Probabilistic Deep Learning

Federico Ravenda[1]

[1]**Università degli Studi di Milano Bicocca, CdLM Clamses - STAT - matricola nr. 829449**

ABSTRACT

Deep learning has revolutionized the fields of machine learning and AI, achieving human level performance in several domains. However, standard deep learning techniques cannot quantify the uncertainty of their predictions. The aim of this report is to go deep into the *Probabilistic Deep Learning* world.

Probabilistic Deep Learning models become especially important when we encounter novel situations. These probabilistic models allowed us to describe the uncertainty inherent in data. We always need to deal with the inherent uncertainty in data if there's some randomness, meaning the observed outcome can't be determined completely by the input. This uncertainty is called ***aleatoric uncertainty***. But, as it turns out, there is also another kind of uncertainty inherent to the model. This uncertainty is called ***epistemic uncertainty***. Epistemic uncertainty occurs because it's not possible to estimate the model parameters without any doubt (due to the fact that we don't have unlimited training data).
Along the discussion we will see some major example of this approach including *Bayesian neural networks, Normalizing Flows* and *Variational Autoencoders* using ***TensorFlow Probability*** library which is a framework for probabilistic modeling and inference implemented by Google. I have included code examples for illustration in my github page.

Keywords: Probabilistic Deep Learning, Bayesian Neural Networks, Variational Inference, Normalizing Flows, Bayes By BackProp, MC Dropout, Variational Autoencoders

CONTENTS

1 BAYESIAN LEARNING - AN INTRODUCTION

This section introduces Bayesian models. Besides the likelihood approach, the Bayesian approach is the most important method to fit the parameters of a probabilistic model and to estimate the associated parameter uncertainty. The Bayesian modeling approach is able to incorporate the so-called *epistemic uncertainty*.

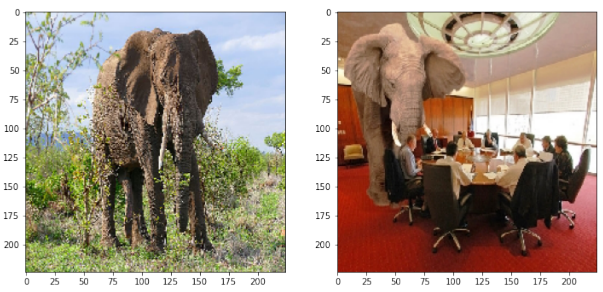
Incorporating epistemic uncertainty results in better prediction performance. This type of uncertainty becomes especially important during the prediction step, when the models see different situations from the ones seen during training phase.

Traditional non-Bayesian models don't express uncertainties, but Bayesian models do. Therefore, it's better to use Bayesian approaches when we have little training data or when we can't rule out that we'll encounter situations not seen during the training step.

As we will see further, even state-of-the-art Deep Learning models like those that win the ImageNet challenge are usually great in classifying elephants, but these are not able to classify correctly an elephant in a room. Instead, a wrong class is predicted, often with a high probability. This inability to communicate uncertainty when dealing with new situations and so produce unreliable predictions are a serious deficit of non-Bayesian DL models. Bayesian Deep Learning models, on the other hand, have the ability to express this type of uncertainty.

1.1 What's wrong with non-Bayesian DL? The Elephant In The Office Example

Deep Learning models can sometimes tell a completely wrong story. Usually, the predictions of traditional DL models are highly reliable when applied to the same data used in training, which somehow can take us into a false sense of security. Let's say we have two images: one, on the left, represents an elephant in his habitat, the other one, on the right, represents an elephant in an office.



Suppose we want to classify this two images using an architecture such as VGG16, trained on ImageNet.

In ImageNet there are a lot of elephant images that belong to 2 classes: 'Indian Elephant' and 'African Elephant'. Since VGG16 is a top architecture with high performance on ImageNet dataset, we can expect it is able to correctly classify the two elephant images. However, the same network that nicely classified the elephant in the left image as an african elephant species completely fails for the image on the right — the Deep Learning model can't see the elephant in the office!

If we consider the top 5 predictions of the high-performance network are :

1. library
2. wall_clock
3. shoe_shop
4. restaurant
5. toyshop

Not even close!

Why does the Deep Learning model fail to see the elephant? This is because in the training set used to fit the VGG16 model, there were no pictures of elephants in rooms. Not surprisingly, the elephant images in the training set show these animals in their natural environment. This is a typical situation where a trained DL model fails: when presented with an instance of a novel class or situation not seen during the training phase.

Exaggerating a bit, but DL crucially depends on the big lie:

$$P(train) = P(test)$$

The condition $P(train) = P(test)$ is always assumed; but in reality, the training and the test data don't often come from the same distribution. How can the network tell us that it feels unsure about a specific prediction? The solution is to introduce a new kind of uncertainty: the *epistemic uncertainty*.

1.1.1 *Aleatoric Uncertainty vs. Epistemic Uncertainty*

As mentioned, BNN is used to measure the uncertainties of the model. In fact, there are two types of uncertainties.

Aleatoric uncertainty is also known as statistical uncertainty. In Statistics, it is representative of unknowns that differ each time we run the same experiment (train the model). In deep learning, it refers to the uncertainty of the model outputs. As shown in the Figure 1 above, given that the black line is the prediction, the orange area would be the aleatoric uncertainty.
We can regard it as the confidence level of the prediction. In the other words, it tells us how confident our prediction results are. If the interval is small, the actual value would have a larger chance to have a closer value towards our prediction value. On the contrary, if the interval is large, the actual value may have a big discrepancy with our prediction value.

Epistemic uncertainty is also known as a systematic uncertainty. In deep learning, epistemic uncertainty refers to the uncertainty of the model weights. As shown in the Figure 1 below, every time we train the model, the weights may slightly vary. This variation is actually epistemic uncertainty.

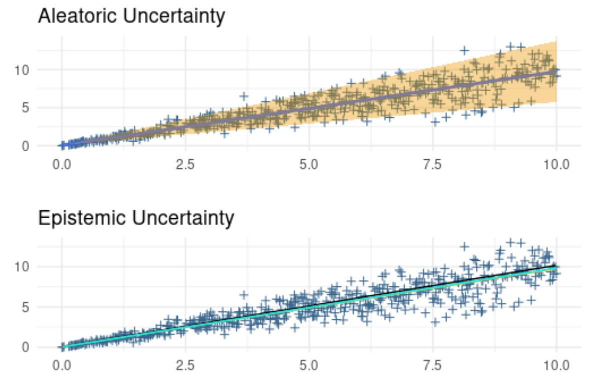


Figure 1. Graphical Representation of Aleatoric and Epistemic Uncertainty

1.2 The Bayesian Approach For Probabilistic Modelling

The idea of setting up models that incorporate the uncertainty about its parameter values via a probability distribution is quite old.
Thomas Bayes developed this approach in the 18th century. Nowadays, a whole branch in statistics is called Bayesian statistics.

The Bayesian approach is a new approach to fit probabilistic models that capture different kinds of uncertainties. It's an alternative way of doing statistics and interpreting probability.

In frequentist statistics, probability is a stranger concept (it is, indeed, replaced with confidence) and it is defined by analyzing repeated (frequent) measurements.

In Bayesian statistics, in contrast, probability is defined in terms of *degree of belief*. The more likely an outcome or a certain value of a parameter is, the higher the degree of belief in it.

Bayesian methods can be used to calculate the distribution of a model parameter given some data. The key step relies on Bayes' theorem. This theorem states, in mathematical notation, that

$$P(w|D) = \frac{P(D|w)P(w)}{\int P(D|w')P(w')\mathrm{d}w'}$$

where the terms mean the following:

1. D is some data, e.g. x and y value pairs: $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$. This is sometimes called the *evidence*.

2. w is the value of a model weight.

3. $P(w)$ is called the *prior*. This is our 'prior' belief on the probability density of a model weight, i.e. the distribution that we postulate before seeing any data.

4. $P(D|w)$ is the *likelihood* of having observed data D given weight w. It is precisely the same likelihood we discussed in the previous reading and is used to calculate the negative log-likelihood.

5. $P(w|D)$ is the *posterior* density of the distribution of the model weight at value w, given our training data. It is called *posterior* since it represents the distribution of our model weight *after* taking the training data into account.

We can note that the term $\int P(D|w')P(w')\mathrm{d}w' = P(D)$ does not depend on w (as the w' is an integration variable). It is only a normalisation term. For this reason, we will from this point on write Bayes' theorem as

$$P(w|D) = \frac{P(D|w)P(w)}{P(D)}.$$

Bayes' theorem gives us a way of combining data with some *'prior belief'* on model parameters to obtain a distribution for these model parameters that considers the data, called the *posterior distribution*.

The main idea is as follows. In a traditional neural network, as shown in the left in Figure 2, each weight has a single value. The true value of this weight is not certain. A lot of this uncertainty comes from imperfect training data, which does not exactly describe the distribution the data were drawn from.

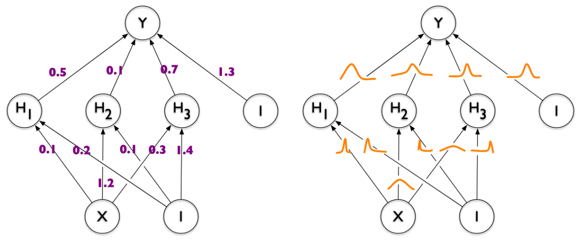


Figure 2. Difference between a traditional NN and the Bayesian one

From a bayesian point of view the idea is to include such uncertainty in deep learning models. This is done by changing each weight from a single deterministic value to a *probability distribution*. We then learn the parameters of this distribution.

Consider a neural network weight w_i. In a standard (deterministic) neural network, this has a single value $\hat{w}_i$, learnt via backpropagation.
In a neural network with weight uncertainty, each weight is represented by a probability distribution, and the *parameters* of this distribution are learned via backpropagation. Suppose, for example, that each weight has a normal distribution. This has two parameters: a mean μ_i and a standard deviation σ_i.

To summarise:

- Classic deterministic NN: $w_i = \hat{w}_i$

- NN with weight uncertainty represented by normal distribution: $w_i \sim N(\hat{\mu}_i, \hat{\sigma}_i)$.

Since the weights are uncertain, the feedforward value of some input x_i is not constant. A single feedforward value is determined in two steps:

- Sample each network weights from their respective distributions – this gives a single set of network weights.

- Use these weights to determine a feedforward value $\hat{y}_i$.

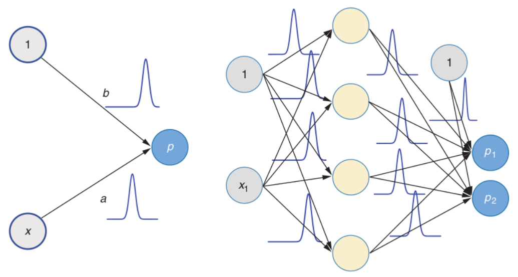


Figure 3. A graphical representation of a Bayesian simple linear regression problem by a simple NN without an hidden layer and only one output node. This provides an estimate for the expected value of the outcome (on the left). In a Bayesian variant of the NN for linear regression, distributions replace the slope (a) and intercept (b). This is also possible for deep networks, yielding a Bayesian neural network (BNN). A simple example of such a network is shown on the right.

Hence, the key question is how to determine the parameters of the distribution for each network weight.

1.3 Bayesian Neural Networks

We can now apply the Bayesian approach described in the previous section to neural networks (NNs).

Figure 3 (on the left) shows the simplest Bayesian networks: **Bayesian linear regression**. Compared to standard probabilistic linear regression, the weights aren't fixed but follow a distribution. There's absolutely no reason that we can't continue to use distributions instead of single weights for a Neural Network. Figure 3 (on the right) also shows such a simple BNN

How to solve a simple linear regression problem like the one in Figure 3? There's an approach called Markov Chain Monte Carlo (MCMC for short). The first MCMC algorithm was the *Metropolis-Hastings algorithm*.
It has the advantage that, given enough computations, it's exact. It also works for small problems, say 10 to 100 variables, but not for larger networks such as DL models with typically millions of weights. There are two alternatives approaches to obtain an approximation for the normalized posterior: one is the ***Variational Inference (VI)*** Bayes; the other is ***Monte Carlo (MC) dropout***. The variational Bayes approach is welded into TFP [1], providing Keras layers to do the VI. MC dropout is a simple approach that can be easily done in standard Keras library as well.

1.4 Variational Inference

In situations where we can't determine the analytical solution for a Bayesian NN or use the MCMC methods, we need to use a technique to approximate the Bayesian model: Variational Inference approach.

The main idea of the Bayes approach in DL is that with BNNs, each weight is replaced by a distribution. Normally, this is quite a complicated distribution, and this distribution isn't independent among different weights. The idea behind the VI Bayes method is that the complicated posterior distributions of the weights are approximated by a simple distribution called *variational distribution*.

Variational Bayes methods approximate the posterior distribution with a second function, called a *variational posterior*. This function has a known functional form, and hence avoids the need to determine the posterior $P(w|D)$ exactly. Of course, approximating a function with another one has some risks, since the approximation may be very bad, leading to a posterior that is highly inaccurate. In order to mediate this, the variational posterior usually has a number of parameters, denoted by θ , that are tuned so that the function approximates the posterior as well as possible.

Often Gaussians are used as parametric distributions (see Figure 4); this is also done by default

[1]TensorFlow Probability (TFP) is a library for probabilistic modeling and inference in TensorFlow. It provides integration of probabilistic models with deep neural networks, gradient-based inference via automatic differentiation, and scalability to large datasets and models via hardware acceleration (e.g., GPUs) and distributed computation.

when using TFP. The Gaussian variational distribution is defined by two parameters: the mean and the variance. Instead of learning a single weight value w, the network has to learn the two parameters of weight distribution: w_μ for the mean of the Gaussian and w_σ for the spread of the Gaussian. Besides the type of the variational distribution that is used to approximate the posterior, we also need to define a prior distribution. A common choice is to use the standard normal, N(0,1), as the prior.

Instead of $P(w|D)$, we assume the network weight has density $q(w|\theta)$, parameterized by θ. $q(w|\theta)$ is known as the *variational posterior*. We want $q(w|\theta)$ to approximate $P(w|D)$, so we want the 'distance' between $q(w|\theta)$ and $P(w|D)$ to be as small as possible.

To get a feeling for the meaning of the different parameters in Figure 4, let's assume a deep BNN. The parameter θ replaces the weights of the non-Bayesian variant of the NN. The parameter θ in a Bayesian network isn't fixed but follows a distribution. Instead of determining the posterior directly, we approximate it with a simple, variational distribution, such $q_\lambda(\theta)$ as a Gaussian (see the bell-shaped density in Figure 4). There are infinitely many Gaussians out there, but these make up only a subgroup of all possible distributions. The job of VI is to tune the variational parameter λ so that $q_\lambda(\theta)$ gets as close as possible to the true posterior $p(\theta|D)$.

This 'difference' between the two distributions is usually measured by the *Kullback-Leibler divergence D_{KL}*.

The Kullback-Leibler divergence between two distributions with densities $f(x)$ and $g(x)$ respectively is defined as

$$D_{KL}(f(x)||g(x)) = \int f(x)\log\left(\frac{f(x)}{g(x)}\right)\mathrm{d}x.$$

Note that this function has value 0 (indicating no difference) when $f(x) \equiv g(x)$, which is the result we expect. We use the convention that $\frac{0}{0} = 1$ here.

Viewing the data D as a constant, the Kullback-Leibler divergence between $q(w|\theta)$ and $P(w|D)$ is hence

$$\begin{aligned}
&D_{KL}(q(w|\theta)||P(w|D)) = \\
&= \int q(w|\theta)\log\left(\tfrac{q(w|\theta)}{P(w|D)}\right)\mathrm{d}w \\
&= \int q(w|\theta)\log\left(\tfrac{q(w|\theta)P(D)}{P(D|w)P(w)}\right)\mathrm{d}w \\
&= \int q(w|\theta)\log P(D)\mathrm{d}w + \int q(w|\theta)\log\left(\tfrac{q(w|\theta)}{P(w)}\right)\mathrm{d}w - \\
&\int q(w|\theta)\log P(D|w)\mathrm{d}w \\
&= \log P(D) + D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}(\log P(D|w))
\end{aligned}$$

where, in the last line, we have used

$$\int q(w|\theta)\log P(D)\mathrm{d}w = \log P(D)\int q(w|\theta)\mathrm{d}w = \log P(D)$$

since $q(w|\theta)$ is a probability distribution and hence integrates to 1. If we consider the data D to be constant, the first term is a constant also, and we may ignore it when minimising the above. Hence, we are left with the function

$$L(\theta|D) = D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}(\log P(D|w))$$

Note that this function depends only on θ and D, since w is an integration variable. This function has a nice interpretation as the sum of:

- The Kullback-Leibler divergence between the variational posterior $q(w|\theta)$ and the prior $P(w)$. This is called the ***complexity cost***, and it depends on θ and the prior but not the data D.

- The expectation of the negative loglikelihood $\log P(D|w)$ under the variational posterior $q(w|\theta)$. This is called the ***likelihood cost*** and it depends on θ and the data but not the prior.

$L(\theta|D)$ is the loss function that we minimise to determine the parameter θ. Note also from the above derivation, that we have

$$\begin{aligned}
&\log P(D) = \\
&\mathbb{E}_{q(w|\theta)}(\log P(D|w)) - D_{KL}(q(w|\theta)||P(w)) + D_{KL}(q(w|\theta)||P(w|D)) \\
&\geq \mathbb{E}_{q(w|\theta)}(\log P(D|w)) - D_{KL}(q(w|\theta)||P(w)) =: ELBO
\end{aligned}$$

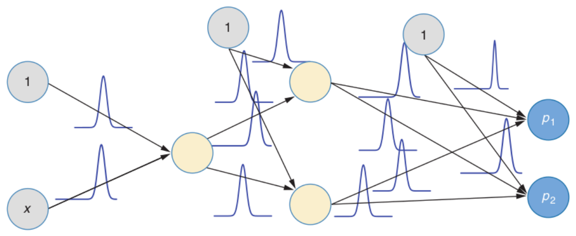


Figure 4. A Bayesian network with two hidden layers. Instead of fixed weights, the weights now follow a distribution.

which follows because $D_{KL}(q(w|\theta)||P(w|D))$ is nonnegative. The final expression on the right hand side is therefore a lower bound on the log-evidence, and is called the *evidence lower bound*, often shortened to **ELBO**. The ELBO is the negative of our loss function, so minimising the loss function is equivalent to maximising the ELBO.

Maximising the ELBO requires a tradeoff between the KL term and expected log-likelihood term. On the one hand, the divergence between $q(w|\theta)$ and $P(w)$ should be kept small, meaning the variational posterior shouldn't be too different to the prior. On the other, the variational posterior parameters should maximise the expectation of the log-likelihood $\log P(D|w)$, meaning the model assigns a high likelihood to the data.

We can use the above ideas to create a neural network with weight uncertainty, which we will call a *Bayesian neural network*. From a high level, this works as follows. Suppose we want to determine the distribution of a particular neural network weight w. The idea is to:

1. Assign the weight a prior distribution with density $P(w)$, which represents our beliefs on the possible values of this network before any training data. This may be something simple, like a unit Gaussian.

2. Assign the weight a variational posterior with density $q(w|\theta)$ with some trainable parameter θ.

3. $q(w|\theta)$ is the approximation for the weight's posterior distribution. We tune θ to make this approximation as accurate as possible as measured by the ELBO.

The remaining question is then how to determine θ. Recall that neural networks are typically trained via a backpropagation algorithm, in which the weights are updated by perturbing them in a direction that reduces the loss function.
We aim to do the same here, by updating θ in a direction that reduces $L(\theta|D)$.

Hence, the function we want to minimise is

$$L(\theta|D) = \\ D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}[\log P(D|w)] \\ = \int q(w|\theta)(\log q(w|\theta) - \log P(D|w) - \log P(w))\mathrm{d}w$$

In principle, we could take derivatives of $L(\theta|D)$ with respect to θ and use this to update its value. However, this involves doing an integral over w, and this is a calculation that may be impossible or very computationally expensive. Instead, we want to write this function as an expectation and use a **Monte Carlo approximation** to calculate derivatives. At present, we can write this function as

$$L(\theta|D) = \mathbb{E}_{q(w|\theta)}(\log q(w|\theta) - \log P(D|w) - \log P(w)) \tag{1}$$

However, taking derivatives with respect to θ is difficult because the underlying distribution and the expectation is taken with respect to depends on θ. One way we can handle this is with the ***reparameterization trick***.

Briefly, the reparameterization trick is a way to move the dependence on θ around so that an expectation may be taken independently of it. For example, suppose $q(w|\theta)$ is a Gaussian, so that $\theta = (\mu, \sigma)$. Then, for some arbitrary $f(w; \mu, \sigma)$, we have:

$$
\begin{aligned}
&\mathbb{E}_{q(w|\mu,\sigma)}[f(w;\mu,\sigma)] \\
&= \int q(w|\mu,\sigma) f(w;\mu,\sigma) \mathrm{d}w \\
&= \int \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(w-\mu)^2\right) f(w;\mu,\sigma)\mathrm{d}w \\
&= \int \frac{1}{\sqrt{2\pi}} \exp\left(-\tfrac{1}{2}\varepsilon^2\right) f(\mu+\sigma\varepsilon;\mu,\sigma)\,\mathrm{d}\varepsilon \\
&= \mathbb{E}_{\varepsilon\sim N(0,1)}\left(f(\mu+\sigma\varepsilon;\mu,\sigma)\right)
\end{aligned}
$$

where we used the change of variable $w = \mu + \sigma\varepsilon$. Note that the dependence on $\theta = (\mu, \sigma)$ is now only in the integrand and we can take derivatives with respect to μ and σ:

$$
\begin{aligned}
&\frac{\partial}{\partial\mu}\mathbb{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) \\
&= \frac{\partial}{\partial\mu}\mathbb{E}_{\varepsilon\sim N(0,1)}(f(w;\mu,\sigma)) \\
&= \mathbb{E}_{\varepsilon\sim N(0,1)}\frac{\partial}{\partial\mu} f(\mu+\sigma\varepsilon;\mu,\sigma) \\
&\frac{\partial}{\partial\sigma}\mathbb{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) \\
&= \frac{\partial}{\partial\sigma}\mathbb{E}_{\varepsilon\sim N(0,1)}(f(w;\mu,\sigma)) \\
&= \mathbb{E}_{\varepsilon\sim N(0,1)}\frac{\partial}{\partial\sigma} f(\mu+\sigma\varepsilon;\mu,\sigma)
\end{aligned}
$$

Finally, note that we can approximate the expectation by its Monte Carlo estimate:

$$
\begin{aligned}
&\mathbb{E}_{\varepsilon\sim N(0,1)}\frac{\partial}{\partial\theta} f(\mu+\sigma\varepsilon;\mu,\sigma) \approx \\
&\textstyle\sum_i \frac{\partial}{\partial\theta} f(\mu+\sigma\varepsilon_i;\mu,\sigma), \qquad \varepsilon_i \sim N(0,1).
\end{aligned}
$$

The above reparameterization trick works in cases where we can write the $w = g(\varepsilon, \theta)$, where the distribution of the random variable ε is independent of θ.

Putting this all together, for our loss function $L(\theta|D) \equiv L(\mu, \sigma|D)$, we have

$$
f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)
$$

$$
\frac{\partial}{\partial\mu}L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial\mu}\right)
$$

$$
\frac{\partial}{\partial\sigma}L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\varepsilon_i + \frac{\partial f(w_i;\mu,\sigma)}{\partial\sigma}\right)
$$

$$
f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)
$$

where $w_i = \mu + \sigma\varepsilon_i$, $\varepsilon_i \sim N(0,1)$. In practice, we often only take a single sample ε_1 for each training point. This leads to the following back-propagation scheme:

- Sample $\varepsilon_i \sim N(0,1)$
- Let $w_i = \mu + \sigma\varepsilon_i$
- Calculate

$$
\nabla_\mu f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial\mu}
$$

$$
\nabla_\sigma f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\varepsilon_i + \frac{\partial f(w_i;\mu,\sigma)}{\partial\sigma}
$$

- Update the parameters with some gradient-based optimiser using the above gradients.

This is how we learn the parameters of the distribution for each neural network weight.

1.5 Monte Carlo Dropout

Variational Inference allows us to fit a Bayesian DL model by learning an approximative posterior distribution for each weight. The default in TFP is to approximate the posterior by a Gaussian. These BNNs have twice as many parameters compared to their non-Bayesian versions, because each weight is replaced by a Gaussian weight distribution that's defined by two parameters (mean and standard deviation).

The idea here is to find a way to use a Bayesian Neural Network with as many parameters as its non-Bayesian counterpart. In 2015, a PhD student, Yarin Gal, was able to show that the dropout method was similar to Variational Inference, allowing us to approximate a BNN.

The traditional dropout approach during training was introduced as a simple way to prevent an NN from overfitting. When doing dropout during training, we set some randomly picked neurons in the Neural Network to zero. We do that in each update run. Mathematically speaking, each neuron has some probability p of being ignored, called the dropout rate. The dropout rate is typically set to be between 0 (no dropout) and 0.5 (approximately 50% of all neurons will be switched off).

Because we actually drop neurons, the weights of all connections that start from the dropped neuron are simultaneously dropped.

In Keras, this can easily be done by adding a dropout layer after a weight layer and giving the dropout the probability p^* as an argument to the `Dropout` Layer. During training, the dropout is often only used in the fully connected layers.
The main idea behind this methodology is that fewer complex features are learned when using dropout. Because dropout forces the Neural Network to deal with missing information, it yields more robust and independent features.

During the prediction step, observation go back to the full Neural Network with fixed weights, but considering only one detail: we need to downweigh the learned weight values to $w^* = p^* w$. This downweighting accounts for the fact that during training each neuron gets, on average, p^* less inputs than in the full Neural Network. The connections are, thus, stronger than these would be if no dropout is applied. During the application phase, there's no dropout and, hence, we downweigh the too strong weights by multiplying those with p^*.

During training, dropout can easily enhance the prediction performance. If we turn it on during testing we can use it as a Bayesian Neural Network!

In a BNN, a distribution of weights replaces each weight. With VI, we use a Gaussian weight distribution with two parameters (mean and standard deviation). When using dropout, we also have a weight distribution, but now the weight distribution is simpler and essentially consists of only two values: 0 or w. The idea is that we can treat many different networks (with different neurons dropped out) as Monte Carlo samples from the space of all available models. To do that we simply apply dropout at test time. Then, instead of one prediction, we get many, one by each model. We can then average them or analyze their distributions

$$p(y|x_{test}, D) = \sum_i p(y|x_{test}, w_i) p(w_i|D) \tag{2}$$

We predict for the same input x from test set T-times a conditional probability distribution (CPD).
For each prediction, we get different CPDs, $(p(y|x_{test}, w_i))$, corresponding to a sampled weight constellation w_i.

The dropout probability p^* isn't a parameter but is fixed when we define the Neural Network. We turn on dropout during training and use the Negative Log-Likelihood loss function, which we minimize by tuning the weights via stochastic gradient descent (SGD).
In Gal's framework, dropout is similar to fitting a Bayesian Neural Network via the Variational Inference method from previous sections but this time; the distribution from Figure 5 is taken instead of the Gaussian usually used in the Variational Inference approach. By updating the w values, we actually learn the weight distributions that approximate the weight posteriors.

It's called MC dropout because for each prediction, we make a forward pass through another

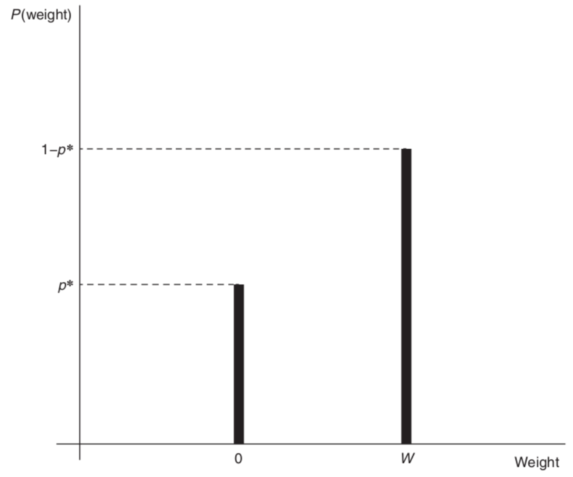


Figure 5. The (simplified) weight distribution with MC dropout. The dropout probability p* is fixed when defining the NN. The only parameter in this distribution is the value of w.

thinned version of the NN, resulting from randomly dropping neurons. You then can combine the dropout predictions to a Bayesian predictive distribution:

$$p(y|x_{test}, D) = \frac{1}{T}\sum_t^T p(y|x_{test}, w_t)$$

This is an empirical approximation to equation 2. The resulting predictive distribution captures the epistemic and the aleatoric uncertainties.

1.6 Classification case study with novel classes

At the beginning of the report, we face the *Elephant in the Office* classification task. When presenting a new input image to a trained classification NN, we get for each class seen during training phase a predicted probability. To predict the Elephant image we used a VGG16 CNN trained on ImageNet, but our model wasn't able to find the elephant in the office (the elephant wasn't among the top five classes either). In this sub-section we want to see, for a simple case study, what are the differences in explaining uncertainties, between a classical CNN and Bayesian Neural Networks. Since we cannot train our BNN using ImageNet, we will use a small built-in dataset provided by keras: **The Fashion MNIST**.

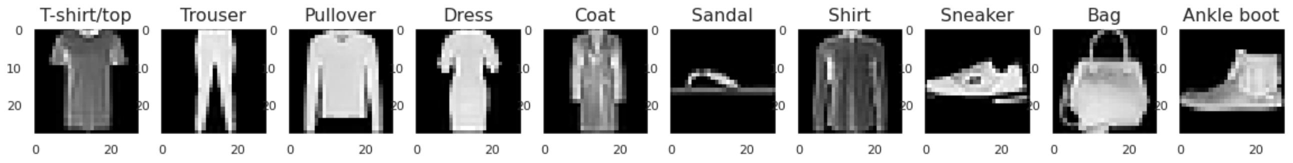
This dataset is a collection of 60'000 images **28x28** in grayscale that represents 10 different categories of clothes. For this task we are going to implement a traditional non-Bayesian neural network and two different types of BNNs, the first one using *Variational Inference* and the second one using the *MC Dropout* approach.

In order to see how these different types of architectures express uncertainty, we will drop all the images from one class (in particular, the *sneaker class*) from the training examples and leave it in the test examples.
When we show a sneaker image to the trained Neural Network it estimates probabilities for the classes on which it was trained. All of these classes are wrong, but the traditional architecture still can't assign a zero probability to all classes because the output needs to add up to one, which is enforced using the softmax layer.

For training we now have 9 classes and we will see how different networks express uncertainties both for known ad unknown classes.
During test time, a traditional probabilistic CNN for classification yields for each input image a multinomial probability distribution. In our case, one that's fitted with nine outcome classes. The probabilities of the k classes provide the parameters for this multinomial distribution.

In the non-Bayesian NN, we have fixed weights, and for one input image, we get one multinomial CPD: $p(y|x,w) = MN(p_1(x,w), ..., p_9(x,w))$. If we predict the same image **T** times, we will always get the same results.

In a Bayesian Neural Network fitted with **VI**, the fixed weights are replaced with *Gaussian distributions*. During test time, we sample from these weighted distributions by predicting the same input image not only once, but T times. For each prediction, we get one multinomial CPD: $p(y|x,w_t) = MN(p_1(x,w_t), ..., p_9(x,w_t))$. Each



time we predict the image, we get a different CPD $p(|y|x, w_t)$, corresponding to the sampled weight constellation w_t.

In the BNN fitted with MC dropout, we replace the fixed weights with binary distributions. For each prediction, we get one multinomial CPD: $p(y|x, w_t) = MN(p_1(x, w), ..., p_9(x, w))$. Each time we predict the image, we get a different CPD $(p(y|x, w_t))$ corresponding to the sampled weight constellation w_i .

Let's look at the results of the network on a known class for two examples first.

In the left panel of Figure 6, we can see that all networks correctly classify the image of a boot. If we take a look at the unknown class image, all predictions in Figure 6 are wrong because, as we said, there was no sneaker images in the training set. But we can see that the BNNs can better express their uncertainty. Of course, the Bayesian networks also predict a wrong class in all of their T runs. But what we see for each of the T runs is that the distributions vary quite a bit. When comparing the VI and MC dropout methods,VI shows more variation. Also, the shapes of the predicted probability distribution look different for MC dropout and VI.

In the case of a traditional, non-Bayesian NN, we get for one image, one CPD. We classify the image to the class with the highest probability: $p_{pred} = max(p_k)$. The CPD only expresses the aleatoric uncertainty, which would be zero if one class gets a probability of one and all other classes get a probability of zero. You can use p_{pred} as a measure for certainty or $-log(p_{pred})$ as a measure for certainty or as a measure for uncertainty, which is the well-known NLL:

$$NLL = -log(p_{pred})$$

Another frequently used measure for the aleatoric uncertainty (using not only the probability of the predicted class) is entropy. There's *no epistemic uncertainty* when working with a non-Bayesian NN. Here's the formula:

$$Entropy : H = -\sum_{k=1}^{9} p_k log(p_k)$$

For each image, we predict T multinomial CPDs:

$$MN(p_1(x, w_t), ..., p_9(x, w_t))$$

From a Bayesian point of view approach, for each class k, we can determine the mean probability

$$p_k^* = \frac{1}{T} \sum_{t=1}^{T} p_{kt}$$

We classify the image to the class with the highest mean probability

$$p_{pred}^* = max(p_k^*)$$

In the literature, there's no consensus on how to best quantify the uncertainty that captures both epistemic and aleatoric contributions; in fact, that's still an open research question (at least if we have more than two classes). Note that this mean probability already captures a part of the epistemic uncertainty because it's determined from all T predictions. Averaging causes a shift to less extreme probabilities away from one or zero. We could use $-log(p_{pred}^*)$ as the uncertainty:

$$NLL^* = -log(p_{pred}^*)$$

The entropy based on the mean probability values, p_k^* , averaged over all T runs is an even better-established uncertainty measure. But also, you can use the total variance (sum of the variances for the individual classes) of the multidimensional probability distribution to quantify the uncertainty:

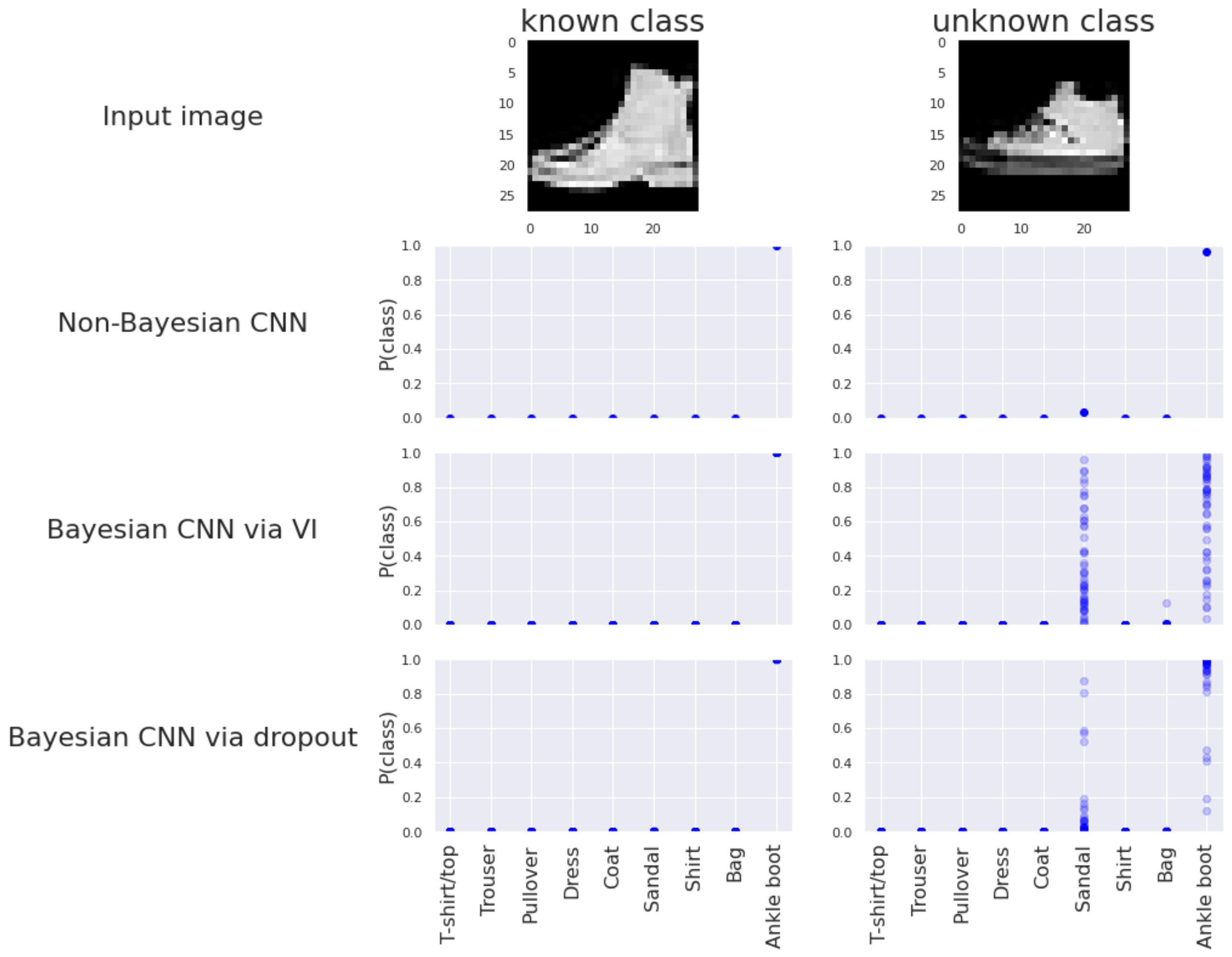


Figure 6. Upper panel: images presented to train CNNs included an image from the known class *boots* (left) and an image from the unknown class *sneakers* (right). Second row of plots: corresponding predictive distributions resulting from non-Bayesian NN. Third row of plots: corresponding predictive distributions resulting from BNN via VI. Fourth row of plots: corresponding predictive distributions resulting from BNN via MC dropout.

- $Entropy^* : H^* = -\sum^9 p_k^* log(p_k^*)$
- Total Variance: $V_{TOT} = \sum^9 var(p_k) = = \sum^9 \frac{1}{T} \sum^T (p_{k_t} - p_k^*)^2$

Since we can calculate the uncertainty of every prediction with a Bayesian Neural Network, we could filter out the observations with high uncertainty and predict a new class, the sneaker class, which is something we can't do with traditional NNs.

2 PROBABILISTIC DEEP LEARNING MODELS

Many real-world data like sound samples or images come from complex and high-dimensional distributions.

Deep probabilistic models are probabilistic models that incorporate deep neural network components which capture complex non-linear stochastic relationships between the random variables.

Unsupervised and semi-supervised learning methods are often based on generative models which are probabilistic models that express hypotheses about the way in which data may have been generated.

Probabilistic Generative Models (PGMs) have emerged as a broadly useful approach to specifying generative models. Deep generative models (DGMs) are PGMs that employ deep neural networks for parameterizing the models. In particular, prescribed DGMs are those that provide an explicit parametric specification of the probability distribution of the observed variable x, specifying the likelihood function $p_\theta(x)$ with parameter θ. DGMs are among the most widely used deep probabilistic models.

One way to model complex distributions are mixtures of simple distributions such as Normal, Poisson, ... Mixture models are used in state-of-the-art networks like Google's parallel WaveNet or OpenAI's PixelCNN++ to model the output.

There is a way to model these complex distributions: the so-called *Normalizing Flows*.

2.1 Bijectors and Normalizing Flows

Learning the representations of complex data representations is a fundamental problem in machine learning. The importance of this task lies in the availability of vast amounts of unstructured and unlabeled data that can be only understood through unsupervised learning.

We can find applications of this tasks in density estimation, outlier detection, text summarization, data clustering, bioinformatics, DNA modeling, etc.

Normalizing Flows (NFs) allow us to learn a transformation from a simple distribution to a complicated distribution. In simple cases, this can be done with a statistical method called the *change of variable method*.

But *how does one set up and fit a flexible high-dimensional distribution?* If we think, for example, of color images with $256 \times 256 \times 3 =$ 195,840 pixels defining a 195,840-dimensional space where each image can be represented by one point. If we pick a random point in this space, then we would most probably get an image that looks like noise. *How can we learn the 195'840-dimensional distribution from which facial images can be drawn?*

In a nutshell, a Normalizing flows learns a transformation *(flow)* from a simple high-dimensional distribution to a complex one. In a valid distribution, probabilities need to sum up to 1 in the discrete case or the integral needs to be 1 in the continuous case, and these need to be *normalized*. NFs are probabilistic models that we can fit with the same Maximum Likelihood approach.

The basic idea is that a Normalizing Flow can fit a complex distribution without picking in advance an appropriate distribution family or setting up a mixture of several distributions.

A probability density allows us to sample from that distribution. In the case of the facial image distribution, we can generate facial images from the distribution. The generated faces aren't the ones from the training data. For this reason, Normalizing Flows fall under the class of *generative models*.

In contrast to GANs and VAEs, NFs are probabilistic models that really learn the probability distribution and allow for each sample to determine the corresponding probability (likelihood). Normalizing Flow models do not need to put noise on the output and thus can have much more powerful local variance models and they are much easier to converge when compared to GANs and VAEs. Furthemore, the training process of a flow-based model is very stable compared to GANs' training, which requires careful tuning of hyperparameters of both generators and discriminators.

Say we've used Normalizing Flow to learn the distribution of facial images, and we have an image x, then we can ask the NF via *p(x)* what's the probability of that image? In novelty detection, for example, we want to find out if a data point is from a certain distribution or if it's an original (novel) data point.

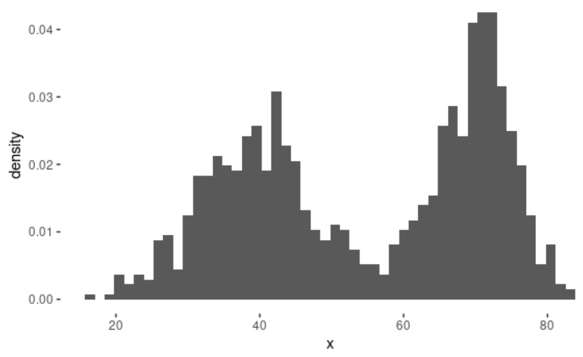


Figure 7

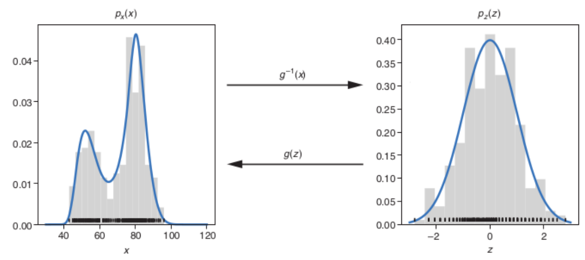


Figure 8

Now let's say we have data coming from the distribution in the Figure 7: in this simple case we can come up with a mixture distribution of, for example, two Gaussians. This works for quite simple distributions, such as the one shown in Figure 7, but for really high dimensional and complex distributions, this approach breaks down.

Provided that we don't know how to infer the density function of the Figure 7, what do we do now?

Figure 8 shows how does Normalizing Flows work. In this case the complicated $p_x(x)$ density distribution is transformed to an easy Gaussian with the Probability Density Function $p_z(z) = N(z;0,1)$.

The transformation function $x = g(z)$ transfers between the easy Gaussian in z and the complicated function in x. Briefly, the idea is, starting from a simple distribution (like a gaussian one) transform it so that at the end, data looks like it's coming from a complicated distribution.

The main task of the NFs is to find these transformations: $g(z)$ and $g^{-1}(x)$. We assume for a moment that we've found such a function pair: g and g^{-1} . First, NF should enable us to sample from the complicated function $p_x(x)$, allowing for applications to generate new, realistic-looking images of faces. Second, it should allow us to calculate the probability $p_x(x)$ for a given x, allowing for applications like novelty detection.

Since we don't know $p_x(x)$ we can't directly sample x from $p_x(x)$, instead we can draw a sample from the simple distribution $p_z(z)$ which is a Gaussian.

Then we apply the transformation function g to get the corresponding sample $x = g(z)$. However it's not possible for all functions g to find an inverse function g^{-1}.

If a function g has an inverse function g^{-1} , then g is called *bijective*. Some functions such as $g(z) = z^2$ are only bijective on a limited range of data; here, for example, for positive values.

In simple words, normalizing flows is a series of simple functions which are invertible, or the analytical inverse of the function can be calculated.

From the Figure 9, it can be seen that the normalizing flows transform a complex data point such as MNIST Image to a simple Gaussian Distribution or vice-versa. Normalizing flows start from a simple distribution and approximate a complex distribution. They do this by transforming the initial distribution multiple times with some functions until the distribution gets complex enough.

2.1.1 The change of variable technique for probabilities

We can see how to use the NF method in one dimension. This is what statisticians call the *change of variable technique*, which is used to properly transform distributions. It's the core method of all NFs, where several transformation layers are stacked to a deep Normalizing Flow model.

To explain what's going on in a single layer of a NF model, we start with the transformation of a one-dimensional distribution.

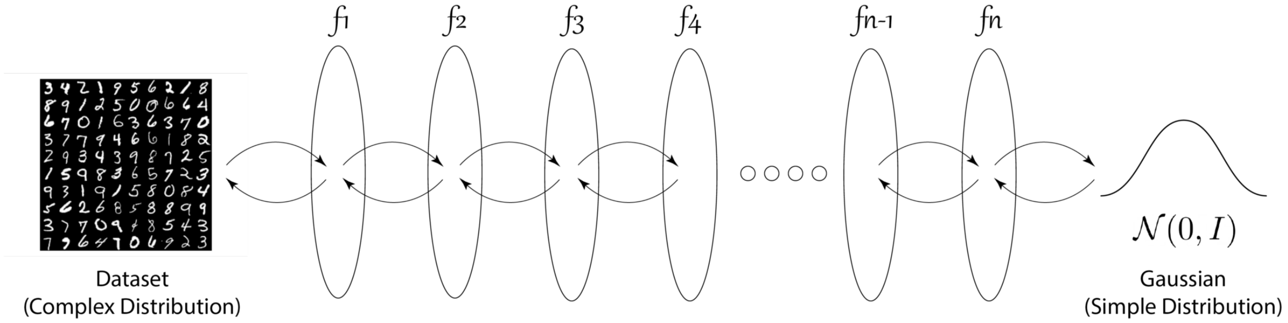
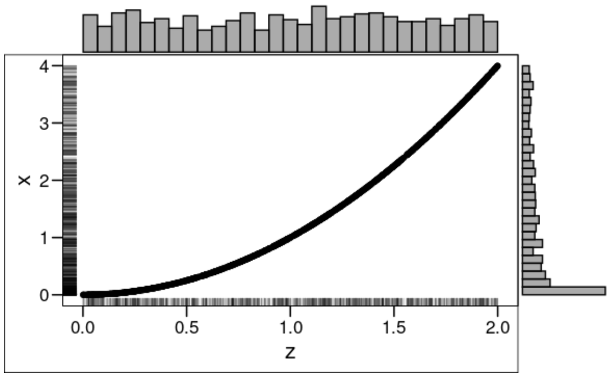


Figure 9

To code such NF models, we use `TFP` and especially the `TFP bijector` package. All TFP bijector classes are about bijective transformations, which apply the change of variables technique to correctly transform probability distributions.

Consider the transformation $x = g(z) = z^2$ and choose z to be uniformly distributed between 0 and 2. The function satisfies for the picked range of z. But now, if we work with the uniformly distributed z between 0 and 2, how does the distribution of $x = g(z) = z^2$ look?



When we apply the square transformation to uniformly distributed samples we can see that ticks are denser in the region around 0 than those are at around 4.

With this procedure, it becomes clear that a transformation function can squeeze samples together in regions where the transformation function is flat and move samples apart in regions where it's steep.
This intuition also implies that linear functions (with constant steepness but different offsets) don't change the shape of the distribution, only the values. Therefore, we need a non-linear transformation function if we want to go from a simple distribution to a distribution with a more complex shape.
Another important property of the transformation function g is that it needs to be monotone in order to be bijective.

We need a formula to describe the transformation. Strictly speaking, $p_z(z)$ is a probability density.
All probability densities are normalized, meaning the area under the density is 1. When using a transformation to go from one distribution to another distribution, this normalization is preserved; hence, the name "normalizing flow." We don't lose probability; it's like a conservation of mass principle.

To come from a probability density value $p_x(x)$ to a real probability for values close to x, we have to look at an area under the density curve $p_x(x)$ within a small interval with the length dx. We get such a probability by multiplying $p_x(x)$ with dx: $p_x(x)dx$. The same is true for z, where $p_z(z)dz$ is a probability. The shaded areas under the curve need to be the same.

From this we get the equation:

$$p_z(z)|dx| = p_x(x)|dx|$$

This equation ensures that no probability is lost during the transformation (the mass is conserved).

Solving this equation we obtain:

$$p_x(x) = p_z(g^{-1}(x))|g'(g^{-1}(x))|^{-1}$$

where $z = g^{-1}(x)$

Normalising Flows, as we said, are a class of models that exploit the change of variables formula to estimate an unknown target data density.

Suppose we have data samples

$$\mathscr{D} := \{x^{(1)}, \ldots, x^{(n)}\}$$

, with each $x^{(i)} \in \mathbb{R}^d$, and assume that these samples are generated i.i.d. from the underlying distribution p_X.

A normalising flow models the distribution p_X using a random variable Z (also of dimension d) with a simple distribution p_Z (e.g. an isotropic Gaussian), such that the random variable X can be written as a change of variables $X = f_\theta(Z)$, where θ is a parameter vector that parameterises the smooth invertible function f_θ.

The function f_θ is modelled using a neural network with parameters θ, which we want to learn from the data. An important point is that this neural network must be designed to be invertible, which is not the case in general with deep learning models. In practice, we often construct the neural network by composing multiple simpler blocks together. In TensorFlow Probability, these simpler blocks are the bijectors.

In order to learn the optimal parameters θ, we apply the principle of maximum likelihood and search for θ_{ML} such that

$$\begin{aligned}\theta_{ML} &:= \arg\max_\theta P(\mathscr{D}; \theta)\\ &= \arg\max_\theta \log P(\mathscr{D}; \theta).\end{aligned}$$

In order to compute $\log P(\mathscr{D}; \theta)$ we can use the change of variables formula:

$$\begin{aligned}P(\mathscr{D}; \theta) &= \prod_{x \in \mathscr{D}} p_Z(f_\theta^{-1}(x)) \cdot \left|\det J_{f_\theta^{-1}}(x)\right|\\ \log P(\mathscr{D}; \theta) &= \sum_{x \in \mathscr{D}} \log p_Z(f_\theta^{-1}(x)) + \log \left|\det J_{f_\theta^{-1}}(x)\right| \qquad (7\end{aligned}$$

The term $p_Z(f_\theta^{-1}(x))$ can be computed for a given data point $x \in \mathscr{D}$ since the neural network f_θ is designed to be invertible, and the distribution p_Z is known.

The term $\det J_{f_\theta^{-1}}(x)$ is also computable, although this also highlights another important aspect of normalising flow models: they should be designed such that the determinant of the Jacobian can be efficiently computed.

The log-likelihood is usually optimised as usual in minibatches, with gradient-based optimisation methods.

While flow-based models come with their advantages, they also have some shortcomings as follows:

- Due to the *lackluster* performance of flow models on tasks such as density estimation, it is regarded that they are not as expressive as other approaches.
- One of the two things required for flow models to be *bijective* is the volume preservation over transformations, which often leads to very high dimensional latent space, which is usually harder to interpret.
- The samples that are generated through flow-based models are not as good when compared to GANs and VAEs.

3 VARIATIONAL AUTOENCODERS VS. AUTOENCODERS

In the last years, *deep learning based generative models* have gained more and more interest due to some improvements in the field.

In contrast to the more standard uses of neural networks as regressors or classifiers, **Variational**

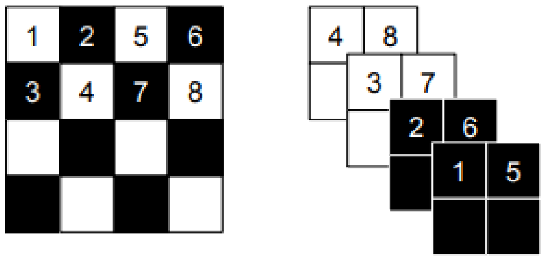


Figure 10. How does a Squeeze function works?

Autoencoders (VAEs) are powerful generative models, now having applications as diverse as from generating fake human faces, to producing purely synthetic music.

The variational autoencoder (VAE) is a prescribed DGM realized using an autoencoder neural network that consists of an encoder network and a decoder network, which encodes a data sample to a latent representation and generates data samples from the latent space, respectively.
Both networks are jointly trained using variational learning. Variational learning uses the variational lower bound of the marginal log-likelihood as the single objective function to optimize both the generator network and the auxiliary inference network.
For variational autoencoders, the encoder model is sometimes referred to as the **recognition model** whereas the decoder model is sometimes referred to as the **generative model**.

When using generative models, we could simply want to generate a random, new output, that looks similar to the training data. But more often, we would like to alter, or explore variations on data we already have, and not just in a random way either, but in a desired, specific direction. This is where VAEs work better than any other method currently available.

In an autoencoder, the decoder samples directly from latent variables. **Variational Autoencoders (VAE)**, differ in that the sampling is taken from a distribution parameterized by the latent variables.

In a nutshell, a VAE is an autoencoder whose encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing us to generate some new data.
To be clear, let's say we have an autoencoder with two latent variables and we draw samples randomly and get two samples of 0.4 and 1.2. We then send them to the decoder for image generation.

In a VAE, these samples don't go to the decoder directly. Instead, they are used as a **mean** and a **variance** of a **Gaussian Distribution**, and we draw samples from this distribution to be sent to the decoder for image generation. Gaussian distributions in VAEs are assumed to be **i.i.d.** and therefore covariance matrix will be diagonal.

What we hope to achieve is to create a *nicely* distributed latent space where latent variables' distributions for different data classes are as follows:

- Evenly spread so we have a better variation to sample from
- Overlap slightly with each other to create a continuous transition

But why we can't use an autoencoder for image generation? What is the link between autoencoders and content generation? After an autoencoder has been trained we could think that, if the latent space is regular enough, we could take random points from that latent space and decode it to get a new content.
However the regularity of the latent space for autoencoders is a difficult point that depends on the distribution of the data in the initial space, the dimension of the latent space and the architecture of the encoder. Why does this happen? A dimensionality reduction with no reconstuction loss often comes with a price: the lack of interpretable and exploitable structures in the latent

space (lack of regularity). The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, may not be continuous.

To illustrate this point, suppose we have an encoder and a decoder powerful enough to put any N initial training data onto the real axis and decode them without any reconstruction loss. In such case, the high degree of freedom of the autoencoder that makes possible to encode and decode with no information loss (despite the low dimensionality of the latent space) leads to a severe overfitting implying that some points of the latent space will give meaningless content once decoded.
This lack of structure among the encoded data into the latent space is pretty normal: the autoencoder is solely trained to encode and decode with as few loss as possible, no matter how the latent space is organised.

For example, training an autoencoder on the **MNIST** dataset, and visualizing the encodings from a 2D latent space reveals the formation of distinct clusters as we can see in Figure 12. This makes sense, as distinct encodings for each image type makes it far easier for the decoder to decode them. This is fine if we're just replicating the same images. But when we want to build a generative model, we don't want to prepare to replicate the same image we put in. We want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space. If the space has discontinuities (eg. gaps between clusters) and we sample/generate a variation from there, the decoder will simply generate an unrealistic output, because it has no idea on how to deal with that region of the latent space. During training, it never saw encoded vectors coming from that region of latent space.

So, in order to be able to use the decoder of our autoencoder for generative purpose, we have to be sure that the latent spaces are, by design, continuous, allowing easy random sampling and interpolation.

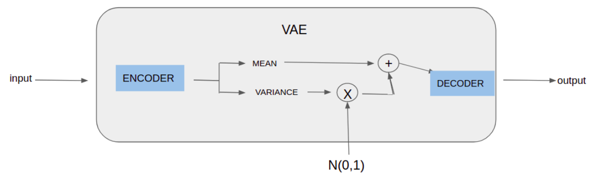
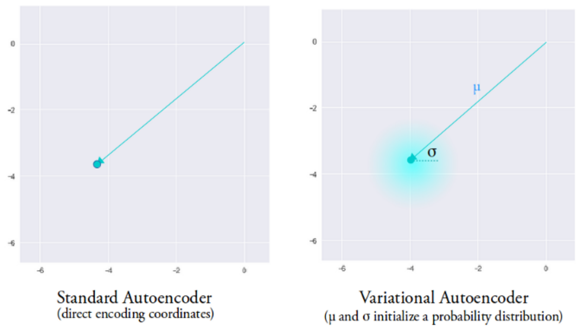


Figure 11. Gaussian Sampling in a VAE

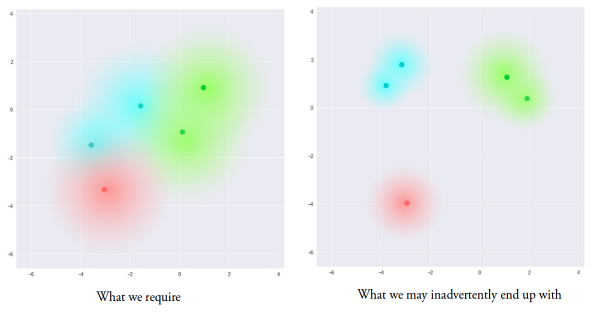
As we said, when we train an autoencoder, the encoded latent variables go straight to the decoder. With a VAE, there is an additional sampling step between the encoder and the decoder. The encoder produces the mean and variance of Gaussian distributions as latent variables, and we draw samples from them to send to the decoder. Here comes a problem: sampling is **not** back-propagatable and therefore is not trainable. To solve this, we can use a *reparameterization trick* where we cast the Gaussian random variable $N(mean, variance)$ into $mean + sigma * N(0,1)$. The sampling becomes an affine transformation and the error could be backpropagated from the output back to the encoder.

The sampling from a standard Gaussian Distribution **N(0, 1)** can be seen as input to the VAE, and do not need to backpropagate back to inputs. This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.



Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the 'area', how much from the mean the encoding can vary. As encodings are generated at random from anywhere inside the 'circle' (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well. This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the encoding of the same input during training.

The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples. Ideally, we want overlap between samples that are not very similar too, in order to interpolate between classes. However, since there are no limits on what values vectors μ and σ can take on, the encoder can learn to generate very different μ for different classes, clustering them apart, and minimize σ, making sure the encodings themselves don't vary much for the same sample (that is, less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the training data.



What we ideally want are encodings, all of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of new samples.

The way VAE do this is by putting in some regularization to encourage the Gaussian distribution *to look like* $N(0,1)$. In other words, we want them to have a *mean close to 0* to keep them close together, and *variance close to 1* for a better variations to sample from. This is done by using **Kullback-Leibler distance (KLD)**.

The loss function that is minimised when training a VAE is composed of a 'reconstruction term' (on the final layer), that tends to make the encoding-decoding scheme as performant as possible, and a 'regularisation term' (on the latent layer), that tends to regularise the organisation of the latent space by making the distributions returned by the encoder close to a standard normal distribution.

That regularisation term is expressed as the Kullback-Leibler divergence between the returned distribution and a standard Gaussian. We can notice that the Kullback-Leibler divergence between two Gaussian distributions has a closed form that can be directly expressed in terms of the means and the covariance matrices of the the two distributions.

The other term is what we typically use for simple autoencoders to compare the generated images with the label images. This is called the ***reconstruction loss***, which measures the difference in reconstructed images with the target image. This can be either **Binary-crossentropy** or **Mean Squared Error**.

The total loss is made up of the Kullback-Leibler loss and the reconstruction loss. Comparatively, the distribution of latent variables is all over the place at the beginning:

$$loss = ||x - x_{reconstructed}||^2 + KL[N(\mu_x, \sigma_x), N(0,1)]$$

To understand deeper how does this loss work, we consider the MNIST dataset, a well-known dataset made of grayscale handwritten digits. As we can see in the left-most Figure 12, focusing only on reconstruction loss does allow us to separate out the classes (in this case, MNIST digits) which should allow our decoder model the ability to reproduce the original handwritten digit, but

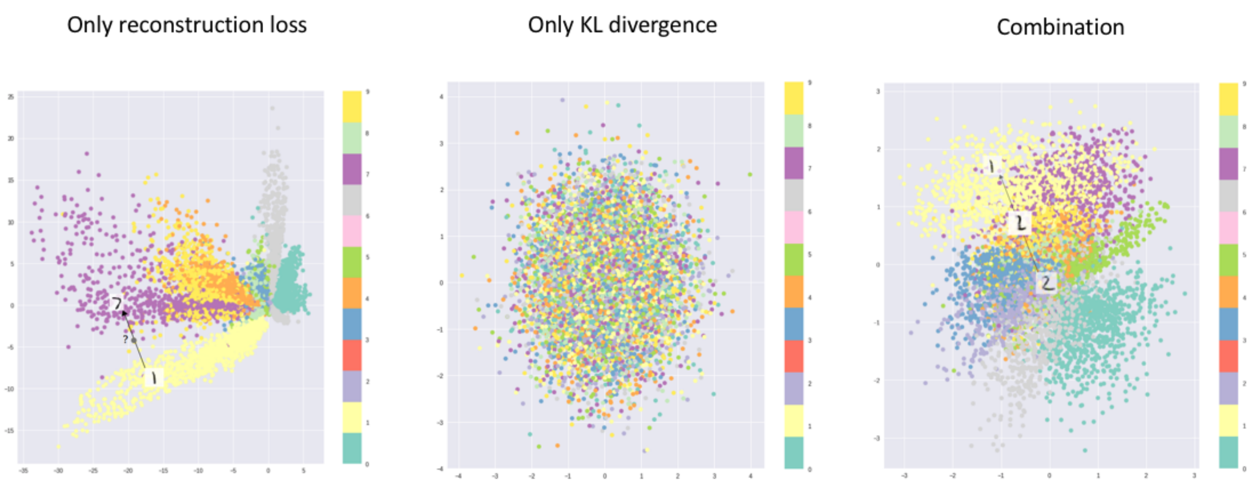


Figure 12. 2D representation of MNIST observations in Latent Space using (from Left to Right): Reconstruction Loss, KL divergence, A combination of both

there's an uneven distribution of data within the latent space. In other words, there are areas in latent space which don't represent any of our observed data.

On the flip side, if we only focus on ensuring that the latent distribution is similar to the prior distribution (through our KL divergence loss term), we end up describing every observation using the same unit Gaussian, which we subsequently sample from to describe the latent dimensions visualized. This effectively treats every observation as having the same characteristics; in other words, we've failed to describe the original data.

However, when the two terms are optimized simultaneously, we're encouraged to describe the latent state for an observation with distributions close to the prior but deviating when necessary to describe salient features of the input.

By sampling from the latent space, we can use the decoder network to form a generative model capable of creating new data similar to what was observed during training. Specifically, we'll sample from the prior distribution p(z) which we assumed follows a unit Gaussian distribution.

4 SUMMARY AND CONCLUSION

Probabilistic deep learning is deep learning that accounts for uncertainty, both model uncertainty and data uncertainty. It is based on the use of probabilistic models and deep neural networks. As background information, we discussed the probabilistic foundation of probabilistic deep learning: uncertainty, probabilistic models, and Bayesian inference.

We distinguish two approaches to probabilistic deep learning: probabilistic neural networks and deep probabilistic models. For probabilistic neural networks, we discussed Bayesian neural networks ; for deep probabilistic models we discussed Normalizing Flows and Variational Autoencoders.

With the availability of libraries for probabilistic modeling and inference, such as TensorFlow Probability, it is hopeful that probabilistic deep learning will become more widely used in the near future so that uncertainty, both model uncertainty and data uncertainty, in deep learning will routinely be accounted for and quantified.

REFERENCES

[1] Chang, D. T. Probabilistic deep learning with probabilistic neural networks and deep probabilistic models. *arXiv preprint arXiv:2106.00120* (2021).

[2] Morgan, P. Probabilistic programming in tensorflow (2018).

[3] Rezende, D. & Mohamed, S. Variational inference with normalizing flows. In *International conference on machine learning*, 1530–1538 (PMLR, 2015).

[4] Blei, D. M., Kucukelbir, A. & McAuliffe, J. D. Variational inference: A review for statisticians. *J. Am. statistical Assoc.* **112**, 859–877 (2017).

[5] Dürr, O., Sick, B. & Murina, E. *Probabilistic deep learning: With python, keras and tensorflow probability* (Manning Publications, 2020).

[6] Gal, Y. & Ghahramani, Z. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, 1050–1059 (PMLR, 2016).

[7] Kingma, D. P. & Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[8] Shafkat, I. Variational autoencoders. *https://medium.com/@irhumshafkat* (2020).