the output calculation, and continuity makes differentiation possible.

In a Bernoulli distribution, a sample is either 0 or 1, with probability p to be 1. This sample, which is not a function of p, is used as dropout mask and the dropout parameter p, is then forgotten in the output. A Concrete distribution is reformulated to use standard uniform distribution to generate stochastic a sample z

$$\begin{aligned}
&\mathbf{u} \sim Uniform(0, 1)\\
&p \in (0, 1)\\
&t \approx 0.1\\
&z = sigmoid(\frac{1}{t}(\log(p) - \log(1 - p) + \log(\mathbf{u}) - \log(1 - \mathbf{u}))).
\end{aligned} \tag{2.7}$$

Lets inspect this sample z closely. The first part is $\frac{1}{t}(\log(p) - \log(1 - p))$. This expression varies w.r.t. p and can be seen in plot 2.3. With $p = 0.5$ this expression is 0, and sigmoid produces 0.5. Increase $p > 0.5$, and we have a positive expression, magnified 10 times by temperature $t = 0.1$ alongside sigmoid close to 1. Decrease $p < 0.5$, and we have a negative expression, magnified negatively 10 times by temperature t, and sigmoid close to 0.

The second part of z introduce randomness with a standard uniform distribution. Higher p yields a higher chance of sampling $^{-}1$, and vice versa, lower p yields a higher chance of sampling 0^{+}. In this form, z is continuous, and differentiable w.r.t. p (u does not depend on p, and will disappear in derivative).

This reparameterization of z is known as the Gumbel-max trick to a softmax function [14], where we pull parameter p out of Bernoulli (discrete) distribution. Note that this is an approximation of Bernoulli distribution; there is no easy way to reparameterize a discrete distribution.

If you are interested in Gaussian dropout, reparameterization is more intuitive; we can get, not an approximate, but an exact sample. We reparameterize Gaussian distribution to standard Gaussian distribution.

$$z \sim \mathcal{N}(1, \alpha) \sim 1 + \sqrt{\alpha}\mathcal{N}(0, 1). \tag{2.8}$$

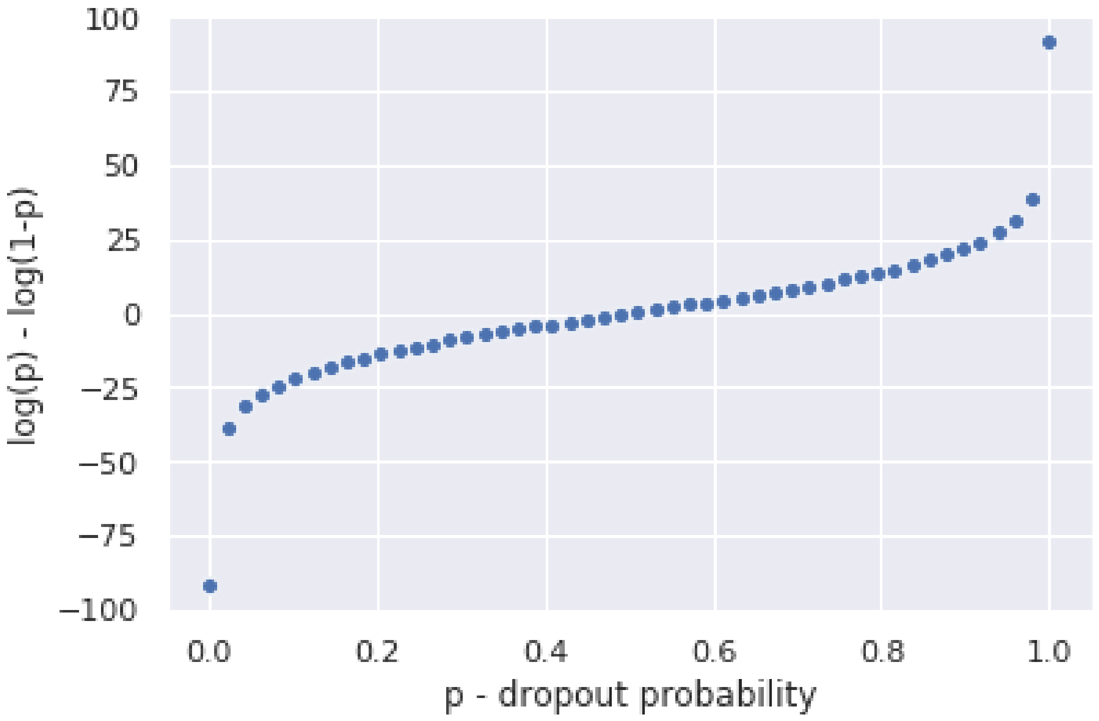

Figure 2.3: Function $\frac{1}{t}(\log(p) - \log(1-p))$ with $t = 0.1$. Notice that this graph is centered at $p = 0.5$. With larger $p > 0.5$, we have positive output, and sigmoid will squash it to 1. With smaller $p < 0.5$, we have negative output, and sigmoid will squash it to 0.

2.5 Predictive uncertainty

The focus of this paper lies in predictive uncertainty, i.e. variance in a predictive distribution

$$p(y^*|\mathbf{x}^*, \mathcal{D}) = \int_\omega p(y^*|\omega, \mathbf{x}^*) p(\omega|\mathcal{D}) d\omega. \tag{2.9}$$

We break down predictive uncertainty into aleatoric uncertainty (uncertainty from noisy data), and epistemic uncertainty (uncertainty from model and parameters). How we formulate the uncertainty depends on our assumption of the predictive distribution. Alex et al. [15] proposed a Gaussian predictive distribution, a natural choice for most **regression tasks**. We can view the traditional regression problem as predicting the mean of Gaussian output (an output is a distribution), and we now wish to find the output variance. Suppose that $p(y^*|\omega, \mathbf{x}^*) = \mathcal{N}(y^*|\mathbf{y}^\omega(\mathbf{x}^*), \sigma^\omega(\mathbf{x}^*))$, where $[\mathbf{y}^\omega(\mathbf{x}^*), \sigma^\omega(\mathbf{x}^*)] = \mathbf{f}^\omega(\mathbf{x}^*)$ is a NN output with parameters ω from posterior $p(\omega|\mathcal{D})$. The variance is given by:

$$Var[y^*] = E_{p(\omega|\mathcal{D})}[\sigma^\omega(\mathbf{x}^*)] + E_{p(\omega|\mathcal{D})}[\mathbf{y}^\omega(\mathbf{x}^*)^2] - E_{p(\omega|\mathcal{D})}[\mathbf{y}^\omega(\mathbf{x}^*)]^2. \tag{2.10}$$

Similarly, in **classification tasks**, Kwon et al. [19] proposed a Categorical predictive distribution, as a natural choice for most classification tasks. Suppose that $p(y^*|\omega, \mathbf{x}^*) = Cat(y^*|\mathbf{y}^\omega(\mathbf{x}^*))$ where $[\mathbf{y}^\omega(\mathbf{x}^*)] = softmax(\mathbf{f}^\omega(\mathbf{x}^*))$, a probability vector output from NN with parameters ω. The variance is given by

$$Var[y^*] = E_{p(\omega|\mathcal{D})}[\mathbf{y}^\omega(\mathbf{x}^*) - \mathbf{y}^\omega(\mathbf{x}^*)^2] + E_{p(\omega|\mathcal{D})}[\mathbf{y}^\omega(\mathbf{x}^*)^2] - E_{p(\omega|\mathcal{D})}[\mathbf{y}^\omega(\mathbf{x}^*)]^2. \tag{2.11}$$

Justification, explanation, and approximation for both will also be derived in section 3.3.

2.6 Variational Inference

A common approach in probabilistic models is to find a simple distribution to approximate an intractable, or difficult to optimise probability distribution, using Kullback–Leibler divergence (KL-divergence) [18]. To recap, KL-divergence is an approximate measure of distance, or similarity between 2 different distributions:

$$KL(q||p) = \int_\theta q(\theta) \log(\frac{q(\theta)}{p(\theta)}) d\theta, \tag{2.12}$$

or, for a discrete parameter θ:

$$KL(q||p) = \sum_\theta q(\theta) \log(\frac{q(\theta)}{p(\theta)}). \tag{2.13}$$

We spoke of an intractable posterior distribution $p(\omega|\mathcal{D}) = \frac{p(\mathcal{D}|\omega)p(\omega)}{p(D)}$, and an approximated distribution $q(\omega)$, which for example can be a Gaussian mixture model for binary dropout. Translated to terms of KL divergence, the goal is then to find $q(\omega)$ that is as close to $p(\omega|\mathcal{D})$ as possible.

$$
\begin{aligned}
KL(q(\omega)||p(\omega|\mathcal{D})) &= \int_\omega q(\omega)\log(\frac{q(\omega)}{p(\omega|\mathcal{D})})d\omega \\
&= \int_\omega q(\omega)\log(q(\omega))d\omega - \int_\omega q(\omega)\log(p(\omega|\mathcal{D}))d\omega \\
&= \int_\omega q(\omega)\log(q(\omega))d\omega - \int_\omega q(\omega)(\log(p(\mathcal{D}|\omega)) + \log(p(\omega)) - \log(p(D)))d\omega \\
&= \int_\omega q(\omega)\log(\frac{q(\omega)}{p(\omega)})d\omega - \int_\omega q(\omega)\log(p(\mathcal{D}|\omega))d\omega + \log(p(D))\int_\omega q(\omega)d\omega \\
&= \int_\omega q(\omega)\log(\frac{q(\omega)}{p(\omega)})d\omega - \int_\omega q(\omega)\log(p(\mathcal{D}|\omega))d\omega + \log(p(D)) \\
&= KL(q(\omega)||p(\omega)) - \int_\omega q(\omega)\log(p(\mathcal{D}|\omega))d\omega + \log(p(D)) \\
&= KL(q(\omega)||p(\omega)) + E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))] + \log(p(D)).
\end{aligned}
\tag{2.14}
$$

Our objective to minimize $KL(q(\omega)||p(\omega|\mathcal{D}))$ is equivalent to minimizing $KL(q(\omega)||p(\omega))+E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))]$, given that $\log(p(D))$ is fixed. remainder of this work is concerned with showing how to minimize the two (2) terms:

- $KL(q(\omega)||p(\omega))$
- $E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))]$ (the expectation of negative log likelihood that we are used to optimising in NN)

and which result in an optimization objective of the form:

$$\mathcal{L} = KL(q(\omega)||p(\omega)) + E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))]. \tag{2.15}$$

How we approximate these two (2) terms will also determine the way we approximate the uncertainty from the general variance formulae presented in section 2.10 and section 2.11. Notice that we are working with simpler distributions $q(\omega)$, $p(\omega)$, and $p(\mathcal{D}|\omega)$, as opposed to working with $p(\omega|\mathcal{D})$.

With the optimised parameter distribution, we can find the approximation for the predictive distribution 2.9 (see section 3.1.1).

Chapter 3

Research material and methods

3.1 Introducing probability to neural network

3.1.1 Predictive posterior distribution as output

When we talk about *probability*, we are talking about random variables; a variable that assumes values from a possible set of values, at some probability. For example, a random variable of a Bernoulli distribution with parameter $p = 0.6$ is more likely to be 1 than to be 0. In contrast, deterministic variables assume exact values. It is usually impractical to express a random variable directly if a set of possible values is large or infinite. Therefore, random variables are usually expressed with probability density functions (or probability mass functions for discrete random variables). When we want to introduce probability to NN, we want the outputs from a NN model to be random variables, rather than deterministic values. In other words, a NN model with probability is a random generator of a distribution that is conditioned on inputs $\mathbf{x}^*$, forming a predictive distribution $p(y^*|\mathbf{x}^*, \mathcal{D})$.

The question arises, why do we not consider an output distribution with regular negative log likelihood optimization approach? This is because, whilst we normally output prior predictive distribution $p(y^*|\omega, \mathbf{x}^*)$, we only get aleatoric uncertainty, which we often ignore, or consider constant; this is a side effect when we do not approach NN model in a probabilistic fashion. During optimization, we only maximize $p(\mathcal{D}|\omega)$, in other words, we find a single ω in an unbound function space that best fits the training data. Now a single set of parameters ω (we only take one sample) does not constitute a distribution. We miss out on other parameters in the same function space that also fit training data, making us overrate our confidence; or underestimate our epistemic uncertainty.

Bayesian statistics provides us with the tools to bridge the predictive distribution that we are interested in, with the data and model we have. First, Bayes's theorem expresses predictive distribution as:

$$p(y^*|\mathbf{x}^*, \mathcal{D}) = \int_\omega p(y^*|\omega, \mathbf{x}^*)p(\omega|\mathcal{D})d\omega. \tag{3.1}$$

For our purposes, instead of picking the *best* ω, we consider all possible parameter sets:

$$p(y^*|\mathbf{x}^*, \mathcal{D}) = E_{p(\omega|\mathcal{D})}[p(y^*|\omega, \mathbf{x}^*)]. \tag{3.2}$$

We cannot practically sample from $p(\omega|\mathcal{D})$ efficiently, so we use KL-divergence to find $q(\omega) \approx p(\omega|\mathcal{D})$, giving us an approximated predictive distribution:

$$\begin{aligned} q(y^*|\mathbf{x}^*) &= \int_\omega p(y^*|\omega, \mathbf{x}^*)q(\omega)d\omega \\ &= E_{q(\omega)}[p(y^*|\omega, \mathbf{x}^*)]. \end{aligned} \tag{3.3}$$

Having access to $q(\omega)$, $p(y^*|\omega, \mathbf{x}^*)$, the bridge to approximated predictive distribution is made.

Note that one might argue that we could structure predictive distribution as $p(y^*|\mathbf{x}^*, \mathcal{D}) \propto \int_\omega p(y^*|\omega, \mathbf{x}^*)p(\mathcal{D}|\omega)p(\omega)d\omega = E_{p(\omega)}[p(y^*|\omega, \mathbf{x}^*)p(\mathcal{D}|\omega)]$. While this may be possible, it is highly inefficient because each time we want to make a prediction, we have to go through training data, with a random parameter set **not** conditioned on training data.

3.1.2 Approximated posterior $q(\omega)$

It is up to us to pick an approximated distribution based on our experience. Some distributions work better for specific cases and worse for other case; for example, if we know that a parameter has value between 0 and 1, a Beta distribution is more suitable than a Gaussian distribution. In the case of binary dropout, the approximated distribution is in formula (2.5) where the variational parameter is $\mathbf{M}$. The reasoning behind this choice is that sometimes (about $p * 100\%$ of the time), we want the neuron weights to be zeros, and some other times (about $(1-p) * 100\%$ of the time), around $\mathbf{M}$ (that we will optimise), i.e.

$$\begin{aligned} q(\omega|p^* = 0) &= \mathcal{N}(0, \sigma^2\mathbf{I}) \\ q(\omega|p^* = 1) &= \mathcal{N}(\mathbf{M}, \sigma^2\mathbf{I}) \\ p(p^* = 0) &= p \\ p(p^* = 1) &= 1 - p \\ \Rightarrow q(\omega) &= p(p^* = 0)q(\omega|p^* = 0) + p(p^* = 1)q(\omega|p^* = 1) \\ &= p\mathcal{N}(0, \sigma^2\mathbf{I}) + (1-p)\mathcal{N}(\mathbf{M}, \sigma^2\mathbf{I}). \end{aligned} \tag{3.4}$$

We want the weights to be as such so that all weights have about the same level of importance, in other words, weights should complement each other so that for example, if some weights happen to be zeroed out, complementary/other weights make up for the loss of information.
The same reasoning applies to an approximating Gaussian dropout distribution $q(\omega) = \mathcal{N}(\mathbf{M}, \alpha\mathbf{I})$, where variational parameters are $\mathbf{M}$ and α.

3.1.3 Approximated Variance

The approximated posterior distribution with $q(\omega)$ gives us approximated predictive variance, for Gaussian outputs:

$$Var[y^*] = E_{q(\omega)}[\sigma^{\omega}(\mathbf{x}^*)^2] + E_{q(\omega)}[\mathbf{y}^{\omega}(\mathbf{x}^*)^2] - E_{q(\omega)}[\mathbf{y}^{\omega}(\mathbf{x}^*)]^2, \tag{3.5}$$

and for Categorical outputs:

$$Var[y^*] = E_{q(\omega)}[\mathbf{y}^{\omega}(\mathbf{x}^*) - \mathbf{y}^{\omega}(\mathbf{x}^*)^2] + E_{q(\omega)}[\mathbf{y}^{\omega}(\mathbf{x}^*)^2] - E_{q(\omega)}[\mathbf{y}^{\omega}(\mathbf{x}^*)]^2. \tag{3.6}$$

3.1.4 Optimising objective

We have established that ω samples come from posterior distribution $p(\omega|\mathcal{D})$, so it is natural that we want to optimise this posterior. We also concluded that optimising this posterior is challenging, so we approximate it with a simple form $q(\omega)$, using KL-divergence. This results in a Variational Inference (VI) objective (derived from section 2.6):

$$\mathcal{L}_{VI} = KL(q(\omega)||p(\omega)) + E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))]. \tag{3.7}$$

The first term generally represents L2 regularizers, and the second term is similar to negative log likelihood, with the distinction that we are sampling ω this time.

3.2 Monte Carlo sampling as Variational Bayesian

In this section, Monte Carlo (MC) sampling is used to approximate the optimising objective and the model outputs (means and variances).

3.2.1 Optimising objective

The optimising objective, in the form of VI loss $\mathcal{L}_{VI}$, hints that we should take several samples from variational distribution to approximate the loss. However, it does not make sense to sample ω (that have been materialized and are thus disconnected from the original distribution), to optimise loss w.r.t. $q(\omega)$. We have to instead use a reparameterization trick again to pull parameters of $q(\omega)$ out of its distribution, so that we can optimise these parameters. To achieve this, Kingma et al. [17] suggested derivative path-wise estimator and Yarin Gal [3], proposed a modification to this estimator. The modification consists of marginalizing the variational distribution as $q(\omega) = \int_{\epsilon} q(\omega|\epsilon)p(\epsilon)d\epsilon$, with $q(\omega|\epsilon) = \delta(\omega - g(\theta, \epsilon))$, where δ is a Dirac delta function, and $g(\theta, \epsilon)$ is a reparameterized transformation. See section 2.4 for examples of reparameterized distributions. The loss, reparameterized by hyper-parameters θ is then:

$$\begin{aligned}\mathcal{L}_{VI} &= KL(q(\omega)||p(\omega)) + E_{q(\omega)}[-\log(p(\mathcal{D}|\omega))] \\ &= KL(q(\omega)||p(\omega)) - \int_{\omega} q(\omega)\log(p(\mathcal{D}|\omega))d\omega \\ &= KL(q(\omega)||p(\omega)) - \int_{\epsilon} p(\epsilon)\log(p(\mathcal{D}|g(\theta, \epsilon)))d\epsilon.\end{aligned} \tag{3.8}$$

Note that we now have distribution of ϵ, which we do not need to optimise (because it does not have the parameters we want to optimise), but from which we now only need to take some samples during optimization.

According to stochastic non-convex optimisation by Rubin [21], optimising this loss approximated by MC estimation will converge to the same optima as the variational inference loss. We use MC estimation to sample ϵ once for every data pass, giving the objective function:

$$\mathcal{L}_{MC} = KL(q(\omega)||p(\omega)) - \log(p(\mathcal{D}|g(\theta, \epsilon))). \tag{3.9}$$

Specifically in our case of binary dropout approximation, we have $q(\omega) = p\mathcal{N}(0, \sigma^2\mathbf{I}) + (1-p)\mathcal{N}(\mathbf{M}, \sigma^2\mathbf{I})$. We want to find the best $\mathbf{M}$ that such that this distribution gives samples that yield the lowest loss. When we sample from this distribution, we do not have $\mathbf{M}$ in the optimization objective function we seek to optimise. By reparameterizing this as:

$$\begin{aligned} \omega &\sim p\mathcal{N}(0, \sigma^2\mathbf{I}) + (1-p)\mathcal{N}(\mathbf{M}, \sigma^2\mathbf{I}) \\ \rightarrow \omega &= z\sigma\xi + (1-z)(\mathbf{M} + \sigma\xi) \\ z &\sim Bernoulli(p) \\ \xi &\sim \mathcal{N}(0, 1). \end{aligned} \tag{3.10}$$

We extract two (2) distributions for z and ξ, which do not depend on the variational parameter $\mathbf{M}$. Now we can plug this reparameterized ω into the loss function $\mathcal{L}_{VI}$, and take one sample of z and ϵ each time loss is evaluated to arrive at $\mathcal{L}_{MC}$. In other words, $g(\theta, \epsilon) = z\sigma\xi + (1-z)(\mathbf{M} + \sigma\xi)$, where $\theta = \mathbf{M}$, and $\epsilon = \{z, \xi\}$. In theory, we should take multiple samples, but because we also run through $\mathcal{L}_{MC}$ for multiple iterations, it will converge to $\mathcal{L}_{VI}$.

Similarly, in the case of Gaussian dropout $\omega \sim \mathcal{N}(\mathbf{M}, \alpha\mathbf{I})$, we can reparameterize $\omega = \mathbf{M} + \alpha\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$. In the next section, we will derive the concrete form (not to be confused with Concrete dropout) of the objective function, including the KL term, and the negative log likelihood term for regression and classification tasks, respectively.

The KL term

$$\begin{aligned} KL(q(\omega)||p(\omega)) &= \int_\omega q(\omega)\log(q(\omega))d\omega - \int_\omega q(\omega)\log(p(\omega))d\omega \\ &= E_{q(\omega)}[\log(q(\omega))] - E_{q(\omega)}[\log(p(\omega))]. \end{aligned} \tag{3.11}$$

We shall employ the same strategy here again; reparameterize sample ω from $q(\omega)$ to have distributions free from optimising parameters, then take 1 sample to get an approximated MC estimator, yielding

$$KL(q(\omega)||p(\omega))_{MC} = \log(q(g(\theta, \epsilon))) - \log(p(g(\theta, \epsilon))), \tag{3.12}$$

where θ is the optimising variables, and ϵ are random variables, that are also free from optimising objective, making the term stochastic. We assume the prior distribution

$p(\omega)$ to be Gaussian with zero mean and variance as precision parameter, which is easy to calculate analytically. However, since the approximated distribution $q(\omega)$ is a Gaussian mixture, it is not easy to calculate analytically. Because of this, Yarin Gal [3] approximated the log (function) of the multivariate, high-dimensional Gaussian mixture distribution. Gal's method yields:

$$KL(q(\omega)||p(\omega))_{MC} \approx \sum_{l=1}^{L}(1-p_l)\frac{1}{2}||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L}\mathcal{H}(p_l) + constant, \tag{3.13}$$

where:

$$\mathcal{H}(p) = p\log(p) + (1-p)\log(1-p), \tag{3.14}$$

and L is the number of stochastic layers of our NN model, for example $q(\omega) = \prod_{l=1}^{L} q(\omega_l)$. It should be simpler to calculate this KL term when using Gaussian dropout because it has only one (1) Gaussian term. Therefore, if you want to use Gaussian dropout, this is where you should update the objective function. The resulting MC loss is:

$$\mathcal{L}_{MC} \propto \sum_{l=1}^{L}(1-p_l)\frac{1}{2}||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L}\mathcal{H}(p_l) - \log(p(\mathcal{D}|g(\theta,\epsilon))). \tag{3.15}$$

Negative log likelihood for Regression

The predictive outputs have Gaussian distribution, thus:

$$\begin{gathered} p(\mathcal{D}|g(\theta,\epsilon)) = \prod_{i=1}^{N} p(y_i|g(\theta,\epsilon),\mathbf{x}_i) = \prod_{i=1}^{N}\mathbf{N}(y_i|\hat{y}_i,\sigma_i^2) \\ s.t. \quad [\hat{y}_i,\sigma_i^2] = \mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i). \end{gathered} \tag{3.16}$$

To make this more numerically stable, we output the log scale of predicted variance (see equation 3.17):

$$\begin{aligned} \hat{y}_i, s_i &= \mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i) \\ s_i &= \log(\sigma_i^2) \\ p(\mathcal{D}|g(\theta,\epsilon)) &= \prod_{i=1}^{N}\mathbf{N}(y_i|\hat{y}_i,e^{s_i}) \\ \log(p(\mathcal{D}|g(\theta,\epsilon))) &= \sum_{i=1}^{N}\log\mathbf{N}(y_i|\hat{y}_i,e^{s_i}) \\ &= \sum_{i=1}^{N}\frac{1}{2}(-\log(2\pi) - \log(e^{s_i}) - \frac{||y_i-\hat{y}_i||^2}{e^{s_i}}) \\ &\propto -\frac{1}{2}\sum_{i=1}^{N}(s_i + \exp(-s_i)||y_i-\hat{y}_i||^2). \end{aligned} \tag{3.17}$$

Note that our NN model $\mathbf{f}^{g(\theta,\epsilon)}(\cdot)$ outputs both mean and variance, due to the fact that we are interested in predicting variance instead of assuming a fixed variance for all training data. Also note that we do not have actual variance values in the objective function. Instead, this objective function encourages small variance. This suggests that a probabilistic model learns to predict outputs (as random variables) with small variance. Putting everything together we get an objective function for regression tasks with stochastic binary dropout as:

$$\mathcal{L}_{MC} = \sum_{l=1}^{L}(1-p_l)\frac{1}{2}||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L}\mathcal{H}(p_l) + \frac{1}{2}\sum_{i=1}^{N}(s_i + \exp(-s_i)||y_i - \hat{y}_i||^2). \tag{3.18}$$

In this objective function, no consideration is given to non-stochastic layers (the usual layers in NN), but if regularizing those non-stochastic layers is necessary, you can add their weight magnitudes to this objective function as well. Additionally, we can put some ratio coefficients to control the importance of each term. As a general practice, we usually divide the objective loss by the number of training instances.

Negative log likelihood for Classification

This time, the outputs have Categorical distribution, thus:

$$\begin{aligned}
p(\mathcal{D}|g(\theta,\epsilon)) &= \prod_{i=1}^{N} p(y_i|g(\theta,\epsilon),\mathbf{x}_i) \\
&= \prod_{i=1}^{N} Cat(y_i|\mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i)) \\
\log(p(\mathcal{D}|g(\theta,\epsilon))) &= \sum_{i=1}^{N}\log(Cat(y_i|\mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i))) \qquad (3.19)\\
&= \sum_{i=1}^{N}\log(\prod_{j=1}^{M}\mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i)_j^{y_{i,j}}) \\
&= \sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(\mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i)_j),
\end{aligned}$$

where N is the number of training instances, and M is the number of categories. In the case of binary classification, we can output just a sigmoid output, and infer the negative category by the complement. This loss happens to be exactly the same as cross-entropy loss that we usually use in classification tasks. The reason why we have extra variance output in a regression case is we no longer assume variance to be static; if variance were to be static, the negative log likelihood in regression would be similar to mean-squared-error (MSE). Putting everything together, we arrive at an objective function for classification task with stochastic binary dropout as follows:

$$\mathcal{L}_{MC} = \sum_{l=1}^{L}(1-p_l)\frac{1}{2}||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L}\mathcal{H}(p_l) - \sum_{j=1}^{M} y_{i,j}\log(\mathbf{f}^{g(\theta,\epsilon)}(\mathbf{x}_i)_j), \tag{3.20}$$

where y_i is a one-hot probability vector.

3.2.2 Concrete Dropout

Initially, we perform dropout with predefined dropout rates. Dropout rate is a stochastic parameter, thus it influences model uncertainty, or epistemic uncertainty. We notice that the dropout rate is also part of the parameters to generate a weight sample, thus we can treat dropout rate as a variational parameter for optimization, so that in the approximation distribution:

$$q(\omega) = p\mathcal{N}(0, \sigma^2 \mathbf{I}_K) + (1-p)\mathcal{N}(\mathbf{M}, \sigma^2 \mathbf{I}_K), \tag{3.21}$$

where we have p as variational parameter to optimise, in addition to $\mathbf{M}$.

Inspecting our MC objective function 3.18, we already have the parameter p (shaded in the following equation), which would have been discarded during differentiation because it would have been considered as predefined and constant.

$$\mathcal{L}_{MC} = \sum_{l=1}^{L} \colorbox{gray}{$(1-p_l)$} \frac{1}{2}||\mathbf{M_l}||_2^2 + \colorbox{gray}{$\displaystyle\sum_{l=1}^{L} \mathcal{H}(p_l)$} - \log(p(\mathcal{D}|g(\theta, \epsilon))). \tag{3.22}$$

It is tempting to just blindly consider dropout parameters as network parameters that can be optimised during training. But, by doing so, we would ignore the relationship between predictions and dropout parameters in the negative log likelihood term, the output $\hat{y}_i$ would then consist of input $\mathbf{x}_i$, and network weights generated by variational parameters. Gal et al. [7] used reparameterization trick to pull dropout parameters out of their distributions in order to become part of outputs. Binary dropout, sampled from Bernoulli distribution, is transformed into Concrete dropout by following the steps described in section 2.4. The Concrete dropout approach uses a discrete quantised Gaussian prior for $p(\omega)$ instead of standard Gaussian, to make the KL-term $KL(q_\theta(\omega)||p(\omega))$ tractable. In conclusion, we arrive at the following objective function 3.23 that incorporates dropout parameters:

$$\mathcal{L}_{MC} = \sum_{l=1}^{L} \colorbox{gray}{$(1-p_l)$} \frac{1}{2}||\mathbf{M_l}||_2^2 + \colorbox{gray}{$\displaystyle\sum_{l=1}^{L} K_l \mathcal{H}(p_l)$} - \log(p(\mathcal{D}|g(\theta, \epsilon))), \tag{3.23}$$

where K_i is the number of input dimensions of layer i. Dropout masks are then sampled from a Concrete distribution using equation 2.7.

3.2.3 Output and uncertainty

From formula 3.3, we are interested in the mean and the variance of the approximated predictive distribution. See equation block below.

$$q(y^*|\mathbf{x}^*) = E_{q(\omega)}[p(y^*|\omega, \mathbf{x}^*)]. \tag{3.24}$$

Parameters for approximated distribution $q(\omega)$ (from which the NN model weights are stochastically sampled), should be optimised after training. One way of doing this is by solving the integral to find the true predictive posterior distribution.

Alternatively, we can estimate this predictive posterior by sampling the means and variances T times using T ω samples from $q(\omega)$, for example, we can use $q(y^*|\mathbf{x}^*) \approx \frac{1}{T}\sum_{t=1}^{T} p(y^*|\omega_t, \mathbf{x}^*), \omega_t \sim q(\omega)$, an average of several approximated distributions.

Equality happens when $T \to \infty$. Further, $p(y^*|\omega, \mathbf{x}^*)$ should take on the form of your output (usually Gaussian for regression, or Categorical for classification). Our NN model $\mathbf{f}$ outputs the parameters for these distributions[1]. We arrive at the approximated predictive mean by averaging the sampled means, and predictive variance by the *definition of variance*. See derivations below.

Regression output and uncertainty

For regression tasks, the standard forms for predictive mean and variance, found in equation block (3.5), take the following form:

$$\begin{aligned}
\omega_t &\sim q(\omega) \\
[y_t^*, s_t] &= \mathbf{f}^{\omega_t}(\mathbf{x}^*) \\
\sigma_t^2 &= \exp(s_t) \\
y_{out} = E_{q(y^*|\mathbf{x}^*)}[y^*] &= \frac{1}{T}\sum_{t=1}^{T} y_t^* \\
uncertainty = Var_{q(y^*|\mathbf{x}^*)}[y^*] &= E[\sigma_t^2] + E[(y_t^*)^2] - E[y_t^*]^2 \\
&= \frac{1}{T}\sum_{t=1}^{T}\exp(s_t) + \frac{1}{T}\sum_{t=1}^{T}(y_t^*)^2 - y_{out}^2 \\
&= \frac{1}{T}\sum_{t=1}^{T}\exp(s_t) + Var_{q(y^*|\omega,\mathbf{x}^*)}[y_t^*].
\end{aligned} \tag{3.25}$$

Classification output and uncertainty

For classification tasks, the standard forms for predictive mean probability vector and variance, found in equation block (3.6), take the following form:

$$\begin{aligned}
\omega_t &\sim q(\omega) \\
\mathbf{p}_t^* &= \mathbf{f}^{\omega_t}(\mathbf{x}^*) \\
\mathbf{p}_{out} = E_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*] &= \frac{1}{T}\sum_{t=1}^{T}\mathbf{p}_t^* \\
uncertainty = Var_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*] &= E[\mathbf{p}_t^* - (\mathbf{p}_t^*)^2] + E[(\mathbf{p}_t^*)^2] - E[\mathbf{p}_t^*]^2 \\
&= \frac{1}{T}\sum_{t=1}^{T}(\mathbf{p}_t^* - (\mathbf{p}_t^*)^2) + \frac{1}{T}\sum_{t=1}^{T}(\mathbf{p}_t^*)^2 - \mathbf{p}_{out}^2 \\
&= \frac{1}{T}\sum_{t=1}^{T}\mathbf{p}_t^*(1-\mathbf{p}_t^*) + Var_{q(\mathbf{p}^*|\omega,\mathbf{x}^*)}[\mathbf{p}_t^*].
\end{aligned} \tag{3.26}$$

[1]We could argue that we have to marginalize over model space for predictive distributions, which means we will end up with an ensemble model (a collection of several NN models), but for practical reasons, we only condition on 1 model $\mathbf{f}$ at a time.

Observe that we denote output probability vector as $\mathbf{p}$ instead of y here, to signify that this is a softmax output.

In both regression and classification cases, the proof and full derivation for predictive mean and variance can be found in section 3.3.

3.3 Why the VB-MC process gives approximated predictive uncertainty

First, we should acknowledge that VB renders an estimate for the predictive posterior, $q(y^*|\mathbf{x}^*) = \int_\omega p(y^*|\omega, \mathbf{x}^*)q(\omega)d\omega$. MC then, rather than evaluate the predictive posterior analytically, helps approximate it, $q(y^*|\mathbf{x}^*) \approx \frac{1}{T}\sum_{t=1}^{T} p(y^*|\omega_t, \mathbf{x}^*)$. Because we evaluate predictions at all ω samples from $q(\omega)$, (what we term *a dropout approximation*) it should be obvious that dropout is performed at inference as well. Next, we shall derive the approximated predictive mean and predictive variance that we have mentioned in equation 3.25 and 3.26.

3.3.1 Regression

Recall from equation 3.25, we have NN outputs Gaussian random variable parameters:

$$[\hat{y}_t^*, \sigma_t^2] = \left[E_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*], Var_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*]\right] = \mathbf{f}^{\omega_t}(\mathbf{x}^*). \tag{3.27}$$

Given these characteristics, ***predictive mean*** and ***predictive variance*** are expressed as:

Predictive mean

$$\begin{aligned}
E_{q(y^*|\mathbf{x}^*)}[y^*] &= \int_{y^*} y^* q(y^*|\mathbf{x}^*)dy^* \\
&= \int_{y^*}\int_\omega y^* p(y^*|\omega, \mathbf{x}^*)q(\omega)d\omega dy^* \\
&= \int_\omega\int_{y^*} y^* p(y^*|\omega, \mathbf{x}^*)dy^* q(\omega)d\omega \\
&= \int_\omega E_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*]q(\omega)d\omega \\
&= \int_\omega \hat{y}_t^* q(\omega)d\omega \\
&= E_{q(\omega)}[\hat{y}_t^*] \approx \frac{1}{T}\sum_{t=1}^{T}\hat{y}_t^*
\end{aligned} \tag{3.28}$$

Predictive variance

$$
\begin{aligned}
Var_{q(y^*|\mathbf{x}^*)}[y^*] &= E_{q(y^*|\mathbf{x}^*)}[(y^*)^2] - E_{q(y^*|\mathbf{x}^*)}[y^*]^2 \\
E_{q(y^*|\mathbf{x}^*)}[(y^*)^2] &= \int_{y^*} (y^*)^2 q(y^*|\mathbf{x}^*) dy^* \\
&= \int_{y^*} (y^*)^2 \int_{\omega} p(y^*|\omega, \mathbf{x}^*) q(\omega) d\omega dy^* \\
&= \int_{\omega} \int_{y^*} (y^*)^2 p(y^*|\omega, \mathbf{x}^*) dy^* q(\omega) d\omega \\
&= \int_{\omega} E_{p(y^*|\omega,\mathbf{x}^*)}[(y_t^*)^2] q(\omega) d\omega \\
&= \int_{\omega} (Var_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*] + E_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*]^2) q(\omega) d\omega \\
&= \int_{\omega} Var_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*] q(\omega) d\omega + \int_{\omega} E_{p(y^*|\omega,\mathbf{x}^*)}[y_t^*]^2 q(\omega) d\omega \qquad (3.29) \\
&= \int_{\omega} \sigma_t^2 q(\omega) d\omega + \int_{\omega} (\hat{y}_t^*)^2 q(\omega) d\omega \\
&= E_{q(\omega)}[\sigma_t^2] + E_{q(\omega)}[(\hat{y}_t^*)^2] \\
&\approx \frac{1}{T} \sum_{t=1}^{T} \sigma_t^2 + \frac{1}{T} \sum_{t=1}^{T} (\hat{y}_t^*)^2 \\
E_{q(y^*|\mathbf{x}^*)}[y^*]^2 &\approx (\frac{1}{T} \sum_{t=1}^{T} \hat{y}_t^*)^2 \\
\Rightarrow Var_{q(y^*|\mathbf{x}^*)}[y^*] &\approx \frac{1}{T} \sum_{t=1}^{T} \sigma_t^2 + \frac{1}{T} \sum_{t=1}^{T} (\hat{y}_t^*)^2 - (\frac{1}{T} \sum_{t=1}^{T} \hat{y}_t^*)^2
\end{aligned}
$$

3.3.2 Classification

Recall from equation 3.26, we have NN outputs Categorical random variable parameters:

$$\hat{\mathbf{p}}_t^* = E_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[\mathbf{p}_t^*] = \mathbf{f}^{\omega_t}(\mathbf{x}^*). \qquad (3.30)$$

Given these characteristics, ***predictive mean*** and ***predictive variance*** are expressed as:

Predictive mean

$$\begin{aligned}
E_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*] &= \int_{\mathbf{p}^*} \mathbf{p}^* q(\mathbf{p}^*|\mathbf{x}^*) d\mathbf{p}^* \\
&= \int_{\mathbf{p}^*} \mathbf{p}^* \int_{\omega} p(\mathbf{p}^*|\omega, \mathbf{x}^*) q(\omega) d\omega d\mathbf{p}^* \\
&= \int_{\omega} \int_{\mathbf{p}^*} \mathbf{p}^* p(\mathbf{p}^*|\omega, \mathbf{x}^*) d\mathbf{p}^* q(\omega) d\omega \\
&= \int_{\omega} E_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[\mathbf{p}_t^*] q(\omega) d\omega \\
&= \int_{\omega} \hat{\mathbf{p}}_t^* q(\omega) d\omega \\
&= E_{q(\omega)}[\hat{\mathbf{p}}_t^*] \approx \frac{1}{T} \sum_{t=1}^{T} \hat{\mathbf{p}}_t^*
\end{aligned} \tag{3.31}$$

Predictive variance

$$\begin{aligned}
Var_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*] &= E_{q(\mathbf{p}^*|\mathbf{x}^*)}[(\mathbf{p}^*)^2] - E_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*]^2 \\
E_{q(\mathbf{p}^*|\mathbf{x}^*)}[(\mathbf{p}^*)^2] &= \int_{\mathbf{p}^*} (\mathbf{p}^*)^2 q(\mathbf{p}^*|\mathbf{x}^*) d\mathbf{p}^* \\
&= \int_{\mathbf{p}^*} (\mathbf{p}^*)^2 \int_{\omega} p(\mathbf{p}^*|\omega, \mathbf{x}^*) q(\omega) d\omega d\mathbf{p}^* \\
&= \int_{\omega} \int_{\mathbf{p}^*} (\mathbf{p}^*)^2 p(\mathbf{p}^*|\omega, \mathbf{x}^*) d\mathbf{p}^* q(\omega) d\omega \\
&= \int_{\omega} E_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[(\mathbf{p}_t^*)^2] q(\omega) d\omega \\
&= \int_{\omega} (Var_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[(\mathbf{p}_t^*)] + E_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[(\mathbf{p}_t^*)]^2) q(\omega) d\omega \\
&= \int_{\omega} Var_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[(\mathbf{p}_t^*)] q(\omega) d\omega + \int_{\omega} E_{p(\mathbf{p}^*|\omega,\mathbf{x}^*)}[(\mathbf{p}_t^*)]^2 q(\omega) d\omega \\
&= \int_{\omega} \hat{\mathbf{p}}_t^* (1 - \hat{\mathbf{p}}_t^*) q(\omega) d\omega + \int_{\omega} (\hat{\mathbf{p}}_t^*)^2 q(\omega) d\omega \\
&= E_{q(\omega)}[\hat{\mathbf{p}}_t^* (1 - \hat{\mathbf{p}}_t^*)] + E_{q(\omega)}[(\hat{\mathbf{p}}_t^*)^2] \\
&\approx \frac{1}{T} \sum_{t=1}^{T} \hat{\mathbf{p}}_t^* (1 - \hat{\mathbf{p}}_t^*) + \frac{1}{T} \sum_{t=1}^{T} (\hat{\mathbf{p}}_t^*)^2 \\
E_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*]^2 &\approx (\frac{1}{T} \sum_{t=1}^{T} \hat{\mathbf{p}}_t^*)^2 \\
\Rightarrow Var_{q(\mathbf{p}^*|\mathbf{x}^*)}[\mathbf{p}^*] &\approx \frac{1}{T} \sum_{t=1}^{T} \hat{\mathbf{p}}_t^* (1 - \hat{\mathbf{p}}_t^*) + \frac{1}{T} \sum_{t=1}^{T} (\hat{\mathbf{p}}_t^*)^2 - (\frac{1}{T} \sum_{t=1}^{T} \hat{\mathbf{p}}_t^*)^2
\end{aligned} \tag{3.32}$$

3.3.3 Uncertainty types

Earlier we mention *aleatoric uncertainty* and *epistemic uncertainty* (see section 1.1). From our derivation of predictive variance, we notice that it has a general form:

$$\begin{aligned} Var_{q(y^*|\mathbf{x}^*)}[y^*] =& E_{q(\omega)}[Var_{p(y^*|\omega,\mathbf{x}^*)}[y^*]] + E_{q(\omega)}[E_{p(y^*|\omega,\mathbf{x}^*)}[y^*]^2] \\ & - E_{q(\omega)}[E_{p(y^*|\omega,\mathbf{x}^*)}[y^*]]^2 \\ =& E_{q(\omega)}[Var_{p(y^*|\omega,\mathbf{x}^*)}[y^*]] + Var_{q(\omega)}[E_{p(y^*|\omega,\mathbf{x}^*)}[y^*]]. \end{aligned} \tag{3.33}$$

The first term is an average of NN output variance, or **aleatoric uncertainty**. The second term is variance of model prediction, in other words, how predictions vary under changes of parameters, or **epistemic uncertainty**. It is a good idea to separate aleatoric and epistemic uncertainty in calculations, to narrow down sources of uncertainty and prepare appropriate plans to improve the model. For example, high aleatoric uncertainty means the input is noisy. Appropriate remedies may include better preprocessing of data, using different data sources/replacing the input. The input may also be simply wrong or irrelevant for the task of a model (input is out of distribution). If the epistemic uncertainty is high, perhaps more data should be used to train the model.
An analogy to help understand aleatoric and epistemic uncertainty better is, suppose you have an upcoming examination, and you have a list of materials to study from. If you study thoroughly, and the questions on the test are related to your study materials, then you can complete the exam with high confidence, and with good result. If you study half of the materials, you feel more uncertain during the exam in general, you are not certain about your answers, you as a learner are expressing some amount of epistemic uncertainty. If you study the material thoroughly, and yet there is a strange question unlike anything you have seen before, you feel less confident about your answer to that question compared to other questions. What you experience is aleatoric uncertainty, an uncertainty when encountering an out-of-distribution question.

3.4 Challenges in Bayesian dropout in LSTM

So far we have not discussed the stochastic process in NN weights, and yet we have derived mathematical expressions for the objective function, predictive mean, and predictive variance (uncertainty). This means that we have a lot of freedom in designing our stochastic NN layer; e.g. deciding that only a subset of weights be dropped out, to dropping out all weights. We are still limited to using binary dropout, dictated by our approximated posterior $q(\omega)$ to be Gaussian mixture with zero mean at one cluster and normal weights at another. Should you decide to use another stochastic process, you should first reevaluate the KL-divergence loss first with a corresponding approximated distribution.

Dropout in a non-recurrent layer is straightforward, every neuron[2] is deactivated at probability p. In the case of RNN, neurons are not deactivated during recurrent

[2]A neuron should interact with all features in input, thus deactivating that neuron is equivalent to zeroing output of that neuron. Dropout is then performed at neuron output.

steps; if we apply dropout at layer output, the neurons are deactivated only in the last recurrent step. And, if all weights can be dropped, we should also apply dropout to the hidden state. Another way to perform dropout, that still conforms with our Gaussian mixture posterior approximation, is by deactivating weights individually.

3.4.1 How to properly perform dropout

In a simple feedforward layer, dropout is performed at neuron level, so that output from that neuron is zero. A layer is represented mathematically as a matrix, where a neuron is equivalent to a column. If a neuron is deactivated, we set a corresponding column to zero. A forward pass has expression $\mathbf{y} = \mathbf{x}\mathbf{A}$. This is an example where we deactivate the first column and third column as illustrated in figure 3.1:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{0} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{0} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & A_{1,2} & 0 & A_{1,4} \\ 0 & A_{2,2} & 0 & A_{2,4} \\ 0 & A_{3,2} & 0 & A_{3,4} \end{bmatrix}. \tag{3.34}$$

The mask matrix is similar to an identity matrix, except that at the row where we want to deactivate, we set all entries to zeros. The feedforward expression with dropout is then $\mathbf{y} = \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{M_{dropout}}$.

Building on these concepts, we can now proceed to any RNN. In RNN, every weight matrix (both input weight matrix and hidden weight matrix), is multiplied with a dropout mask, where the dropout mask has been sampled once for each forward pass, meaning one dropout mask sample for all timesteps. A mask can be generated by sampling from a multivariate[3] Bernoulli distribution with $(1-p)$ probability, where the resulting binary vector is transformed into a diagonal matrix:

$$\begin{gathered} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \sim Ber(1-p)^4 \\ diag(\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \end{gathered} \tag{3.35}$$

The presented approach approach is good because, when a neuron is deactivated, other neurons still consider all input features. Further, if there is only one (1) input feature, we do not run the risk of ignoring input. The drawback of this approach is that neurons may still learn to favour some input features over others. Extending this dropout strategy to built-in RNN layers in Deep learning libraries remains difficult; likely, we will have to reimplement such RNN layers from the beginning.

[3]dimension as number of hidden units

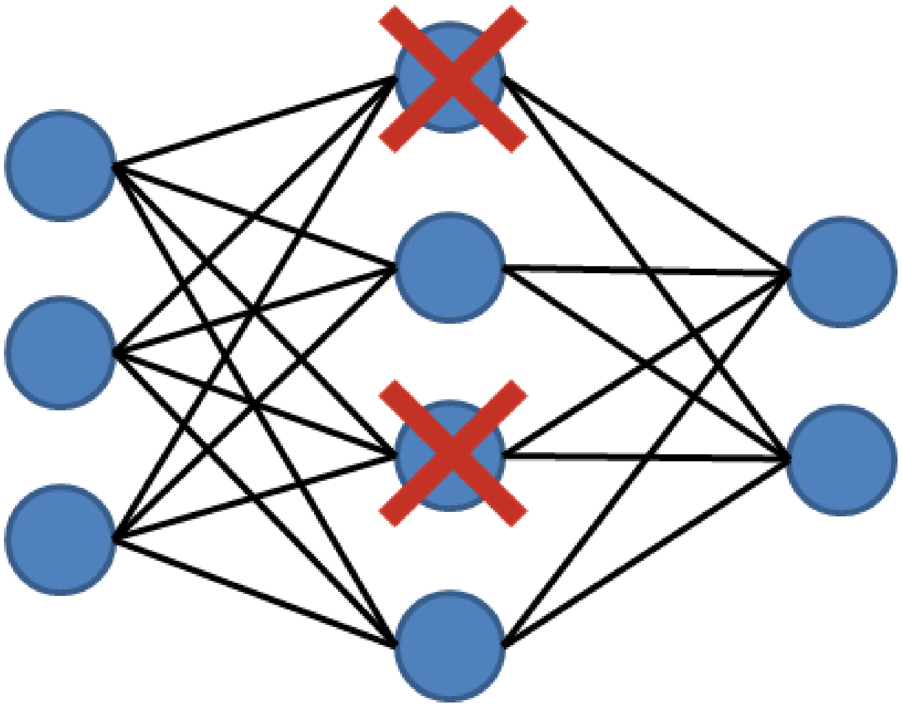

Figure 3.1: Feedforward dropout. In this illustration, a layer is a vertical slice of neurons.

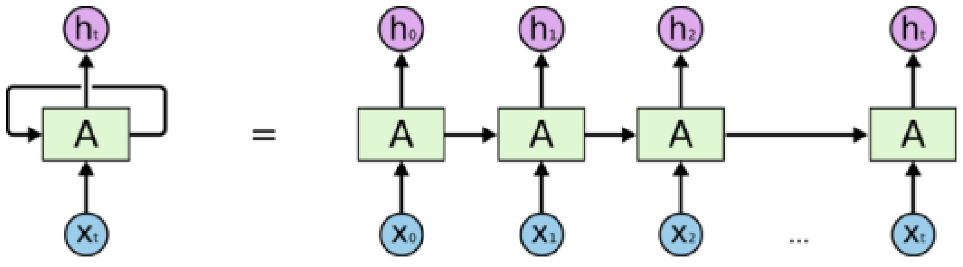

Figure 3.2: A simple architecture of an unrolled RNN. Adapted from [*Towards Data Science, Understanding RNN and LSTM*]. Suppose we have input as a sequence of **t** samples, the first sample is assumed to have hidden state zero, the output from the first sample becomes the hidden state of the second sample, and so on.

RNN dropout in practice

Yarin Gal [4] proposed applying dropout to the inputs instead, or mathematically, $\mathbf{y} = \mathbf{x} \circ \mathbf{z_{dropout}} \cdot \mathbf{A}$. The drawback of this strategy presents when there is only a single input dimension; for this case, we may lose the only available input information. We must handle this case programmatically, to prevent such dropout. An advantage of the method is, it is easy to extend from built-in RNN layers in Keras, since it comes with a naive dropout-on-input implementation. Another advantage of Yarin's approach is that neurons, eventually learn to consider all input features. Some neurons might learn to have less impact (very small weight values compared to other neurons), which should be interpreted as having more neurons than necessary, not a problem per se. Because the advantages of applying dropout at input outweight the drawbacks (we are after all, not working with a single input dimension), we chose to use this dropout implementation in this work.

3.4.2 Long-short-term-memory (LSTM)

LSTM is a form of RNN [13] adapted to handle temporal data, i.e. data with a sequential structure, such as audio samples, where a single sample does not mean anything, but a sequence of audio samples can make up a song, or a conversation. Note that the time order is important in this kind of data. An RNN has a hidden state that retains some characteristic of previous input; the idea is to make use of this property to capture temporal relationship between samples in a sequence.

$$\begin{aligned}
\mathbf{h_0} &= \sigma(\mathbf{0W_h} + \mathbf{x_0W_x} + \mathbf{b}) \\
\mathbf{h_1} &= \sigma(\mathbf{h_0W_h} + \mathbf{x_1W_x} + \mathbf{b}) \\
&\cdots \\
\mathbf{h_t} &= \sigma(\mathbf{h_{t-1}W_h} + \mathbf{x_tW_x} + \mathbf{b}) \\
output &= \mathbf{h_T}.
\end{aligned} \tag{3.36}$$

While simple RNN seems ideal to capture temporal relationships, it does not perform well with long sequences. This is because the derivative during optimization diminishes the gradient, due to multiplications between many small values, or many large values. This phenomenon is known as *Vanishing gradient problem* [10]. LSTM [11] is one solution to address this problem; by selectively choosing which hidden features to remember, and which hidden features to forget. A typical mathematical representation of LSTM is found in equation 3.37 (note that there are many variants of LSTM):

$$\begin{aligned}
\mathbf{f}_t &= sigmoid(\mathbf{x}_t\mathbf{W_f} + \mathbf{h}_{t-1}\mathbf{U_f} + \mathbf{b_f}) \\
\mathbf{i}_t &= sigmoid(\mathbf{x}_t\mathbf{W_i} + \mathbf{h}_{t-1}\mathbf{U_i} + \mathbf{b_i}) \\
\mathbf{o}_t &= sigmoid(\mathbf{x}_t\mathbf{W_o} + \mathbf{h}_{t-1}\mathbf{U_o} + \mathbf{b_o}) \\
\mathbf{g}_t &= tanh(\mathbf{x}_t\mathbf{W_g} + \mathbf{h}_{t-1}\mathbf{U_g} + \mathbf{b_g}) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \circ tanh(\mathbf{c}_t).
\end{aligned} \tag{3.37}$$

These operations form a computation for one (1) timestep, illustrated in figure 3.3. The initial values for cell state and hidden state are zeros. In equation 3.37, $\mathbf{f}_t$ computes which features to forget (notice the use of sigmoid to squash the values between 0 and 1) in $\mathbf{c}_{t-1}$. $\mathbf{i}_t$ projects new input to the cell state. $\mathbf{g}_t$ selects which features to forget, or retain from projected input. With these ideas, the cell state $\mathbf{c}_t$ is updated accordingly, with selected values from previous cell state and new input. $\mathbf{o}_t$ selects feature from updated cell state as output and new hidden state. By being selective in which features to retain in cell state and hidden state, LSTM is suited to handle longer sequences.

3.4.3 Dropout in LSTM

Additionally, we also use the approach from the work of Gal et al. [4] to perform dropout for LSTM. At every timestep, illustrated in equation 3.37, we apply dropout to input and hidden state, for every weight matrix. We have 8 such weight matrices, thus we need to sample 8 dropout masks (4 for input, and 4 for hidden state). During implementation, we should also account for batched input. If we do not want all inputs in a batch to have the same dropout masks, we must sample dropout masks for every input in a batch as well. A timestep in LSTM is then updated as:

$$\begin{aligned}
\mathbf{f}_t &= sigmoid(\mathbf{x}_t \circ \mathbf{z}_{W,f}\mathbf{W_f} + \mathbf{h}_{t-1} \circ \mathbf{z}_{U,f}\mathbf{U_f} + \mathbf{b_f}) \\
\mathbf{i}_t &= sigmoid(\mathbf{x}_t \circ \mathbf{z}_{W,i}\mathbf{W_i} + \mathbf{h}_{t-1} \circ \mathbf{z}_{U,i}\mathbf{U_i} + \mathbf{b_i}) \\
\mathbf{o}_t &= sigmoid(\mathbf{x}_t \circ \mathbf{z}_{W,o}\mathbf{W_o} + \mathbf{h}_{t-1} \circ \mathbf{z}_{U,o}\mathbf{U_o} + \mathbf{b_o}) \\
\mathbf{g}_t &= tanh(\mathbf{x}_t \circ \mathbf{z}_{W,g}\mathbf{W_g} + \mathbf{h}_{t-1} \circ \mathbf{z}_{U,g}\mathbf{U_g} + \mathbf{b_g}) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \circ tanh(\mathbf{c}_t),
\end{aligned} \tag{3.38}$$

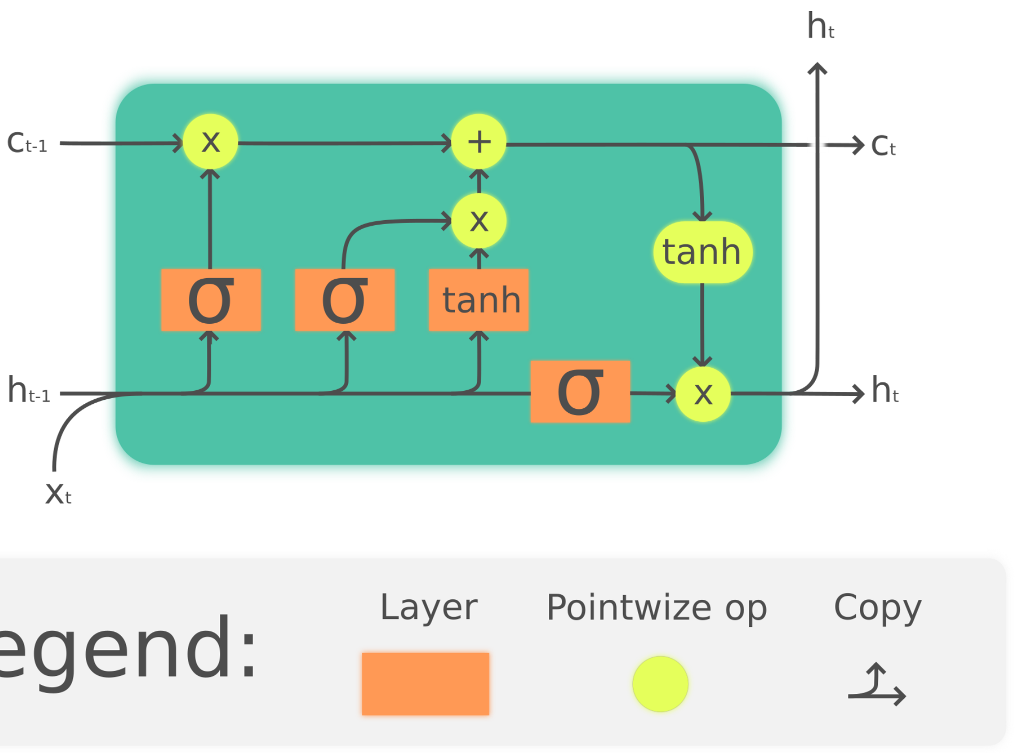

Figure 3.3: A typical LSTM diagram with forget gate [11]. Adapted from [*Wikipedia, Long short-term memory*]. The 4 orange squares correspond to the first 4 operations from equation group 3.37. Dropout is applied to x_t and h_{t-1} at every orange box. Dropout masks can be different in each box. We keep the same dropout masks at all timesteps in a single forward pass. This is further explained in chapter 3.4.

or equivalently from [4]:

$$\begin{pmatrix} \underline{\mathbf{i}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{o}} \\ \underline{\mathbf{g}} \end{pmatrix} = \begin{pmatrix} sigmoid \\ sigmoid \\ sigmoid \\ tanh \end{pmatrix} \left(\begin{pmatrix} \mathbf{x}_t \circ \mathbf{z}_x \\ \mathbf{h}_{t-1} \circ \mathbf{z}_h \end{pmatrix} \cdot \mathbf{W} \right). \tag{3.39}$$

There are other variants mentioned in the same paper, but we choose this approach because it performs dropout on all LSTM weights.

Extend to other RNN and non-RNN network types

From the dropout scheme with LSTM above, we see a general rule: whenever there is an interaction between a weight matrix, and an input or hidden state vector, we sample a mask for that input or hidden state vector, and that mask is tied to a weight matrix. In RNN, we have to remember to use the same mask at every timestep. Conversely, in non-RNN, we can drop and forget.

3.5 Implementation of stochastic LSTM layer

In our implementation, we choose to use Concrete distribution, described in section 3.2.2 to sample dropout masks for both a) optimising dropout rates, and b) specifying dropout rates. This should unify the mask sampling implementation. In general, whenever we apply dropout, we scale results by a factor $\frac{1}{1-p}$ to boost the remaining features. We reparameterize this scaling factor as part of weight variational parameter, that is, $\mathbf{M} \to \frac{\mathbf{M}}{1-p}$. Our loss function becomes:

$$\begin{aligned} \mathcal{L}_{MC} &= \sum_{l=1}^{L} (1-p_l) \frac{1}{2} \left|\left| \frac{\mathbf{M_l}}{(1-p_l)} \right|\right|_2^2 + \sum_{l=1}^{L} K_l \mathcal{H}(p_l) - \log(p(\mathcal{D}|g(\theta, \epsilon))) \\ &= \sum_{l=1}^{L} (1-p_l) \frac{1}{2} \frac{1}{(1-p_l)^2} ||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L} K_l \mathcal{H}(p_l) - \log(p(\mathcal{D}|g(\theta, \epsilon))) \\ &= \sum_{l=1}^{L} \frac{1}{2} \frac{1}{(1-p_l)} ||\mathbf{M_l}||_2^2 + \sum_{l=1}^{L} K_l \mathcal{H}(p_l) - \log(p(\mathcal{D}|g(\theta, \epsilon))). \end{aligned} \tag{3.40}$$

3.5.1 Pytorch

We concluded that we needed to apply the same dropout masks to input and hidden state at every timestep. Current LSTM and LSTMCell implementations in Pytorch do not support dropout at input and hidden state at every timestep; the LSTM layer was therefore reimplemented for this thesis (while maintaining the same API).

We also concluded that our stochastic LSTM layer should expose its weight and dropout regularizing terms so that these can be added to loss value during optimization. The code implementation can be found in appendix D. As of this thesis, the version of Pytorch is 1.5.0.

One of our implementation goals was to follow the original LSTM API from Keras and Pytorch as much as possible, so that our stochastic LSTM layer can be used as a drop-in replacement for the built-in LSTM layer[4]. The *dropout* argument accepts a float between 0 and 1 exclusively, or can be None, to activate Concrete dropout. This means that dropout is always active.

There is no mechanism to constrain parameter value to a certain range in Pytorch, so we model the logit of dropout rate instead with an unconstrained range, and use sigmoid on that logit value to acquire the dropout rate.

3.5.2 Keras

Keras provided LSTM and LSTMCell layer with dropout masking for input and hidden state at every timestep. Therefore, for our implementation, we only need to override the default LSTM behaviour from optionally applying dropout during training, to always applying dropout given dropout masks that we sampled and maintain throughout all timesteps. Our implementation also allows for weight and dropout regularizers to be added to the loss value at layer level. The default Keras implementation of LSTM has separate dropout rates for inputs and hidden states, so in our Keras implementation, we also added two (2) different dropout rate Keras parameters. The code implementation can be found in appendix E. As of this thesis, the current version of Keras is 2.3.1. All arguments should behave similarly to built-in LSTM layer, except for dropout rates for input and recurrent data, where, for the case of dropout rate being specified as 1.0, it activates Concrete dropout. Otherwise, it should function as a regular dropout rate. The standard implementation features a mechanism to constrain parameter value to a certain range, so we model the dropout parameter directly. The internal structure of our solution was kept intact with regards to the built-in LSTM, so that our stochastic LSTM layer can be used in Bidirectional and TimeDistributed wrappers as well.

[4]In our implementation, we do not implement bidirectional feature.

Chapter 4

Results

4.1 Data and evaluation goals

To evaluate the performance of our stochastic LSTM scheme, we use several sequential datasets and build models with stochastic LSTM layers. The models will be assessed to determine a) whether the optimising objective function yields lower loss and better metrics, b) whether test metrics are good, and c) whether the dropout parameters are updated sensibly.

- *Occupancy Detection Data Set* (`https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+`): Experimental data used for binary classification (room occupancy) from Temperature, Humidity, Light and CO2 readings. Ground-truth occupancy was obtained from time stamped pictures that were taken every minute.

- *Hill-Valley Data Set* (`http://archive.ics.uci.edu/ml/datasets/hill-valley`): Each record represents 100 points on a two-dimensional graph. When plotted in order (from 1 through 100) as the Y co-ordinate, the points will create either a Hill (a "bump" in the terrain) or a Valley (a "dip" in the terrain).

- *Apple Stock data* (2012/01/01 - 2019/12/17) from Yahoo Finance[1]: Used for regression (closing price), from a sequence of previous closing prices.

- *MIMIC-III clinical time series data*: the main data for full implementation evaluation.

The scientific benchmark **MIMIC-III** [9] includes *Clinical time-series data* of anonymous patients, and a suite of NN models to perform different tasks, mostly classification. We use this benchmark as a standard, against which we compare our approach. There are many different benchmark tasks, but we limit our scope to *In-hospital-mortality* binary classification task, and we collect only training and test data related to this task. The train-test split of the data was prepared by the MIMIC

[1]Acquired using Pandas Data Reader `https://pydata.github.io/pandas-datareader/stable/index.html`

team; we use the same train-test dataset to recreate the benchmark for the original provided models. With the described scope, we have the following statistics on our data:

training samples	validation samples	test samples
14681	3222	3236

	train	test
Negative	15480	2862
Positive	2423	374

The provided data have been preprocessed; as suitable vector inputs for NN models. Although there is a class imbalance, where we have significantly more negative samples than we do positive samples, no class imbalance remedies are utilized. An input unit is a sequence of input vectors spanning over a 48-hour period, where each vector has 17 features. From training dataset, we created 9 other subsets, with fractions of samples from the original training set, ranging from 10% to 90%. We trained our stochastic models from the beginning with each of these subsets, to monitor the behaviour of epistemic uncertainty. Then, we augmented our test data values to create *out-of-distribution* test data and thus see how the augmentation affected aleatoric uncertainty. Test data were augmented as follows:

- *Glascow coma scale total* were randomly assigned an integer value from 3 to 15.
- *Heart Rate* were all set to a flat value of 400.
- *Diastolic blood pressure, Temperature, pH* were transformed according to function $f(x) = \frac{0.2x^3+0.2x^2+0.2x+0.2}{(1+\exp(-x))}$.
- *Glucose, Systolic blood pressure* were transformed according to function $f(x) = 0.3arctan(x) + 0.3arcsinh(x) + 0.4$

4.2 Experiments

Occupancy detection

This task detects if a room is occupied based on environment readings over a period of five (5) minutes. We shall use this simple dataset to validate the implementation of Dropout LSTM in a classification task with multi-dimensional inputs. Our objective is to check if using MC dropout can yield better accuracy, precision, or receiver operating characteristic (ROC). The model for this classification task has a single LSTM layer, and two (2) fully-connected layers, with softmax in the output layer.

Hill-Valley detection

This task detects if a terrain is a hill or valley, based on 100 sequential terrain positions. We shall use this simple dataset to validate the implementation of Dropout LSTM in a classification task. Our objective is to check if using MC dropout can yield better accuracy, precision, or receiver operating characteristic (ROC), and relatively interpretable uncertainty. The model for this classification task has a single LSTM layer, and a single fully-connected layers, with softmax in the output layer.

Stock prediction

This task predicts the closing stock price, given closing prices from the previous 50 days. We shall use this to validate the implementation of Dropout LSTM in a regression task. Our objective here is to see if using MC dropout can remedy overfitting. The model for this regression task has a single LSTM layer, and two (2) fully connected layers.

MIMIC-III clinical time series benchmark

This is a good set of benchmark tasks to evaluate the solution in this paper in a real medical application context. The benchmark (implemented in Keras) as been provided at `https://github.com/YerevaNN/mimic3-benchmarks`. We only focus on the NN model benchmark, and replace the original LSTM layers with our custom MC dropout version.
There are 4 main predictive tasks using clinical time-series data in this benchmark:

- In hospital mortality - binary classification
- Length of stay - reformulated as multiclass classification
- Decompensation[2] - binary classification
- Phenotyping - multilabel classification

In the scope of this paper, our objective is to replicate the *In hospital mortality* benchmark using our stochastic RNN model, and compare its performance against the existing standard RNN solution. The other benchmarks are also classification tasks, and thus provide little gain to the justification of the uncertainty quantification solution in this paper.

Stochastic dropout models enable us to get prediction calibration (as suggested by Chuan Guo et al. [8], and Yaniv Ovadia, Emily Fertig et al. [20]).

We have the following models for this task:

[2]https://en.wikipedia.org/wiki/Decompensation

LSTM	original model provided by benchmark suite using LSTM layers
Channel-wise LSTM	a channel-wise variant where each input features are processed separately in different LSTM layers. There are 17 parallel LSTM layers for 17 input features
LSTM with dropout rate 0.3	
Channel-wise LSTM with dropout rate 0.3	
LSTM with dropout rate 0.5	
Channel-wise LSTM with dropout rate 0.5	
LSTM with dropout rate concrete	
Channel-wise LSTM with dropout rate concrete	

Models with a dropout rate replace built-in LSTM layers with custom stochastic LSTM, and outputs are stochastically sampled 10 times to approximate a predictive posterior. In Keras, we use TimeDistributed wrapper to parallelize the sampling step, resulting in the same inference runtime. Each model is trained for 100 epochs, and parameters from iterations with the lowest validation crossentropy loss, are selected for testing. LSTM models with dropout, and the Channel-wise LSTM model with dropout 0.5 are also subjected to training with varying training set sizes. All stochastic models are tested against augmented test data (see section 4.1).

To quantify performance, we use the following metrics to compare our models with the provided standard models:

- Binary cross entropy loss (excluding weight regularizers)
- Accuracy
- Precision for class 0 (patient survives)
- Precision for class 1 (patient dies)
- Recall for class 0
- Recall for class 1
- AUC of ROC (Area under Receiver operating characteristic curve)
- AUC of PRC (Area under Precision - Recall curve)

And finally, we evaluate the predictive uncertainty in two (2) fashions:

- We observe how epistemic uncertainty behaves with different training set sizes (see section 4.1). We expect to see greater epistemic uncertainty when the models are trained with less training data.
- We observe how epistemic and aleatoric uncertainty changes when the models are trained under the same original training set, and later tested with the augmented test data. We expect greater epistemic and aleatoric uncertainty for these augmented test data because the models have not seen these data before.

Objective function

We use the following weighted objective function:

$$\begin{aligned} Loss = &- \frac{1}{N} \sum_{n=1}^{N} (\mathbf{p}_n \log(\mathbf{q}_n) + (1-\mathbf{p}_n) \log(1-\mathbf{q}_n)) \\ &+ \frac{\lambda_1}{2} \sum_{l=1}^{L} \frac{1}{1-p_l} ||\mathbf{M}_l||^2 \\ &+ \lambda_2 \sum_{l=1}^{L} K_l (p_l \log(p_l) + (1-p_l)\log(1-p_l)), \end{aligned} \tag{4.1}$$

where $\lambda_1 = \frac{1}{14681}$ and $\lambda_2 = \frac{2}{14681}$. Note that in our derived loss function, we did not divide the total loss by the number of training instances N. It is in general a good practice to normalize loss by N, and we do this in our implementation. The regularizing coefficients should also be adapted to different training sizes in the varying-training-size experiment, but for simplicity, we keep these coefficients fixed.

4.3 Results on auxiliary experiments for implementation validation

Occupancy detection

Model	AUC-ROC	Accuracy	Precision
Deterministic	0.98687782921	0.96061185468	**0.93796791443**
0.5 dropout	0.98350162570	0.96654135338	0.92292870905
Concrete dropout	**0.98718489845**	**0.96804511278**	0.92403846153

Table 4.1: Metrics for different models in Occupancy detection task.

The Concrete dropout model gives the best performance in AUC-ROC and accuracy. The deterministic model gives best precision. All metrics are very similar across all models. In the Concrete model, the learned dropout rate is 0.33.

The takeaway from this preliminary result is that, too much stochastic dropout can hurt performance, but an adequate dropout rate can improve performance.

Hill-Valley detection

Model	AUC-ROC	Accuracy	Precision
Deterministic	1.0	1.0	1.0
0.5 dropout	1.0	1.0	1.0
Concrete dropout	1.0	1.0	1.0

Table 4.2: Metrics for different models in Hill-Valley detection task.

All the metrics are maximized, and thus it is not possible to compare these different models on such a simple dataset. In the Concrete model, the learned dropout rate is 0.24. Uncertainty is inspected, instead, in this experiment.

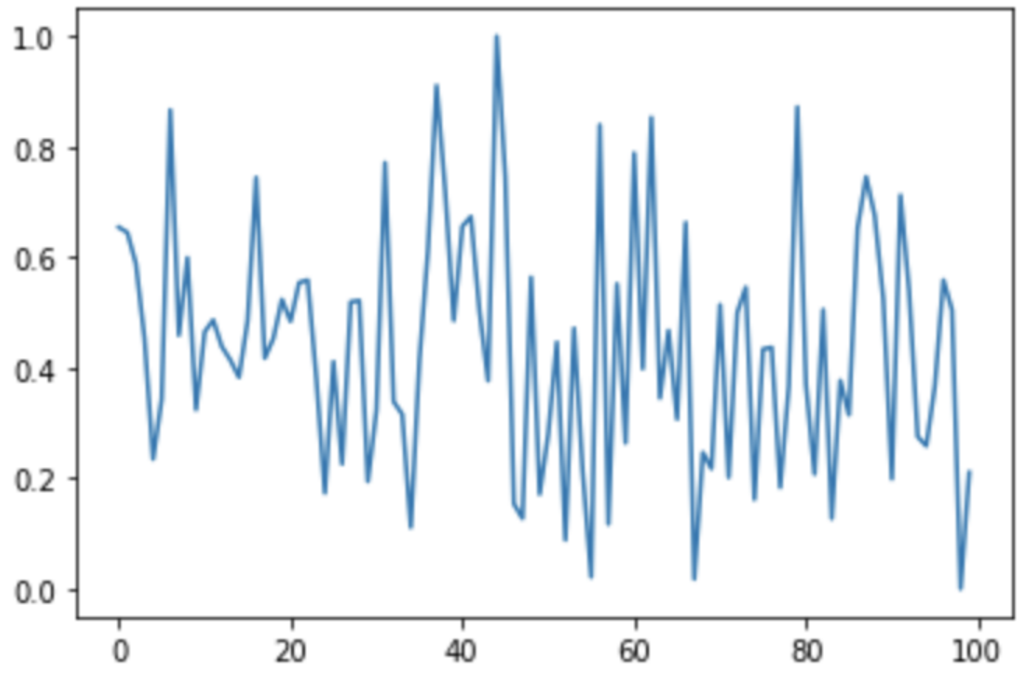

Figure 4.1: Synthetic input for Hill-Valley detection task for stochastic models. This terrain was generated in the experiment to observe how prediction and uncertainty behave with an unclassifiable input; neither a hill, nor a valley.

Figure 4.1 is a sample generated on purpose to not look like a hill or a valley. In stochastic model with dropout rate 0.5, the predicted probability that this is a hill is 0.5574, and the uncertainty is 0.2467; very uncertain about the prediction. In stochastic model with Concrete dropout, the predicted probability that this is a hill is 0.9804, and the uncertainty is 0.0192; almost very certain that this is a hill. In the stochastic model with 0.5 dropout, the uncertainty is almost at maximum (maximum variance for a Bernoulli distribution is 0.25). This is justified because it is almost impossible to classified this terrain as a hill or a valley. In the stochastic model with

Concrete dropout, however, the uncertainty is very low, hinting that uncertainty is unreliable in models with Concrete dropout.

Stock price prediction

Model	Mean square error
Deterministic	**5.893952**
0.5 dropout	6.474945
Concrete dropout	7.818875

Table 4.3: Mean square error for stock price prediction task in different models

Stochastic dropout seems to hurt performance. This is not enough evidence to conclude that stochastic models are not good for regression tasks, however. The uncertainty for all stochastic models are very small (in the order of 10^{-3}). The predictions are shown in the following figures:

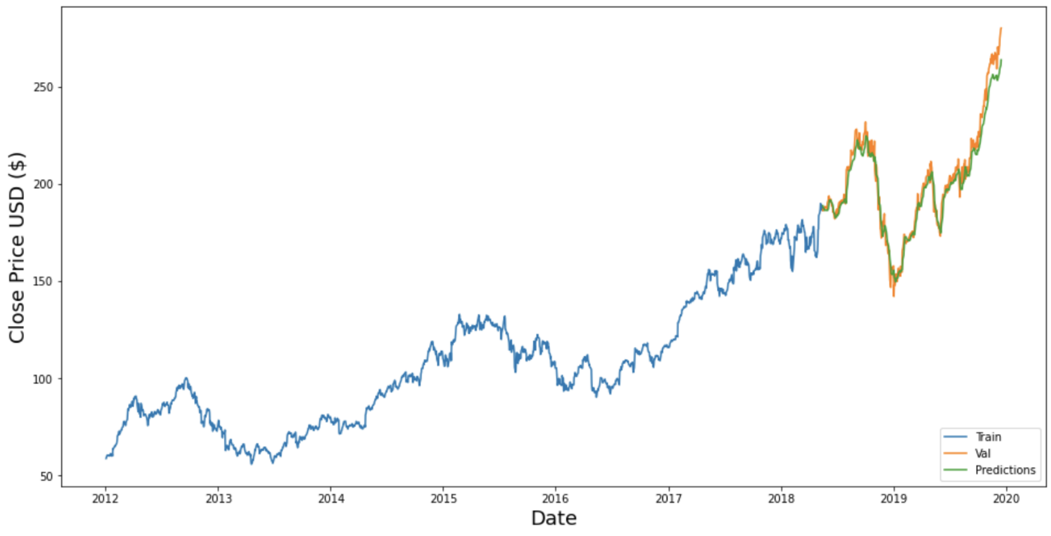

Figure 4.2: Stock price prediction using deterministic LSTM model. MSE = 5.893952

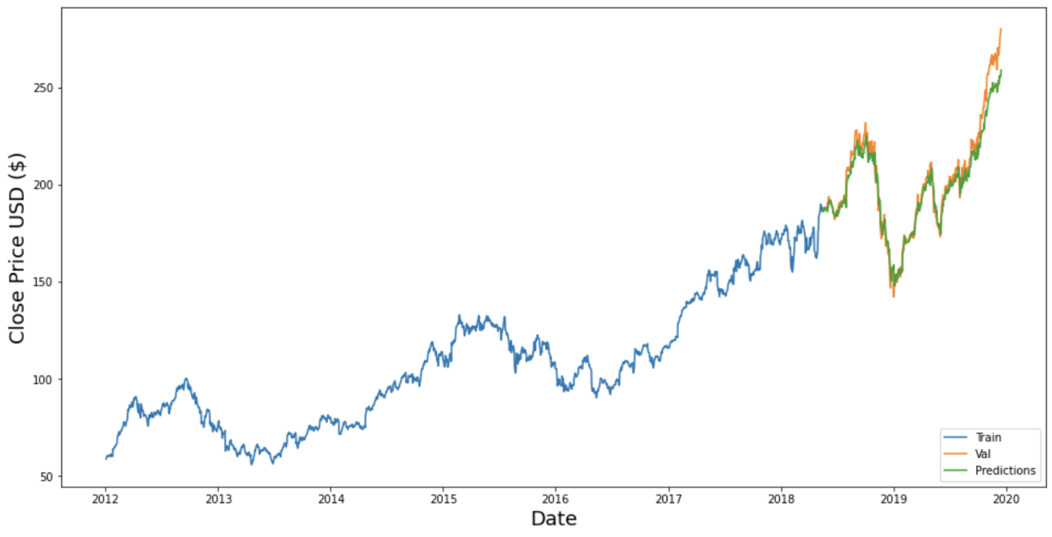

Figure 4.3: Stock price prediction using stochastic LSTM model with dropout 0.5. MSE = 6.474945

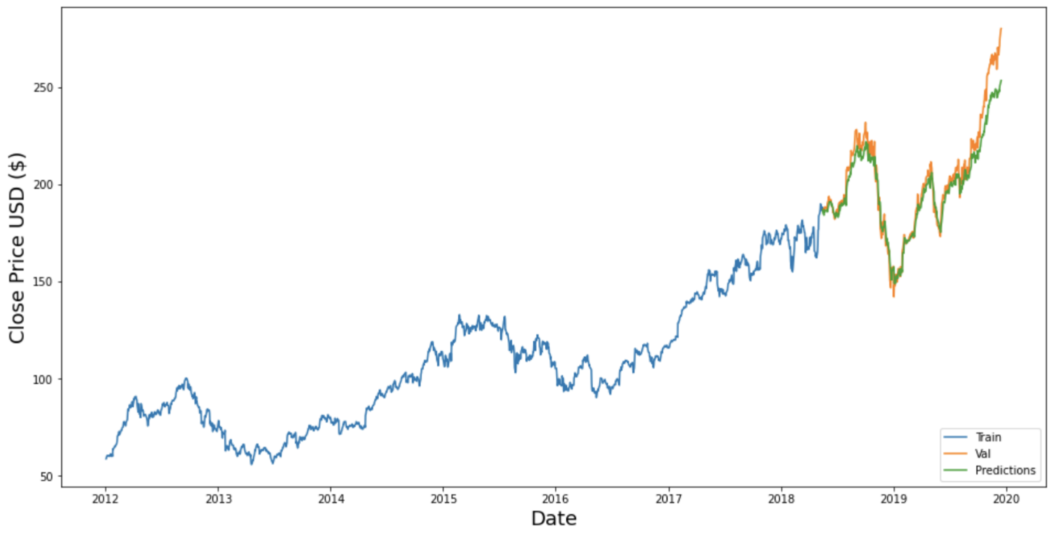

Figure 4.4: Stock price prediction using stochastic LSTM model with Concrete dropout. MSE = 7.818875

4.4 MIMIC-III benchmark results

4.4.1 Training

The training progresses for regular models are shown in figures 4.5, 4.6, 4.7, and 4.8. The training progresses for channel-wise models are in figures 4.9, 4.10, 4.11, and 4.12. Commentary on the training progress for each model is provided in the caption of each corresponding figure. Channel-wise models take about 3 days to train on a full training dataset, and regular models takes about 7 hours to train on a full training dataset. Training is carried out on NVIDIA Tesla V100 GPU.

The *baseline regular model* and the baseline *channel-wise model* show signs of overfitting as validation loss increases. However, they yield the lowest validation loss.

Stochastic models with dropout 0.3 and 0.5 show slightly worse validation loss (excluding weight regularizers), but they do not show signs of overfitting, hinting that more epochs may improve these models. However, the loss value oscillates aggressively, hinting that these models may benefit from using a smaller learning rate.

Regular and channel-wise models with Concrete dropout overfit very quickly. While yielding better validation loss than fixed dropout models, they still perform slightly worse in terms of validation loss when compared to the baseline models. These Concrete dropout models reach the lowest validation loss values very quickly.

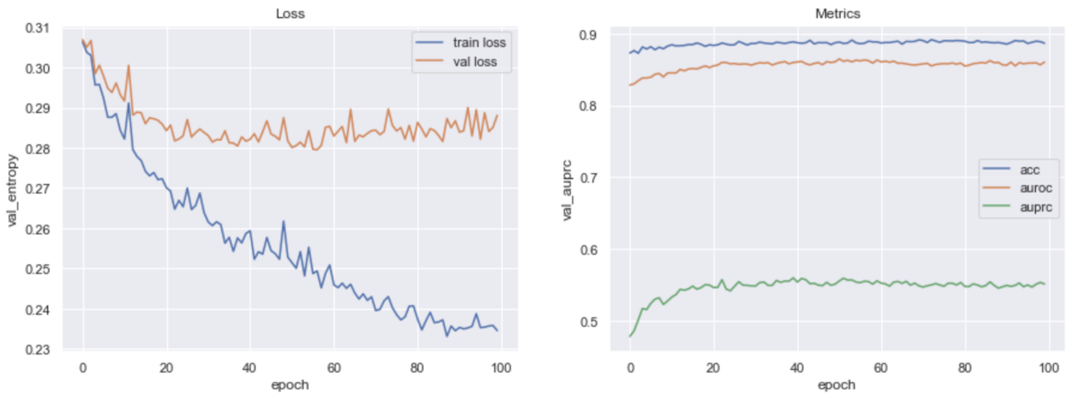

Figure 4.5: Baseline training progress, best validation loss = 0.279520 at epoch 57.

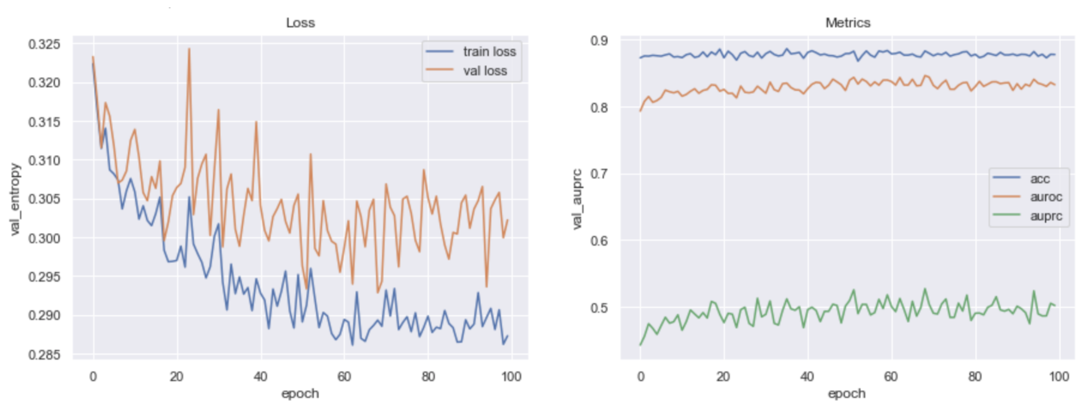

Figure 4.6: Training progress with dropout = 0.3, best validation loss = 0.292824 at epoch 57.

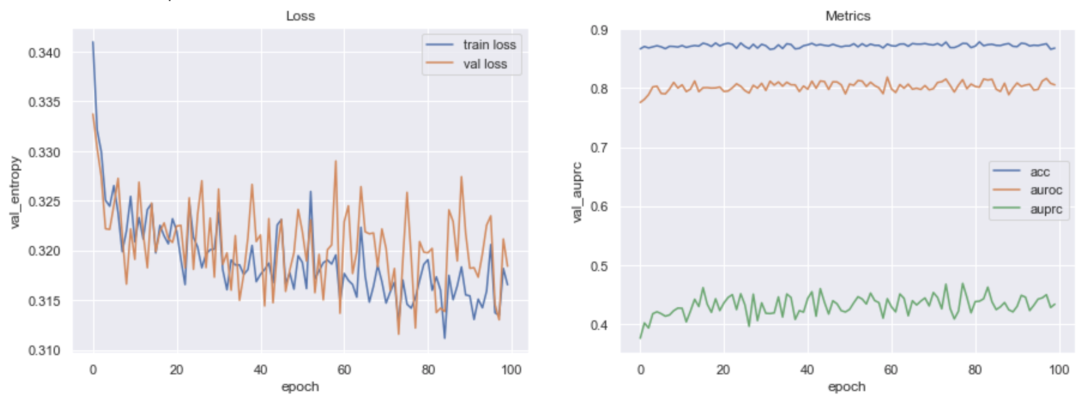

Figure 4.7: Training progress with dropout = 0.5, best validation loss = 0.311563 at epoch 74.

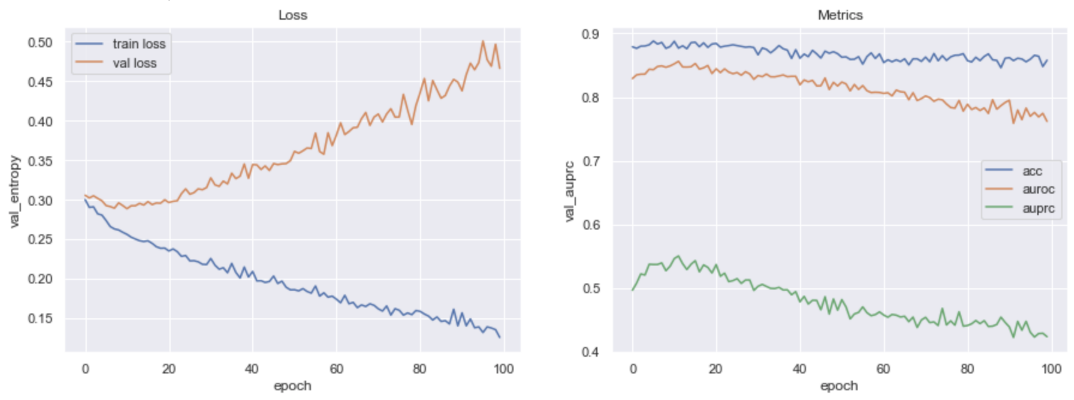

Figure 4.8: Training progress with Concrete dropout, best validation loss = 0.288171 at epoch 11.

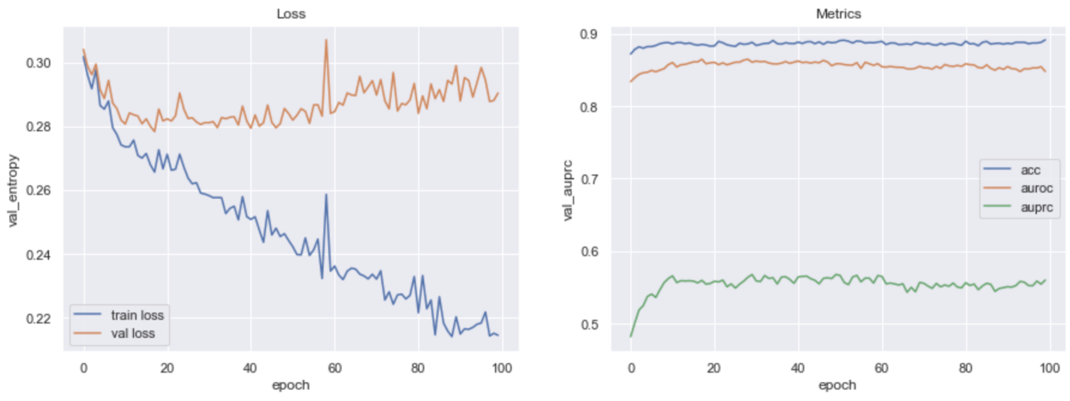

Figure 4.9: Channel-wise Baseline training progress, best validation loss = 0.278306 at epoch 18.

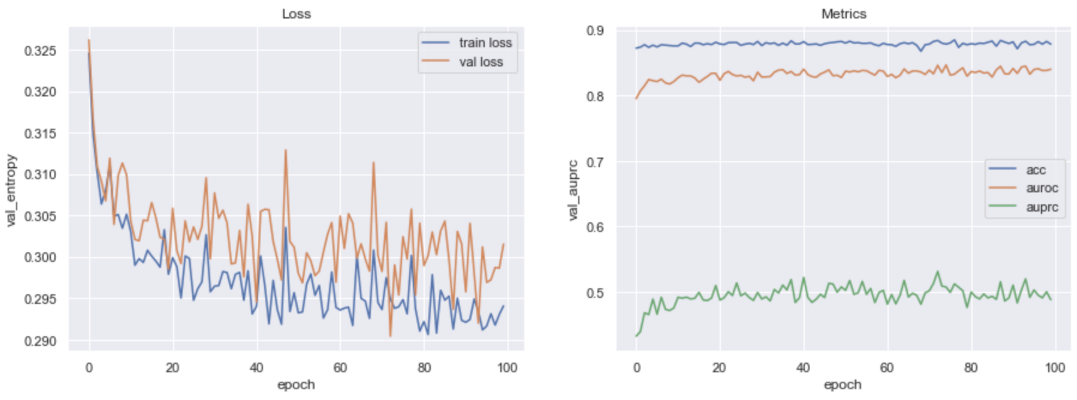

Figure 4.10: Channel-wise training progress with dropout = 0.3, best validation loss = 0.290423 at epoch 73.

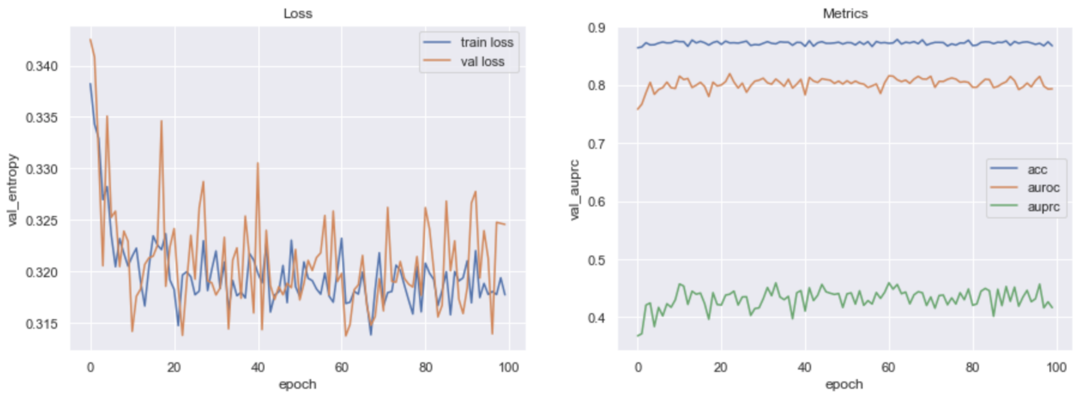

Figure 4.11: Channel-wise training progress with dropout = 0.5, best validation loss = 0.313742 at epoch 62.

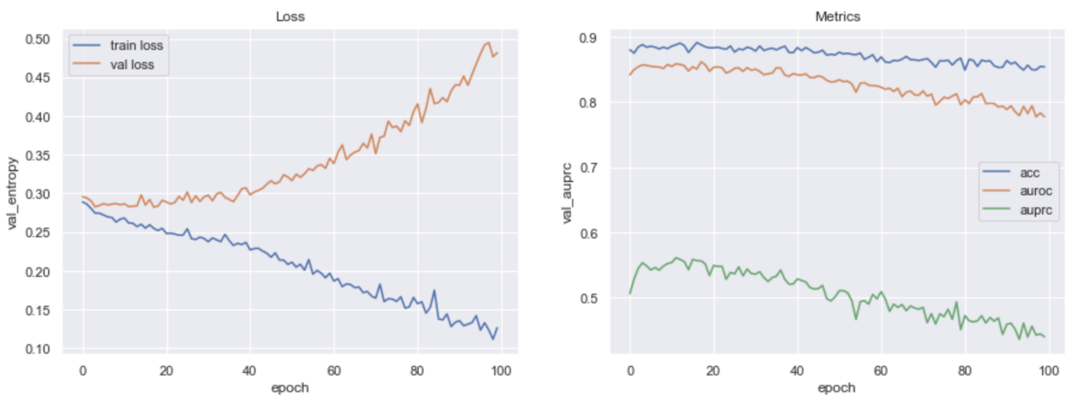

Figure 4.12: Channel-wise training progress with Concrete dropout, best validation loss = 0.282398 at epoch 18.

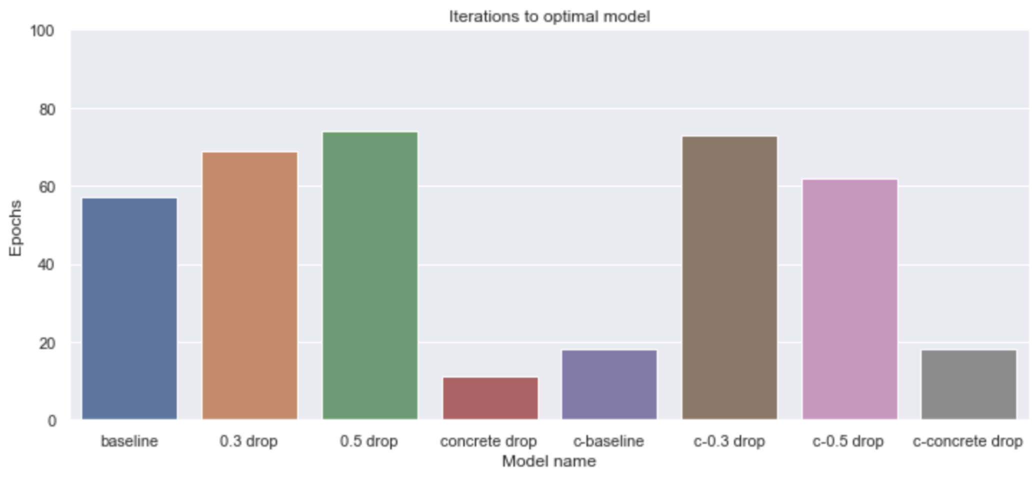

Figure 4.13: Training epochs to lowest validation loss. Lower is better.

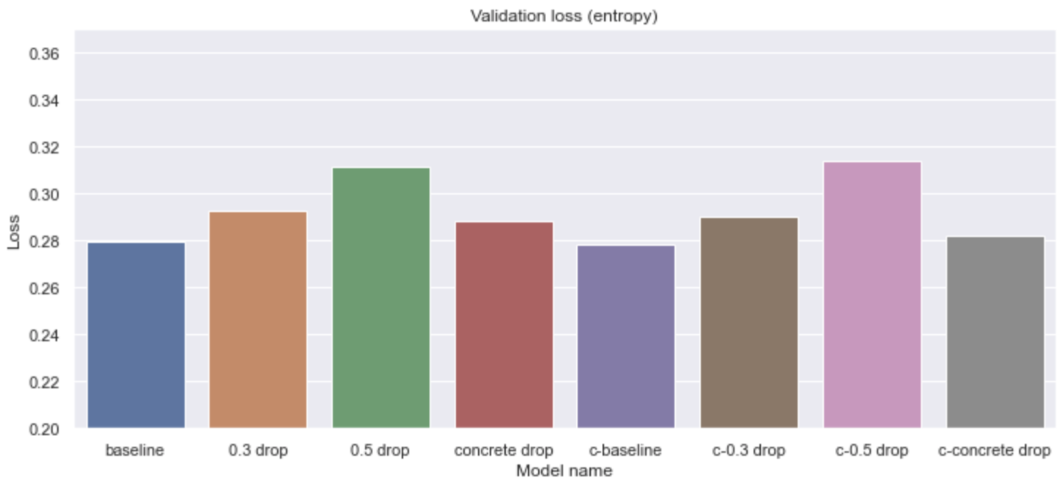

Figure 4.14: Comparison of crossentropy validation loss among models. Lower is better.

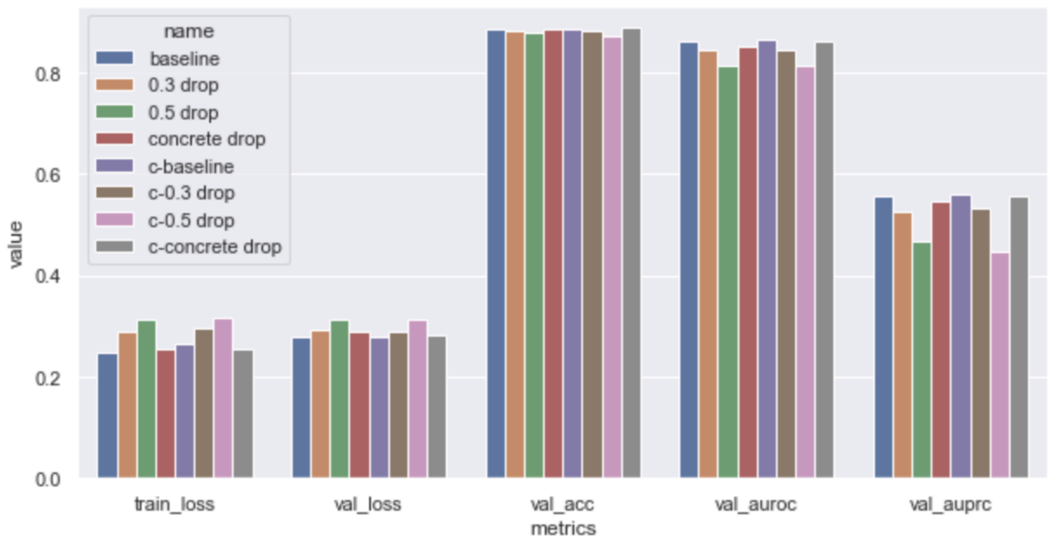

Figure 4.15: Metrics collected on training and validation data. With loss, lower is better. With accuracy, AUC_ROC, and AUC_PRC, higher is better. Concrete dropout models yield the best accuracy. Although the difference in metrics among models are very small.

4.4.2 Benchmark results

Benchmark metrics are shown in figure 4.16. We evaluate cross entropy again during test with true labels to see if we see similar rankings. The two (2) baseline models show the lowest cross entropy loss. Models with higher dropout rates have higher loss.

The stochastic regular models have better accuracy than their baselines. The channel-wise baseline model has the best accuracy, but stochastic channel-wise models are not far behind; the difference is in the order of magnitude of 10^{-3}.

Precision for negative samples is similar across models, with stochastic regular with Concrete dropout coming out on top. Stochastic models show obvious advantages in precision for positive samples, with difference of up to 0.2, compared to baseline models. Regular stochastic models also outperform channel-wise baseline models. Although this is not the most important medical metric, it can be useful in other scenarios where we want to reach many positive predictions.

Recalls for negative samples show the advantage of stochastic models over baseline models. It is interesting to see that Concrete models do not perform as well as other dropout models for recalling negative samples. In recalling positive samples, Concrete dropout models outperform other dropout models. In regular (non-multi-channel) models, Concrete dropout model gives the same metric as the baseline model. This indicates that too much dropout can hurt recall for *positive* class, which is the most important metric in medical context.
AUC-ROC for all models is similar, with baseline models coming out on top. AUC-PRC is also similar across models. We can take a closer look at AUC-ROC in figure 4.17, and AUC-PRC in figure 4.18. Stochastic models with dropout 0.5 perform worst in most metrics (by a small margin), but they also exhibit the highest uncertainty in figure 4.19.

The models perform similarly for most metrics but advantages of the stochastic models emerge for **precision for positive class metric** and **recall for positive class metric** (See Figure 4.16).

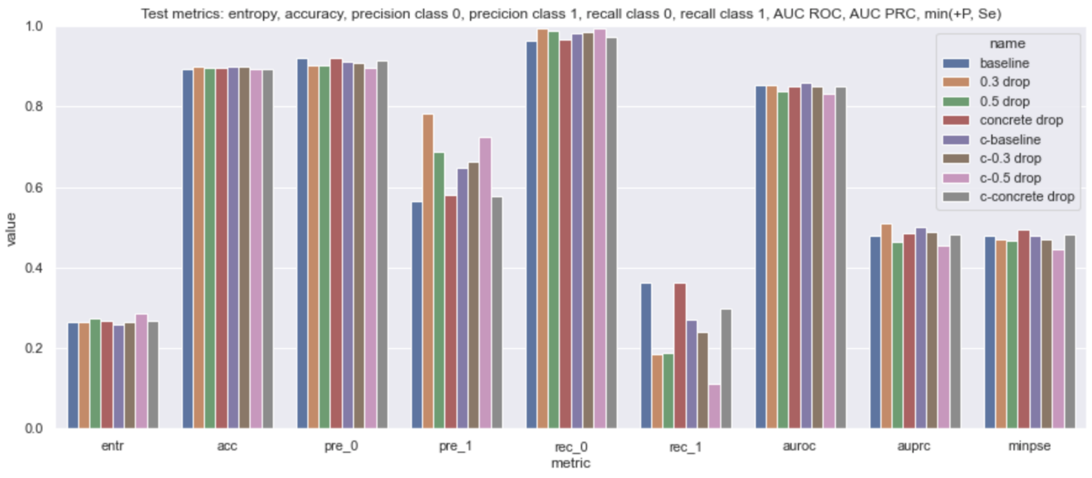

Figure 4.16: Test metrics for 8 models

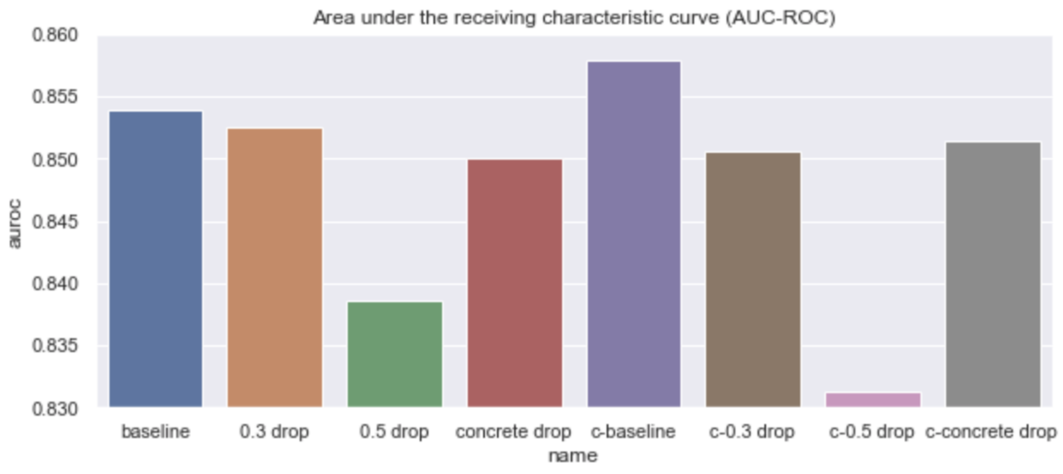

Figure 4.17: Test AUC-ROC for 8 models

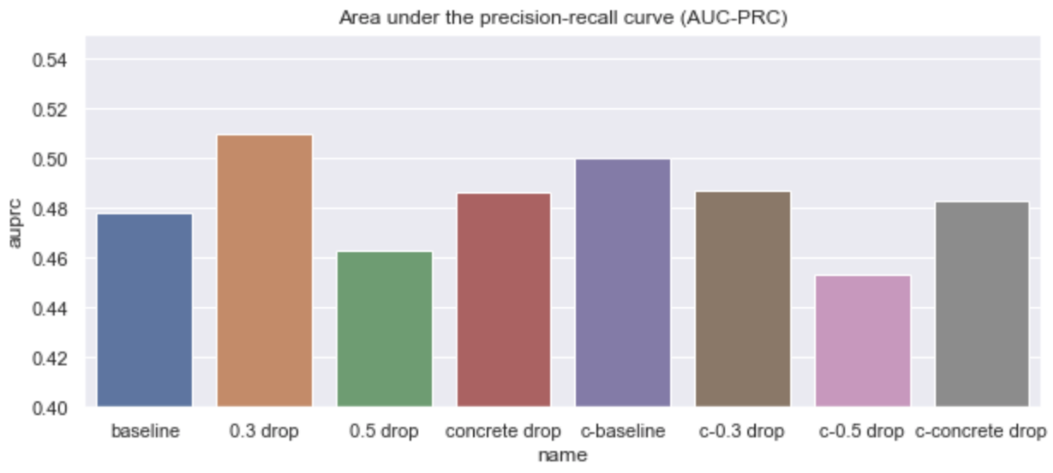

Figure 4.18: Test AUC-PRC for 8 models

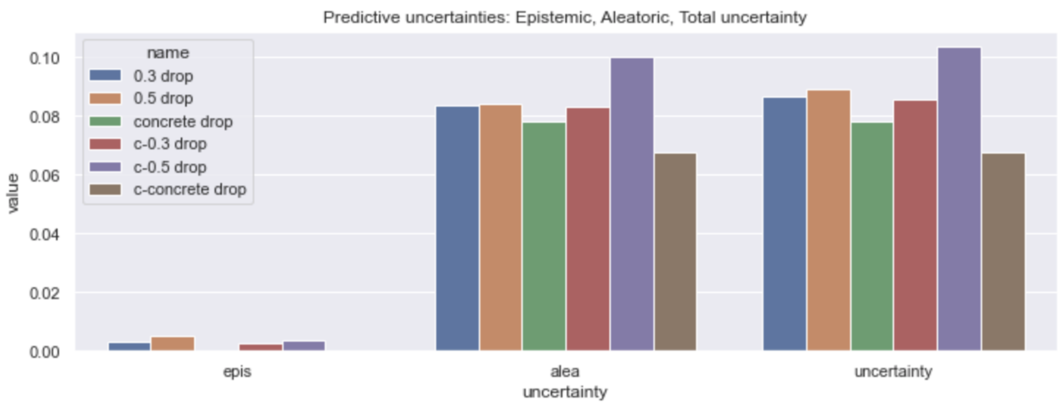

Figure 4.19: Test uncertainties for stochastic models. The uncertainties are averaged over all predictions. Well-trained models exhibit little epistemic. As expected, higher dropout rates exhibit higher uncertainty as expected.

4.4.3 Uncertainty evaluation

We evaluate how uncertainty shifts with changes in data.

Under different training dataset sizes

Figure 4.20 shows a decreasing epistemic trending associated for an increasing training dataset size, on four (4) stochastic models. This shows that we can judge how well training data span over possible cases. In the same figure, we see that the epistemic uncertainty for Concrete dropout model is almost a flat line at zero, with learned dropout rates around 0.25 to 0.35 for selected models. There seems to be no correlation between training size and epistemic uncertainty for Concrete dropout models.
We repeat the same comparison, this time looking at aleatoric uncertainty. Figure 4.21 shows that there is no clear trending here. The uncertainties vary up and down not irrespective of training size.

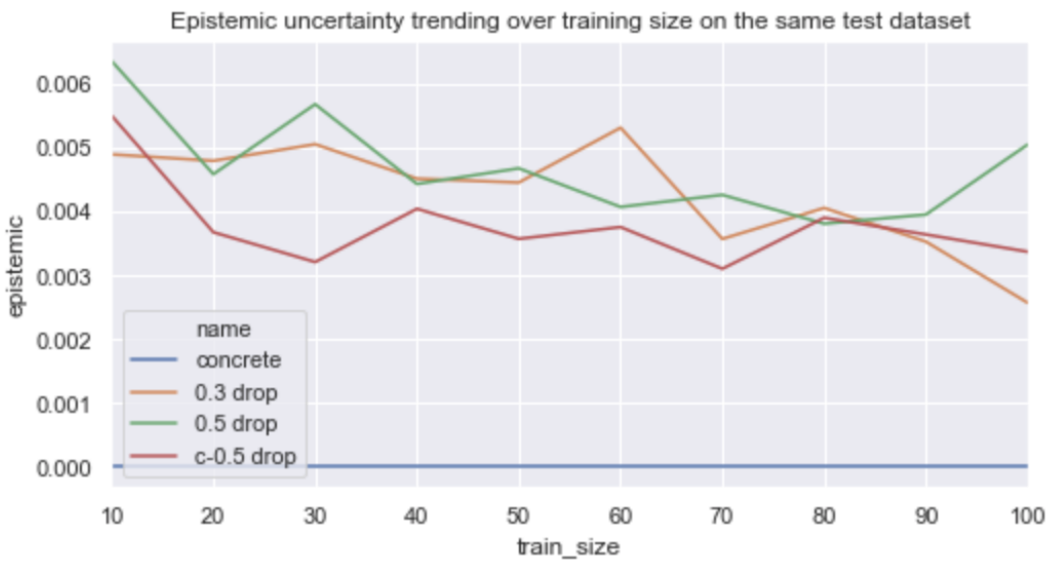

Figure 4.20: Epistemic uncertainty trending of different models over training datasets with different sizes (in percentage).

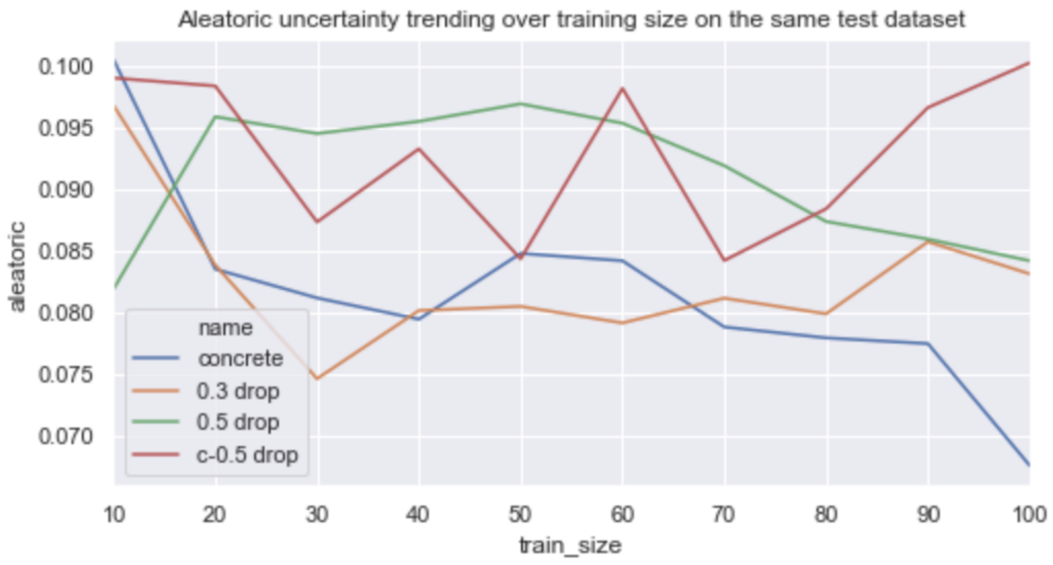

Figure 4.21: Aleatoric uncertainty trending of different models over training datasets with different sizes.

Under augmented test data

Finally, we test fully trained models against augmented test data (see the end of section 4.1 for test data augmentation); the uncertainties are compared in figure 4.22. In regular models, we see some unexpected results; augmented test data yield lower aleatoric uncertainty, yet higher epistemic uncertainty, except for the stochastic model with dropout 0.5, which behaves as expected. Channel-wise models show expected results, where augmented test data result in a massive aleatoric uncertainty increase. It is worth to point out that channel-wise models have a lot of stochastic layers compared to deterministic layers (17 stochastic LSTM layers for each features, and 1 more to combined the intermediate outputs), while there are only two (2), or even one (1) stochastic LSTM layers in regular models.
Epistemic uncertainty increases dramatically for augmented test data, except for Concrete dropout models, where epistemic uncertainty is almost zero.
These results reinforce that aleatoric uncertainty increases for *out-of-distribution* data.

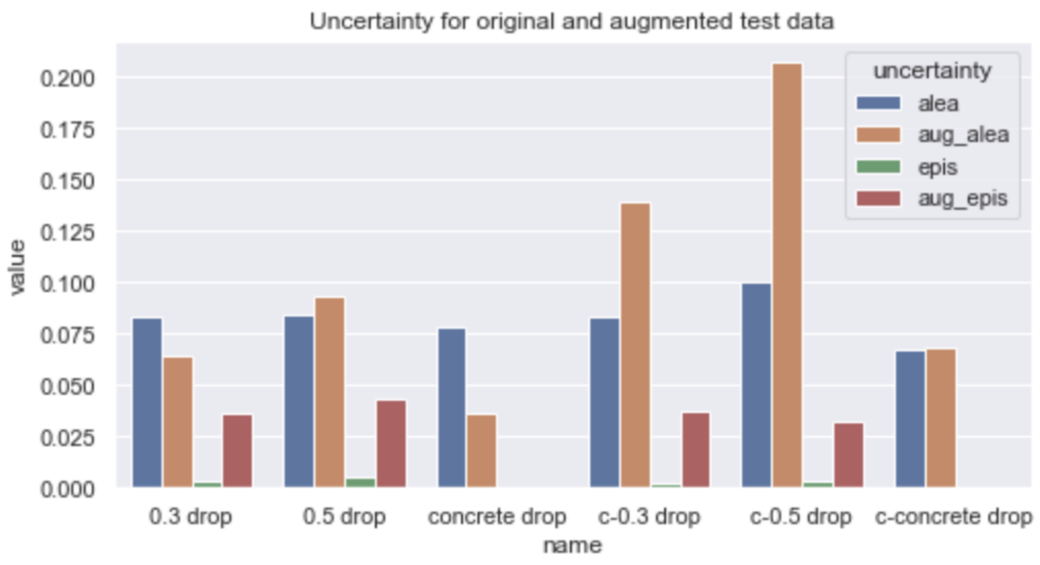

Figure 4.22: Uncertainty comparison between original test dataset and augmented test dataset of different models. Models with "c-" prefix are channel-wise models.

Chapter 5

Summary

We have laid the ground work for acquiring predictive uncertainty from stochastic Bayesian NN, specifically on RNN with a specific example on LSTM. Although we have not seen major improvement in metrics of Bayesian NN models over traditional models, the presented approach allows us to quantify epistemic and aleatoric uncertainty in predictions.

The concept can also be extended to other types of NN (see appendix B), such as fully-connected layer, because our proposal is not tied to the recurrent nature of our custom layer.

Our stochastic variational approach is able to quantify uncertainty and distinguish between epistemic and aleatoric uncertainties, by producing expected behaviours; the lack of training data leads to higher epistemic uncertainty, and out-of-distribution leads to higher epistemic and aleatoric uncertainty. However, we could see that Concrete dropout models tend to have no epistemic uncertainty at all; a side effect of learning dropout parameters is that the models learn to minimize epistemic uncertainty with more training data. It may therefore be best to avoid Concrete dropout, if we have an interest in epistemic uncertainty. If epistemic uncertainty is not our concern, Concrete dropout models presented here offer the advantage of training fast with competitive benchmark metrics compared to baseline models.

This work also informs that it is a good idea to make a stochastic model fully stochastic; this means that, excluding dropout parameters in Concrete dropout, all weight parameters should be stochastic. Further, the models being tested use LSTM and fully-connected layers. Therefore, we need to turn fully-connected layers stochastic. This can be done using the same technique presented in this paper, by applying dropout to inputs without concern about timestep.

A further contribution from this work is, we can boost the importance of positive samples because positive data (with outcome as death), are much fewer than negative data.

Using cross entropy loss is not always the best metric to select models. In the case of binary classification in general, AUC-ROC or AUC-PRC are better choices. In particular, AUC-PRC is a good choice for imbalanced data like our clinical data. *Recall for positive class* is also a good choice in case where we want to save as many people as we can.

In this work, we mentioned Gaussian dropout, but did not derive the equivalent objective function. A comparison between Gaussian dropout and our approach is a natural area of exploration; the predictive mean and variance should be the same, but objective function and our LSTM implementation should be adapted accordingly.

Finally, we did not perform any extensive analysis of uncertainty in regression task. The next appropriate step is to find a time series data benchmark on the scale of MIMIC-III for regression tasks to evaluate how well this solution can quantify uncertainty.

Bibliography

[1] Yarin Gal. "Uncertainty in Deep Learning". PhD thesis. University of Cambridge, Sept. 2016. Chap. 1.2, pp. 7–8.

[2] Yarin Gal. "Uncertainty in Deep Learning". PhD thesis. University of Cambridge, Sept. 2016. Chap. 1.2, pp. 9–13.

[3] Yarin Gal. "Uncertainty in Deep Learning". PhD thesis. University of Cambridge, Sept. 2016. Chap. 3, pp. 30–56.

[4] Yarin Gal and Zoubin Ghahramani. *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks*. 2015. arXiv: 1512.05287 [stat.ML].

[5] Yarin Gal and Zoubin Ghahramani. *Dropout as a Bayesian Approximation: Appendix*. 2015. arXiv: 1506.02157 [stat.ML].

[6] Yarin Gal and Zoubin Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. 2015. arXiv: 1506.02142 [stat.ML].

[7] Yarin Gal, Jiri Hron, and Alex Kendall. *Concrete Dropout*. 2017. arXiv: 1705.07832 [stat.ML].

[8] Chuan Guo et al. "On Calibration of Modern Neural Networks". In: *CoRR* abs/1706.04599 (2017). arXiv: 1706.04599. URL: http://arxiv.org/abs/1706.04599.

[9] Hrayr Harutyunyan et al. "Multitask Learning and Benchmarking with Clinical Time Series Data". In: *Scientific Data* 6 (Mar. 2017). DOI: 10.1038/s41597-019-0103-9.

[10] S. Hochreiter et al. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". In: *A Field Guide to Dynamical Recurrent Neural Networks*. Ed. by S. C. Kremer and J. F. Kolen. IEEE Press, 2001.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[12] Jiri Hron, Alexander G. de G. Matthews, and Zoubin Ghahramani. *Variational Gaussian Dropout is not Bayesian*. 2017. arXiv: 1711.02989 [stat.ML].

[13] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks: Design and Applications*. Subs. of Times Mirror 2000 Corporate Blvd. NW Boca Raton, FLUnited States: CRC Press, Inc., Jan. 1999.

[14] Eric Jang, Shixiang Gu, and Ben Poole. *Categorical Reparameterization with Gumbel-Softmax*. 2016. arXiv: 1611.01144 [stat.ML].

[15] Alex Kendall and Yarin Gal. "What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?" In: *CoRR* abs/1703.04977 (2017). arXiv: 1703.04977. URL: http://arxiv.org/abs/1703.04977.

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

[17] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. arXiv: 1312.6114 [stat.ML].

[18] 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.

[19] Yongchan Kwon et al. "Uncertainty quantification using Bayesian neural networks in classification: Application to biomedical image segmentation". In: *Computational Statistics & Data Analysis* 142 (2020), p. 106816. ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2019.106816. URL: http://www.sciencedirect.com/science/article/pii/S016794731930163X.

[20] Yaniv Ovadia et al. *Can You Trust Your Model's Uncertainty? Evaluating Predictive Uncertainty Under Dataset Shift*. 2019. arXiv: 1906.02530 [stat.ML].

[21] Donald B. Rubin. "The Bayesian Bootstrap". In: *The Annals of Statistics* 9.1 (1981), pp. 130–134. ISSN: 00905364. URL: http://www.jstor.org/stable/2240875.

Appendix A

Reparameterized distribution sampling

In this work, we use a lot of Gaussian distribution reparameterization. The original distribution cannot be transformed, but we can instead transform the way we sample. Given a distribution $\mathbf{N}(x|\mu, \sigma^2)$, if we sample x directly, we do not have any μ or σ variable in the samples. We want to have μ and σ because we want to optimise them. If instead, we sample from a standard Gaussian distribution $\epsilon \sim \mathcal{N}(0, 1)$, and transform these samples as:

$$\mu + \sigma\epsilon. \tag{A.01}$$

The samples now have μ and σ that we are interested in optimising. Similarly, a Bernoulli distribution can be approximately reparameterized as Concrete distribution, as presented by Kingma et al. [16].

Appendix B

Apply VB-MC to other types of NN

The general technique to MC dropout is to apply dropout to the input vector before it engages with a weight matrix for a single forward pass. The approximated posterior distribution determines a dropout technique that manifests as a dropout mask. In the case of fully-connected layer, it is just a simple matrix multiplication:

$$out = (\mathbf{x} \circ \mathbf{mask}_i)\mathbf{W}_i. \tag{B.01}$$

In the case of CNN, dropout can simply be performed on input image before passing that dropped image to a regular CNN module.

Appendix C

VB-MC for regression tasks

Unlike in classification, where we use the same cross entropy loss, we cannot use the same MSE loss like normal regression problem. From the objective function equation 3.18, for regression, we have this negative log likelihood term.

$$\frac{1}{2}\sum_{i=1}^{N}(s_i + \exp(-s_i)||y_i - \hat{y}_i||^2). \tag{C.01}$$

We have the extra $s_i = \log(\sigma_i^2)$ output that we cannot ignore. In a regular regression context, we assume outputs to have a fixed noise, and more specifically, we assume output noise is 1.0.However, since we also want to predict output noise, we cannot fixed these. Hence, we are seeing a full Gaussian log likelihood:

$$\begin{aligned}\log(\mathbf{N}(x|\mu,\sigma^2) &= -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{1}{2}\frac{1}{\sigma^2}(x-\mu)^2 \\ &\propto -\log(\sigma) - \frac{1}{2}\frac{1}{\sigma^2}(x-\mu)^2,\end{aligned} \tag{C.02}$$

where x is equivalent to our $\hat{y}_i$, and μ is equivalent to y_i.

We mentioned that it is more numerically stable to output log of variance than to output variance (because variance cannot have non-positive value). This means we have to exponentiate predicted log variance first, before attempting to calculate uncertainty.

Appendix D

Pytorch implementation

```python
"""Dropout variant of RNN layers
Binary dropout is applied in training and in inference
User can specify dropout rate, or
dropout rate can be learned during training
"""
from typing import Optional, Tuple
import torch
from torch import nn, Tensor


class StochasticLSTMCell(nn.Module):
    def __init__(self, input_size: int, hidden_size: int, dropout: Optional[float]=None):
        """
        Args:
        - dropout: should be between 0 and 1
        """
        super(StochasticLSTMCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        if dropout is None:
            self.p_logit = nn.Parameter(torch.empty(1).normal_())
        elif not 0 < dropout < 1:
            raise Exception("Dropout rate should be between in (0, 1)")
        else:
            self.p_logit = dropout

        self.Wi = nn.Linear(self.input_size, self.hidden_size)
        self.Wf = nn.Linear(self.input_size, self.hidden_size)
        self.Wo = nn.Linear(self.input_size, self.hidden_size)
        self.Wg = nn.Linear(self.input_size, self.hidden_size)

        self.Ui = nn.Linear(self.hidden_size, self.hidden_size)
        self.Uf = nn.Linear(self.hidden_size, self.hidden_size)
        self.Uo = nn.Linear(self.hidden_size, self.hidden_size)
        self.Ug = nn.Linear(self.hidden_size, self.hidden_size)

        self.init_weights()

    def init_weights(self):
        k = torch.tensor(self.hidden_size, dtype=torch.float32).reciprocal().sqrt()

        self.Wi.weight.data.uniform_(-k,k)
        self.Wi.bias.data.uniform_(-k,k)

        self.Wf.weight.data.uniform_(-k,k)
        self.Wf.bias.data.uniform_(-k,k)

        self.Wo.weight.data.uniform_(-k,k)
        self.Wo.bias.data.uniform_(-k,k)

        self.Wg.weight.data.uniform_(-k,k)
        self.Wg.bias.data.uniform_(-k,k)

        self.Ui.weight.data.uniform_(-k,k)
        self.Ui.bias.data.uniform_(-k,k)

        self.Uf.weight.data.uniform_(-k,k)
        self.Uf.bias.data.uniform_(-k,k)

        self.Uo.weight.data.uniform_(-k,k)
        self.Uo.bias.data.uniform_(-k,k)

        self.Ug.weight.data.uniform_(-k,k)
        self.Ug.bias.data.uniform_(-k,k)

    # Note: value p_logit at infinity can cause numerical instability
    def _sample_mask(self, B):
        """Dropout masks for 4 gates, scale input by 1 / (1 - p)"""
        if isinstance(self.p_logit, float):
            p = self.p_logit
        else:
            p = torch.sigmoid(self.p_logit)
        GATES = 4
        eps = torch.tensor(1e-7)
        t = 1e-1

        ux = torch.rand(GATES, B, self.input_size)
        uh = torch.rand(GATES, B, self.hidden_size)

        if self.input_size == 1:
            zx = (1-torch.sigmoid((torch.log(eps) - torch.log(1+eps)
                                   + torch.log(ux+eps) - torch.log(1-ux+eps))
                                 / t))
        else:
            zx = (1-torch.sigmoid((torch.log(p+eps) - torch.log(1-p
+eps)
                                   + torch.log(ux+eps) - torch.log
(1-ux+eps))
                                 / t)) / (1-p)
        zh = (1-torch.sigmoid((torch.log(p+eps) - torch.log(1-p+eps
)
                               + torch.log(uh+eps) - torch.log(1-uh
+eps))
                             / t)) / (1-p)
        return zx, zh

    def regularizer(self):
        if isinstance(self.p_logit, float):
            p = torch.tensor(self.p_logit)
        else:
            p = torch.sigmoid(self.p_logit)

        # Weight
        weight_sum = torch.tensor([
            torch.sum(params**2) for name, params in self.
named_parameters() if name.endswith("weight")
        ]).sum() / (1.-p)

        # Bias
        bias_sum = torch.tensor([
            torch.sum(params**2) for name, params in self.
named_parameters() if name.endswith("bias")
        ]).sum()

        if isinstance(self.p_logit, float):
            dropout_reg = torch.zeros(1)
        else:
             # Dropout
            dropout_reg = self.input_size * (p * torch.log(p) + (1-
p)*torch.log(1-p))
        return weight_sum, bias_sum, dropout_reg

    def forward(self, input: Tensor, hx: Optional[Tuple[Tensor,
Tensor]]=None) -> Tuple[Tensor, Tuple[Tensor, Tensor]]:
        """
        input shape (sequence, batch, input dimension)
        output shape (sequence, batch, output dimension)
        return output, (hidden_state, cell_state)
        """

        T, B = input.shape[0:2]

        if hx is None:
            h_t = torch.zeros(B, self.hidden_size, dtype=input.
dtype)
            c_t = torch.zeros(B, self.hidden_size, dtype=input.
dtype)
        else:
```

```python
            h_t, c_t = hx

        hn = torch.empty(T, B, self.hidden_size, dtype=input.dtype)

        # Masks
        zx, zh = self._sample_mask(B)

        for t in range(T):
            x_i, x_f, x_o, x_g = (input[t] * zx_ for zx_ in zx)
            h_i, h_f, h_o, h_g = (h_t * zh_ for zh_ in zh)

            i = torch.sigmoid(self.Ui(h_i) + self.Wi(x_i))
            f = torch.sigmoid(self.Uf(h_f) + self.Wf(x_f))
            o = torch.sigmoid(self.Uo(h_o) + self.Wo(x_o))
            g = torch.tanh(self.Ug(h_g) + self.Wg(x_g))

            c_t = f * c_t + i * g
            h_t = o * torch.tanh(c_t)
            hn[t] = h_t

        return hn, (h_t, c_t)


class StochasticLSTM(nn.Module):
    """LSTM stacked layers with dropout and MCMC"""

    def __init__(self, input_size: int, hidden_size: int, dropout: Optional[float]=None, num_layers: int=1):
        super(StochasticLSTM, self).__init__()
        self.num_layers = num_layers
        self.first_layer = StochasticLSTMCell(input_size, hidden_size, dropout)
        self.hidden_layers = nn.ModuleList([StochasticLSTMCell(hidden_size, hidden_size, dropout) for i in range(num_layers-1)])

    def regularizer(self):
        total_weight_reg, total_bias_reg, total_dropout_reg = self.first_layer.regularizer()
        for l in self.hidden_layers:
            weight, bias, dropout = l.regularizer()
            total_weight_reg += weight
            total_bias_reg += bias
            total_dropout_reg += dropout
        return total_weight_reg, total_bias_reg, total_dropout_reg

    def forward(self, input: Tensor, hx: Optional[Tuple[Tensor, Tensor]]=None) -> Tuple[Tensor, Tuple[Tensor, Tensor]]:
        B = input.shape[1]
        h_n = torch.empty(self.num_layers, B, self.first_layer.hidden_size)
        c_n = torch.empty(self.num_layers, B, self.first_layer.hidden_size)

        outputs, (h, c) = self.first_layer(input, hx)
        h_n[0] = h
        c_n[0] = c

        for i, layer in enumerate(self.hidden_layers):
            outputs, (h, c) = layer(outputs, (h, c))
            h_n[i+1] = h
            c_n[i+1] = c

        return outputs, (h_n, c_n)
```

Appendix E

Keras implementation

```python
from keras.layers import LSTM
from keras import initializers
from tensorflow.keras import backend as K


def get_mask(batch_size, dim, p):
    """Sample bernoulli mask from concrete distribution
    p: dropout rate"""
    t = 1e-1
    eps = K.epsilon()

    u = K.random_uniform(shape=(4, batch_size, dim))
    z = (1-K.sigmoid((K.log(p+eps) - K.log(1-p+eps) + K.log(u+eps)
    - K.log(1-u+eps)) / t))/(1-p)

    return z


class StochasticLSTM(LSTM):
    """StochasticLSTM that apply dropout to input and hidden state
    Note 1: do not set regularizers because dropout regularizers
    will be applied
    Note 2: there are 2 dropout rates: dropout for input, and
    recurrent_dropout for hidden state
    Note 3: to enable learning dropout rates, set dropout rates to
    1.0"""

    def build(self, input_shape):
        super().build(input_shape)

        reg = 1/14681
        dropout_reg = 2/14681
        def dropout_constraint(p):
            """Constraint probability between 0.0 and 1.0"""
            return K.clip(p, K.epsilon(), 1. - K.epsilon())

        if self.dropout == 1.0:
            self.p = self.cell.add_weight(name='p',
                                       shape=(),
                                       initializer=initializers.
uniform(minval=0.3, maxval=0.7),
                                       constraint=
dropout_constraint,
                                       trainable=True)
         self.add_loss(dropout_reg*input_shape[-1] *
                       (self.p * K.log(self.p) +
                        (1-self.p) * K.log(1-self.p)))
     else:
         self.p = self.dropout

     if self.recurrent_dropout == 1.0:
         self.p_r = self.cell.add_weight(name='p_recurrent',
                                       shape=(),
                                       initializer=initializers.
uniform(minval=0.3, maxval=0.7),
                                       constraint=
dropout_constraint,
                                       trainable=True)
         self.add_loss(dropout_reg*self.units *
                       (self.p_r * K.log(self.p_r) +
                        (1-self.p_r) * K.log(1-self.p_r)))
     else:
         self.p_r = self.recurrent_dropout

     # weight loss
     self.add_loss(reg / (1.-self.p) * K.sum(K.square(self.cell.
kernel)))
     self.add_loss(reg / (1.-self.p_r) * K.sum(K.square(self.
cell.recurrent_kernel)))
     self.add_loss(reg * K.sum(K.square(self.cell.bias)))

     self.built = True

 def call(self, inputs, mask=None, training=None, initial_state=
None):
     input_shape = K.shape(inputs)
     B = input_shape[0]
     D = input_shape[2]
     self.cell._dropout_mask = get_mask(B, D, self.p)
     self.cell._recurrent_dropout_mask = get_mask(B, self.units,
 self.p_r)
     return super(LSTM, self).call(inputs,
                                   mask=mask,
                                   training=training,
                                   initial_state=initial_state)
```

Appendix F

Code repository

There are two (2) code repositories in this thesis:

- The first one contains PyTorch implementation for the stochastic dropout LSTM layer, and the Jupyter notebooks for the three (3) small experiments (Occupancy detection, Hill-Valley detection, Stock price prediction): `https://github.com/nlhkh/dropout-in-rnn`
- The second one is a fork from the official MIMIC-III benchmark, with Keras implementation for the stochastic dropout LSTM layer, along with code to generate custom training and test dataset: `https://github.com/nlhkh/mimic3-benchmarks`