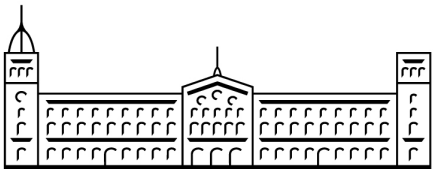
Master's Thesis

Mixture Density Networks for distribution and uncertainty estimation



Axel Brando Guillaumes

Master in Artificial Intelligence

Facultat de Matemàtiques i Informàtica (UB)

Universitat de Barcelona

Advisor: Jordi Vitrià Marca

Co-advisor: Santi Seguí Mesquida

Departament de Matemàtiques i Informàtica de la UB

Barcelona, 1 de Febrer del 2017

Abstract

The deep learning techniques have made neural networks the leading option for solving some computational problems and it has been shown the production of the state-of-the-art results in many fields like computer vision, automatic speech recognition, natural language processing, and audio recognition. In fact, we may be tempted to make use of neural networks directly, as we know them nowadays, in order to make predictions and solve many problems, but if the decision that has to be taken is of high risk. For instance, we could have a problem regarding the control of a nuclear power plant or the prediction of the shares evolution in the market; in this case, it would be important to look for methods that allowed us to add more information concerning the certainty of those predictions.

This Master's thesis is divided into three parts: Firstly, we will analyse the state-of-the-art regarding Mixture Density Network models to predict an entire probability distribution for the output and we will develop an implementation to give solutions for many of the numerical stability problems that characterise this type of models. Secondly, in order to propose an initial solution for the uncertainty problems introduced above, we will focus on the extraction of a confidence factor by using neural network outputs of a problem for which we are only interested in the prediction of something if we have a minimum certainty about the prediction we made. In order to do it, we will compile the current literature methods to measure uncertainty through Mixture Density Networks and we will implement all of these works. Consequently, we are going to to go into detail about the concept of uncertainty and we will see to what extent we are able to propose

a solution by using neural network models for the different aspects that include such concept. Finally, the third part will refer to several proposals to measure the confidence factor obtained with the use of Mixture Density Network concerning the problem proposed.

After all the work, our goals will be achieved: we are going to make a stable implementation for all the problems that we have proposed for Mixture Density Networks and we will publish it publicly in our GitHub repository[9]. We will be able to implement the state-of-the-art methods that will allow us to obtain a confidence factor and finally we will be able to propose a method that obtains the expected results regarding the parameters that represent the confidence factor.

Contents

Chapter 1

Introduction

Currently there are many ways to explain the concepts concerning Deep Learning techniques. Some authors like Nielsen [33] explain the different techniques following the chronological evolution of the facts and, therefore, they begin introducing the concepts of Perceptron, the Multi-Layer Perceptron and the first methods of optimisation that were applied to these models. Other authors like MILA lab from University of Montreal [28] prefer to start from a more probabilistic point of view defining a logistic regression that naturally involves minimising a loss function. This allows the definition of the concept of loss function and how to modify the parameters in order to obtain a better solution, and this connects well with the concept of Multi-Layer Perceptron viewed as a logistic regression classifier.

In this introduction we have decided to explain some important layers from Deep Learning as they were building blocks that they can be combined to achieve our goals in this project. This point of view is consistent with most widely used neural networks frameworks as well as it helps to the understanding of the implementation[9] we coded to carry out the aim of this project.

1.1 Multi-Layer Dense Neural Networks

An artificial neuron is a mathematical function which constitutes the basic computing unit for artificial neural networks. As we can see in Figure 1.1, first

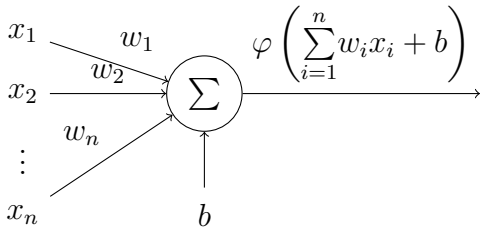


Figure 1.1: Representation of a single neuron model. Note that this generic graph does not impose that this neuron is of the Perceptron type.

the artificial neuron receives n weighted inputs $(w_1x_1, \ldots, w_nx_n)$ and the bias b. Then, it sums them and finally it applies a non-linear function known as *activation function* φ to produce an output. Therefore, to compute an output of a single neuron is like forming a linear combination according to its input and weights and then passing through some activation functions.

Due to historical reasons, in a particular case in which we have the activation function that is a linear threshold function (with a binary output), this neuron is called Perceptron [37].

From this point onwards, when we refer to a *layer* of a neural network we will be talking about the cluster of neurons that forms, specifically, a column in the graph of Figure 1.2, i.e., neurons that are at the same level of depth. In addition, all layers between the input layer and output layer will be called hidden layers.

As we introduced, MILA lab [28] explains that a Multi-Layer Perceptron (MLP) can be viewed as a logistic regression classifier where the input is transformed using a learnt non-linear transformation $h(\boldsymbol{x})$ that constitutes the hidden layer of our neural network (as we can see in Figure 1.2). This transformation projects the input data into a space where it become linearly separable.

The computations performed by a Multi-Layer neural network with a single hidden layer with element-wise non-linear activation function φ_i to the input vector $\boldsymbol{x}$ and the hidden output can be written in a matrix form as

$$\boldsymbol{y} = \varphi_2\left(W_2\varphi_1\left(W_1\boldsymbol{x} + b_1\right) + b_2\right)$$

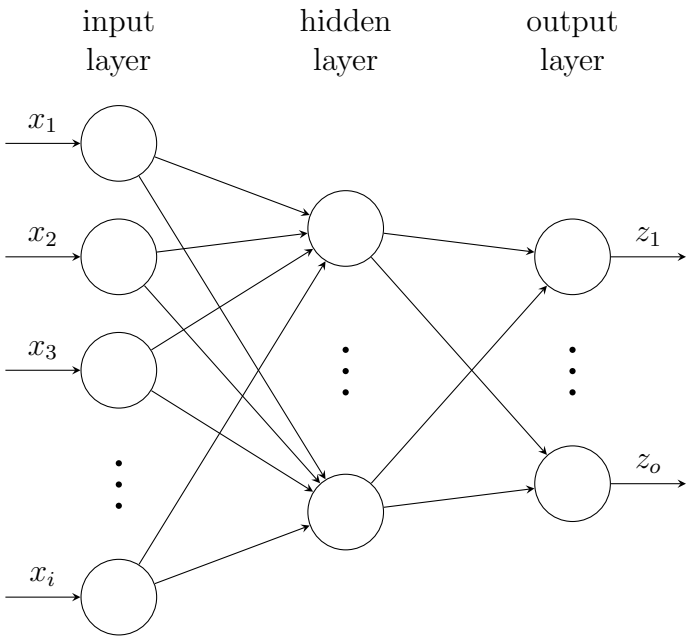


Figure 1.2: Representation of a Multi-Layer model. This generic graph does not impose that its neurons are Perceptron neurons.

where $W_i, \boldsymbol{b}_i$ are the weights matrix and bias vector of the ith-layer.

Following the universal approximation theorem [15] we can state that a neural network with a single hidden layer containing a finite number of neurons (i.e., a multi-layer Perceptron) is enough to approximate any continuous function on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.

As a final point, it is important to know that the type of layers of this Multi-Layer neural network presented above are also called *dense layers*, *fully-connected layers* or even *inner product layers*.

1.2 Recurrent Neural Networks

When humans use their frontal lobe, they are able to project future consequences resulting from current actions, they can choose between good and bad actions (or better and best) and they can determine the similarities and the differences between things or events.

There are many problems that require a knowledge of previous events. One example could be concerning the colour light changes in a traffic light: For instance, if we looked at a traffic light and in this right moment the light on was in orange, our aim would be to know the next colour light. The different position of the lights of the traffic lights do not matter: We are only focusing on the detection of the next colour that will appear taking into account that we know that the previous colour light on was the orange. In most cities we know that the answer would be that the red light will turn on. In that case our experience solves this situation but if we try to solve this problem using the neural network models explained before we cannot find a clear solution. This occurs because the solution has a temporary dependence. Our learning process is based on a temporary sequence. There is a sequence of events and, in order to solve the problem, we must learn at a sequence level.

We need a way to address these problems. An, as usual in Artificial Intelligence field, we get the inspiration again from the use of our biological models in order to design a class of artificial neural network, where connections between units form a directed cycle (as we can see in Figure 1.3), called Recurrent Neural Networks (RNN). These connections create an internal state of the network that allows it to exhibit a dynamic temporary behaviour.

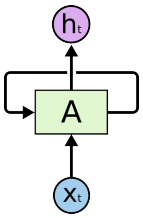


Figure 1.3: A simple Recurrent neural network example. Image extracted from [13]

We have a lot of knowledge in no recurrent neural networks in order to obtain good parameters. Thus, it would be important to find a way to transfer this knowledge to this new type of neural networks. A naive idea was to unroll the recurrent neural network in multiple copies of the same network, as we can show in Figure 1.4, transmitting information from hidden layers A to the next copy of

these layers A in order to create Memory.

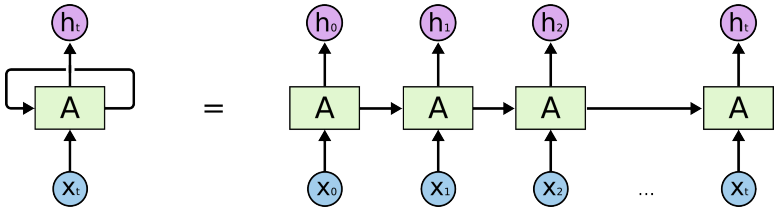
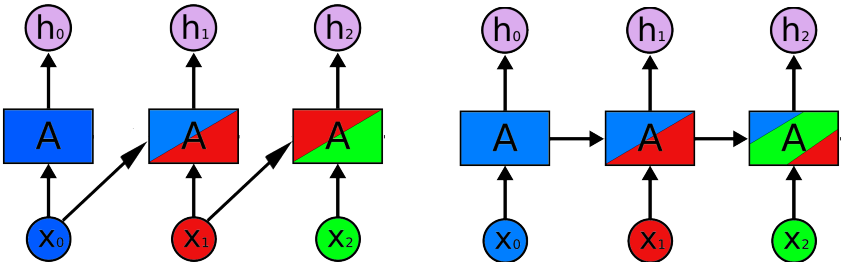


Figure 1.4: An unrolled Recurrent neural network. Image extracted from [13]

However, as Trask [1] explains, there could be different ways of understanding memory and combine our input data at the current time-step with the previous time-step information. For example, we can obtain our previous time-step information saving the previous input or saving the previous hidden layer and the result is totally different. In order to visualise this effect, we will colour 3 time-steps of the previous Figure 1.4 where each colour will represent the effect of each input in the hidden layer A.



(a) 3 time-steps visualisation of Memory using input recurrence

(b) 3 time-steps visualisation of Memory using hidden layer recurrence

Figure 1.5: Images modified from original image extracted from [13]

As we can see in Figure 1.5a, on the one hand, if we use the *input recurrence* we will only remember the current and the previous input but, on the other hand, if we use the *hidden layer recurrence* we will learn a mixture of all previous inputs, as it is shown in Figure 1.5b. In order to understand why this is better we could use an example explained in [1]: we want to deduce the word after "*I love*" in this given sequence but the text had the statements "*I love you*" and "*I love*

carrots". If our aim is to predict the decision that will be made between these two options and we do not know more information than input information (the *input recurrence* case), the neural network does not have enough information to decide. In contrast, if the neural network has more information about the context it could change. At first sight, the *hidden layer recurrence* could be understood as a way to try to remember everything that the neural network saw but in practice there is a limitation to go back just a few steps.

According to Karpathy [27], another point of view of what recurrent neural networks are is that they allow us to operate over sequences of vectors: Sequences in the input, in the the output, or in both cases. As it can be seen in Figure 1.6, there can be five possible cases to analyse:

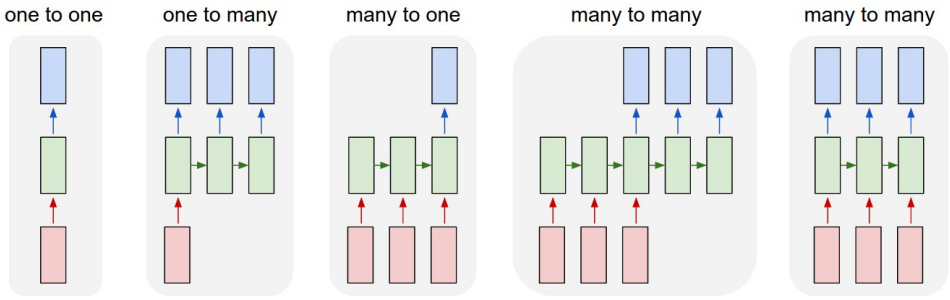


Figure 1.6: Image and examples extracted from [27]

1. One to one: When no recurrent neural network is used. (An example of this application could be a classification of images).

2. One to many: When there is a sequence of outputs (Commonly used in captioning; it takes an image and outputs a sentence of words).

3. Many to one: When there is a sequence of inputs (An example of its use could be the analysis of the positive/negative feelings given in a sentence).

4. Many to many: When there is a sequence of inputs and a sequence of outputs (Used, for example, in machine translators: it reads a sentence in one language and then it outputs a sentence in another one).

5. Synced sequence: When there is a synced sequence of inputs and outputs (Used in video classification where we wish to label each frame of the video).

1.2.1 How to develop a RNN: The Long-Term Dependencies Problem

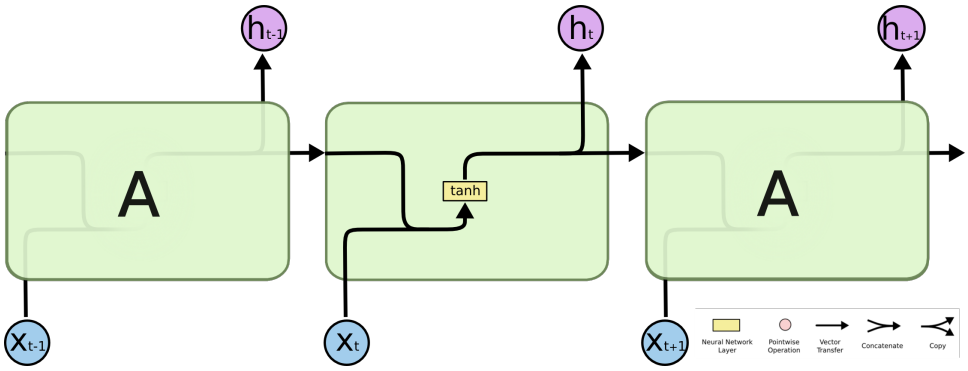


Figure 1.7: standard RNN containing a single hidden layer. Image and examples extracted from [13]

When we are designing a RNN we must take into consideration that there are situations in which we need a lot of information from the context in order to solve a problem. If we return to the problem of word prediction in a text and we have to complete the sentence "*I am going to start swimming in the*", recent information suggests that the next word is probably the name of a place where we can swim but if we know that in the previous sentence we were talking about "*how sunny it is in the beach*" we could think that we will go swimming in the "*sea*". The problem here is that it is not an easy task to highlight relevant information. Then, the explanation of the RNN given until now is absolutely able of handling "long-term dependencies" in theory. However, as in other cases, even if for humans it seems very easy to find a good solution to a problem, if we try to solve this problem with a simple RNN, as it is shown in Figure 1.7, we realise that these good results cannot be obtained, as it is explained by Colah [13], and more deeply by Bengio et al. [5] or by Weinlein [40]. One of the typically problems to overcome is the vanishing gradient problem (explained by Hochreiter et al. [23]

and by Gers et al. [18]). Nevertheless, there are some tricks to solve this issue.

1.2.2 Long Short Term Memory networks

In order to be able to learn long-term dependencies, in 1997, Hochreiter and Schmidhuber [23] introduce a special kind of RNN called Long Short Term Memory networks (LSTM).

Instead of using the simple internal repeating structure of the standard RNNs, as we can see in Figure 1.7, LSTMs have a different way of computing the hidden state. As it is described by Colah [13], rather than having a simple concatenated function in order to obtain the output, the LSTM layer has the ability to remove, add or let information go through the current internal state of the layer using a sort of structure called gates. These gates are composed by a sigmoid neural network layer and a point-wise multiplication operation describing how much of each component should be let go through.

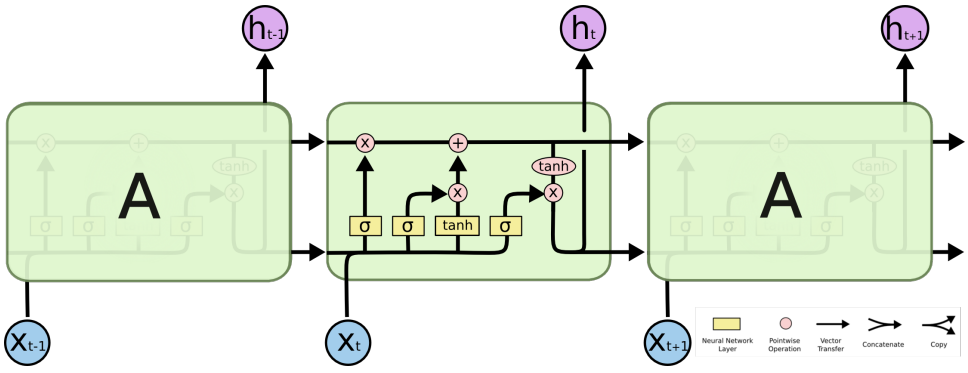


Figure 1.8: LSTM diagram visualising the interconnection between the four inner layers inside a LSTM layer. Image and examples extracted from [13]

If we briefly follow the LSTM diagram in Figure 1.8, we show, at first sight, the sigmoid layer, which is called the *forget gate layer*. Its goal is to decide what information is going to be thrown away from the previous hidden layer. Then, there is a second sigmoid layer known as *input gate layer*, which is in charge of choosing what values will be updated using the output values of a *tanh* layer. This combination will update the current internal state of the layer (the upper

horizontal line) that will be transferred to the next iteration of the recurrent hidden layer. And finally, after applying a *tanh* function to the internal state value (to push the values to be between -1 and 1), there is a sigmoid gate that regulates the output of the current hidden layer h_t.[1]

1.3 Decision Theory for Supervised Learning: The loss function

According to Jeff Miller [32], in supervised learning we got some data $(X, Y) = (x_1, y_1), \ldots, (x_n, y_n)$ where $x_i \in \mathbb{R}^I$ is the information that we have and $y_i \in \mathbb{R}^O$ is the information that we want to deduce. Y could be, for example, the class, in classification problems, or a simple value, in the case of regression problems.

If the prediction of the information we want to deduce is noted as $\hat{y}$, our goal is to have $Y = \hat{y}$. In order to obtain this, we could construct a function $\mathcal{L}(Y, \hat{y})$ that maps Y and $\hat{y}$ into a real number representing some cost that means how different $\hat{y}$ is from Y. This could be seen as an optimisation problem where we try to minimise this loss function. The problem is that we cannot do this directly because we do not always know the true value of y when we are doing the prediction.

1.3.1 Minimising conditional expected loss

The solution is to think the minimisation in a probabilistic framework: (X, Y) are random variables where Y is the true value we are going to predict using X. We suppose that there exist a probability distribution on this random variables $(X, Y) \sim p$. The average in probabilistic language would be the expected value. Therefore, we are going to minimise our expected loss

$$\mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x\right]$$

[1] There is a wide variety of models based on LSTM cited in [13], for those who are interested in reading the LSTM with "peephole connections" [17], the Gated Recurrent Unit (GRU) [10] and the Depth Gated RNNs [43].

i.e. a conditional expectation given that X is x.

If we suppose that (X, Y) are discrete random variables, using the definition, this would be

$$\mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x\right] = \sum_{y \in Y} \mathcal{L}(y, \hat{y}) p(y \mid x)$$

Now we have a well-defined problem. Even though we might not know what $p(y \mid x)$ is, we can estimate it or even we can assume a particular probability distribution. Therefore, this is at least a well-posed problem so we can minimise this expression to find $\hat{y} = \underset{Y}{argmin}\, \mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x\right]$.

For example, if we consider the $0-1$ loss where $\mathcal{L}(y, \hat{y}) = \mathbf{1}_{y \neq \hat{y}} = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{if } y \neq \hat{y} \end{cases}$, the expected loss will be $\mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x\right] = \sum_{y \neq \hat{y}} p(y \mid x) = 1 - p(\hat{y} \mid x)$ and if we want to minimise this expression we will have $\hat{y} = \underset{Y}{argmin}\, \mathbb{E}\left(\mathcal{L}(Y, \hat{y}) \mid X = x\right) = \underset{y}{argmax}\, p(y \mid x)$ and this has a very natural interpretation: we will choose the y that is the most probable y for that x. That is exactly what minimising the $0-1$ loss tells us to do. The effect is like finding the most likely class and it is usually used in classification problems.

1.3.2 Choosing f to minimise the expected loss

The connection between this explanation and the first part explained about Artificial Neural Networks is that we can consider a Neural Network like a function f such as $\hat{y} = f(X)$.

Taking into account this fact, if we expand the expectation this time:

$$\mathbb{E}\left[\mathcal{L}(Y, \hat{y})\right] = \mathbb{E}\left[\mathcal{L}(Y, f(X))\right] = \sum_{X,Y} \mathcal{L}(Y, \hat{y}) p(Y, X)$$

$$= \sum_{X,Y} \mathcal{L}(Y, \hat{y}) p(Y \mid X) p(X) = \sum_{X} \left(\underbrace{\sum_{Y} \mathcal{L}(y, f(X)) p(Y \mid X)}_{\text{We are going to call it } g(X, f(X))} \right) p(X)$$

$$= \mathbb{E}^{X} \left[g(X, f(X)) \right]$$

The marginal distribution $p(X)$ is completely independent from $g(X, f(X))$, so we need to minimise $g(X, f(X))$. Furthermore, $p(Y \mid X)$ remains the key quantity.

In fact, all we already did was to reproduce the Law of Iterated Expectations that states that $\mathbb{E}[Y, \hat{y}] = \mathbb{E}^{X}[\mathbb{E}[Y, \hat{y} \mid X]]$.

1.3.3 Conventional Least Squares loss

As Bishop explains [6], the usual approach to network training involves the minimisation of a sum-of-squares error, defined over a set of training data (X, Y), of the form

$$\mathcal{L}(Y, \hat{Y}) = (Y - \hat{Y})^2$$

As previously explained, we can consider (X, Y) random variables but, since Y is a continuous random variable, we can consider as well that (X, Y) will have a density p, i.e. $(X, Y) \sim p$.

The conditional expectation in that case is

$$\mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x \right] = \int \mathcal{L}(Y, \hat{y}) p(Y \mid x) dY = \int (Y - \hat{y})^2 p(Y \mid x) dY$$

Given a smooth (continue and derivative) $p(Y \mid x)$, if we want to minimise this expression we will obtain

$$0 = \frac{\partial}{\partial \hat{y}} \mathbb{E}\left[\mathcal{L}(Y, \hat{y}) \mid X = x \right] = \int \frac{\partial}{\partial \hat{y}} (\hat{y} - Y)^2 p(Y \mid x) dY$$

$$= \int 2(\hat{y} - Y)p(Y \mid x)dY = 2\hat{y}\underbrace{\int p(Y \mid x)dY}_{1} - 2\underbrace{\int Yp(Y \mid x)dY}_{\mathbb{E}[Y|X=x]}$$

This implies that $\hat{y} = \mathbb{E}[Y \mid X = x]$ is a minimum value since the second derivative with respect to $\hat{y}$ is strictly positive (because it is 2).

If we return to the case of neural networks outputs (where $\hat{y} = f(X)$), this implies we are approximating the conditional average of the target data. As Bishop explains [6], most of conventional applications of neural networks only make use of the prediction for the mean. This solution represents the optimal solution for classification problems. However, for problems involving the prediction of continuous variables, the conditional average represents only a very limited statistic. For many applications, it is considerable beneficial to obtaining a much more complete description of the probability distribution of the target data. This was one of the most important motivations of Bishop [6] to introduce the Mixture Density Network.

1.4 Mixture Density Network

Reminder 1 (Multivariate normal distribution) *Given* $X \sim \mathcal{N}(\mu, \Sigma)$ *where* $\mu = [\mu_1, \ldots, \mu_n]^T$ *is the mean and* Σ *is the covariance matrix (positive def. matrix of dimensions* $k \times k$*), the respective probability density function (PDF) is:*

$$f_{\mathbf{x}}(x_1, \ldots, x_k) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \tag{1.1}$$

The conditional density of the complete target vector is:

$$f(x_1, \ldots, x_k \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=1}^{k} f(x_i \mid \mu, \sigma^2) = \left(\frac{1}{2\pi\sigma^2}\right)^{k/2} \exp\left(-\frac{\sum_{i=1}^{k}(x_i - \mu)^2}{2\sigma^2}\right)$$

If the mean and variance matrix are unknown, the standard approach to find the parameters is the ***maximum likelihood method****, which requires maximisation of the log-likelihood function:*

$$\ln \mathcal{L}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{k} \ln f(x_i \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$$

$$= -\frac{1}{2}\ln(|\boldsymbol{\Sigma}|) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) - \frac{k}{2}\ln(2\pi)$$

where $\boldsymbol{x}$ *is a vector of real numbers.*
Taking derivatives with respect to μ *and* σ^2 *and solving the resulting system of first order conditions yields the maximum likelihood estimates:*

$$\hat{\mu} = \overline{x} \equiv \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2.$$

As Bishop explains [6], if we assume that the conditional distribution of the target data is, in fact, Gaussian, then we can obtain the least-squares formalism using maximum likelihood. This motivates the idea of replacing the Gaussian dis-

tribution in the conditional density of the complete target vector with a mixture model [31], which has the flexibility to completely model a general distribution functions. The probability density of the target data is then represented as a linear combination of kernel functions in the form

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{m} \alpha_i(\boldsymbol{x})\phi_i(\boldsymbol{y}|\boldsymbol{x}) \tag{1.2}$$

where m is the number of components in the mixture and $\alpha_i(\boldsymbol{x})$ is called *mixing coefficients*.

In his paper [6], Bishop selected the kernel functions which are Gaussian of the form:

$$\phi_i(\boldsymbol{y}|\boldsymbol{x}) = \frac{1}{(2\pi)^{c/2}\sigma_i(\boldsymbol{x})^c} \exp\left\{-\frac{\| \boldsymbol{y} - \boldsymbol{\mu}_i(\boldsymbol{x}) \|^2}{2\sigma_i(\boldsymbol{x})^2}\right\} \tag{1.3}$$

where $\boldsymbol{\mu}_i$ represents the centre of the i^{th} kernel. The author assumed that the components of the output vector are statically independent within each component of the distribution, and it can be described by a common variance $\sigma_i(\boldsymbol{x})$. As Bishop [6] explains, to be more formal, the assumption of independence can be relaxed by introducing a full covariance matrices for each Gaussian kernel. However, according to [31] and [6], a Gaussian mixture model with this simplified kernel can approximate any given density function to arbitrary accuracy, provided the mixing coefficients and the Gaussian parameters (means and variances) are correctly chosen. Note that this assumption simplifies the calculation of the inverse of the covariance matrix $\boldsymbol{\Sigma}_i$ since we will have a diagonal matrix with the same variance σ_i across all dimensions

$$\boldsymbol{\Sigma}_i = \begin{bmatrix} \sigma_i & 0 & \cdots & 0 \\ 0 & \sigma_i & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \sigma_i \end{bmatrix}$$

Which simplifies the $|\ \boldsymbol{\Sigma}_i\ |^{-1}$ calculation of the equation 1.1 to the σ_i^{-c} of the equation 1.3.

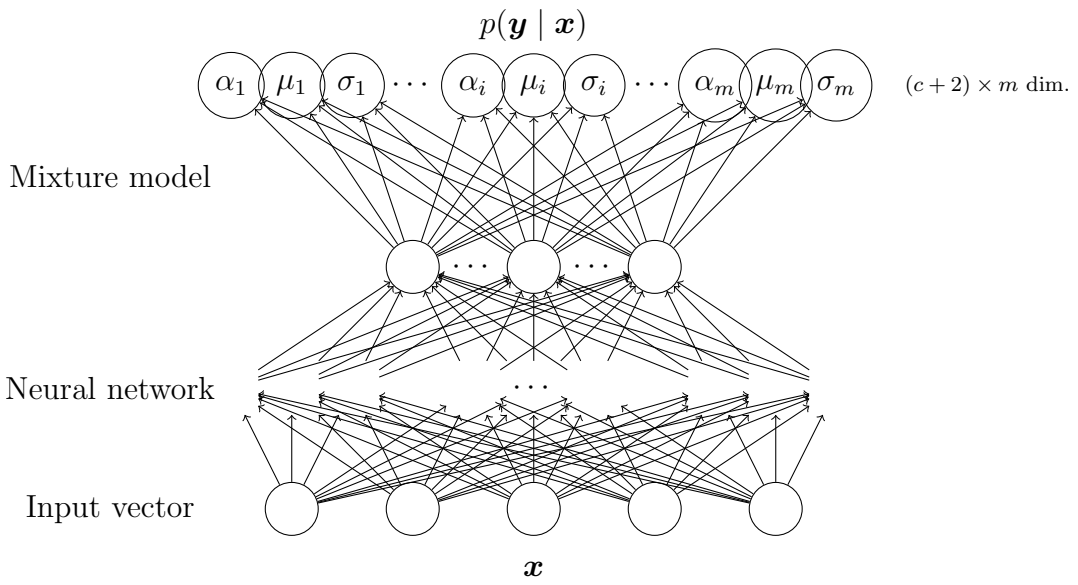


Figure 1.9: Representation of the Mixture Density Network model. The output of the feed-forward neural network determine the parameters in a mixture density model. Therefore, the mixture density model represents the conditional probability density function of the target variables conditioned on the input vector of the neural network.

As we can see in Figure 1.9, given a input vector $\boldsymbol{x}$, the Mixture Density Network model provides a general formalism for modelling an arbitrary conditional density function $p(\boldsymbol{y} \mid \boldsymbol{x})$. This union between the traditional neural network and the mixture model part is achieved by using the log-likelihood of the linear combination of kernel functions as a loss function of the neural network. According to Bishop [6], by choosing a mixture model with a sufficient number of kernel functions, and a neural network with a sufficient number of hidden units, the Mixture Density Network can approximate any conditional density $p(\boldsymbol{y} \mid \boldsymbol{x})$ as closely as desired.

Building this Mixture Density Network increases the number of parameters from c output parameters to $(c+2) \times m$ parameters, where c remains to be the dimension of the output and m is the number of mixtures we are using in the model.

There are some restrictions that Bishop [6] proposes in his article to the

different parameters to satisfy:

1. As required for probabilities, it is important that the mixing coefficients α_i satisfy the constraint $\sum_{i=1}^{m}\alpha_i = 1$. To achieve this restriction, in principle, it is enough to have a *softmax* activation function in the nodes corresponding to α_i.

   As a reminder, the *softmax* function for the outputs corresponding to the α_i parameters, z^{α}, is

   $$\alpha_i = \frac{exp(z_i^{\alpha})}{\sum_{j=1}^{m} exp(z_j^{\alpha})}$$

   This restriction force α_i to lie in the range $(0, 1)$ and sum to unity.

2. Since variance σ_i represents scale parameters, Bishop [6] recommends to represent them in terms of the exponential of the corresponding network output z_i^{σ}

   $$\sigma_i = exp(z_i^{\sigma})$$

   which, in a Bayesian framework, would correspond to the choice of an uninformative Bayesian prior, assuming that the corresponding network outputs z_i^{σ} had uniform probability distribution ([35] [25] and [6]). According to Bishop [6], the use of this representation avoids pathological configurations in which one or more of the variances goes to zero.

3. The centres parameters μ_i represent location parameters. Taking into account the notion of an uninformative prior it suggests that these would be represented directly by the network outputs, i.e.

   $$\mu_{i,k} = z_{i,k}^{\mu}$$

As it is explained in the Multivariate normal distribution Reminder 1, it is possible to construct a likelihood function using the conditional density of the

complete target vector. Then, to define an error function, to use as a loss function, the standard approach is the maximum likelihood method, which requires maximisation of the log-likelihood function or, equivalently, minimisation of the negative logarithm of the likelihood. Therefore, the error function for the Mixture Density Network is:

$$\log \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}) = -\log\left(p(\boldsymbol{y} \mid \boldsymbol{x})\right) = -\log\left(\sum_{i=0}^{m} \alpha_i(\boldsymbol{x})\phi_i(\boldsymbol{y}|\boldsymbol{x})\right) \tag{1.4}$$

Where $\phi_i(\boldsymbol{y}|\boldsymbol{x})$ is the same of the Equation 1.3. As it is explained in [6] and we did in the Subsection 1.3.2, the term $\sum p(\boldsymbol{x})$ has been dropped as it is independent from the parameters of the mixture model, and hence it is independent from the network weights. Thus, the aim of Mixture Density Networks is to model the complete conditional probability density of the output variables. From this density function, any desired statistic involving the output variables can, in principle, be computed.

1.5 How to minimise the error function with respect to the weights in the neural network

Once our neural network architecture is decided, we need a way to minimise the error function to modify the weights in order to obtain an expected result. In order to do this we need to calculate the derivatives of the loss function with respect to the weights in the neural network. According to Bishop [6], one method to solve this problem is by using the standard *back-propagation* procedure, provided we obtain suitable expressions for the derivatives of the error with respect to the activations of the output units of the neural network. Since the loss function is a composition of a sum of terms, one for each pattern, we can consider the derivatives $\delta_k = \frac{\partial \mathcal{L}(\boldsymbol{y}|\boldsymbol{x})}{\partial z_k}$ for a particular pattern and then we can find the derivatives of $\mathcal{L}$ by summing over all patterns. The derivatives δ_k act as *errors* which can be back-propagated through the network to find the derivatives with respect to the network weights. There is a lot of bibliography about this process of optimisation

like [33], [20], [7]. As [6] notes, standard optimisation algorithms, such as conjugate gradients or quasi-Newton methods, can then be used to find a minimum of $\mathcal{L}$. Alternatively, if an optimisation algorithm such as stochastic gradient descent is to be used, the weight updates can be applied after presentation of each pattern separately. In recent years, many new gradient descend optimisation algorithms have been developed, such as Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop or Adam [38].

Nowadays, this differentiation process is implemented in the most Deep Learning relevant libraries in the way it can automatically differentiate native code. As it is used in Autograd Library [16], most part of the libraries use reverse-mode differentiation (also called *reverse accumulation*[1]), which means by using this libraries we can efficiently take gradients of scalar-valued functions with respect to array-valued arguments. Thus, to use this libraries simplifies the gradient-based optimisation problem and this allows us to focus on other problems.

[1]Process well explained on the Automatic_differentiation Wikipedia website or on page 7 of Ilya Sutskever's PhD thesis.

Chapter 2

Motivations And Goals

2.1 The advantages and limitations of the original Mixture Density Network implementation

As we explained in Section 1.3.3 a conventional neural network approximates the conditional average of the target data, conditioned by the input vector. As we introduced, this solution represents the optimal solution for classification problems but not for problems involving the prediction of continuous variables where the conditional average represents a very limited statistic. The effect of approximating the conditional average of the target data, conditioned on the input vector is illustrated in the Figure 2.1, and according to Bishop [6] it has many important implications for practical applications of neural networks.

As an example, we are going to raise a problem that was proposed in the Studio Otoro's blog [36]: Imagine we are trying to do a regression process to approximate the sinusoidal data generated like the following one

$$y = 7\sin(0.75x) + \epsilon$$

Where ϵ is a standard Gaussian random noise.

The shape of the data will look like Figure 2.2a.

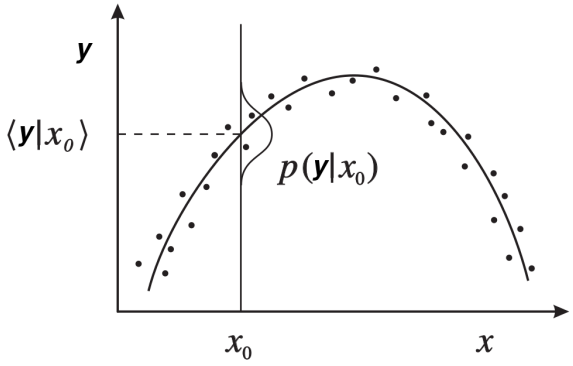


Figure 2.1: A schematic illustration of a set of data points (black dots) consisting of values of the input variable $\boldsymbol{x}$ and the corresponding target variable $\boldsymbol{y}$. Here it is also shown the optimal least-squares function (solid curve), which is given by the conditional average of the target data. Thus, for a given value of $\boldsymbol{x}$, such as the value $\boldsymbol{x}_0$, the least-squares function $\langle \boldsymbol{y} \mid \boldsymbol{x} \rangle$ is given by the average of $\boldsymbol{y}$ with respect to the probability density $p(\boldsymbol{y} \mid \boldsymbol{x})$ at the given value of $\boldsymbol{x}$. Image extracted and adapted from [6].

As we have previously seen in Section 1.1, a neural network of even one hidden layer has universal approximation capabilities. This allows us to obtain a good enough solution to this problem, as we can show in Figure 2.2b, using a simple one-hidden layer of 20 nodes.

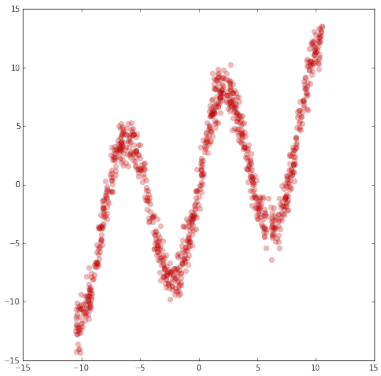
However, imagine we are trying to solve the inverse problem, as we can see in Figure 2.3a.

The generation of this data in that case verifies the following equality:

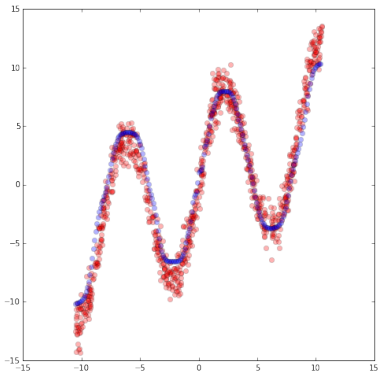
$$x = 7\sin(0.75y) + 0.5y + \epsilon$$

The problem is that, as we can see in Figure 2.3a, there are multiple values of y's that have the same x result. As a consequence, this time we are trying to learn a multi-valued function[1]. However, at the same time, as we previously explained in Section 1.3.3, we are approximating the conditional average of the target data, conditioned on the input vector. In this situation, the average of several correct

[1]The original function's definition in mathematics forces to have no more than one anti-image. However, there is the definition of a multi-valued function that relaxes this constraint.

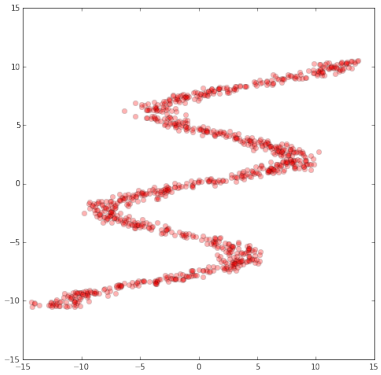


(a) Sinusoidal data plot generated using 1.000 samples from random x points.

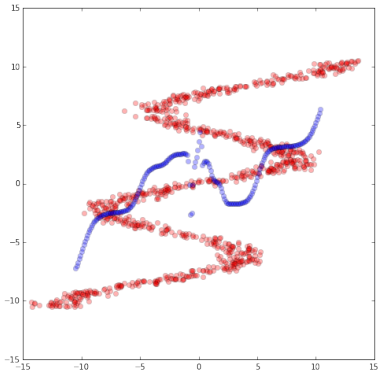


(b) Neural network prediction using a fully connected one-hidden layer of 20 nodes with 1.000 epochs of RMSProp optimisation method and a simple subtraction loss between prediction and real values.

Figure 2.2: Images extracted from Studio Otoro's blog [36]



(a) Sinusoidal data plot generated using 1.000 samples from random y points.



(b) Neural network prediction using a fully connected one-hidden layer of 20 nodes with 1.000 epochs of RMSProp optimisation method and a simple subtraction loss between prediction and real values.

Figure 2.3: Images extracted from Studio Otoro's blog [36]

target values in Figure 2.3a is not a correct prediction value. Consequently, if we try to solve this problem by using a conventional neural network, the result is not the desired one, as we can see in Figure 2.3b.

To solve this problem, we can predict an entire probability distribution for the output using the Mixture Density Network model proposed by Bishop [6], but Bishop in his original implementation made his own program where instability problems were self-contained. However, when we try to solve these problems currently, we usually do it by using software like Autograd and this will force us to take into account the characteristics of such software to solve the problem, as we will explain later. If we use some software like Adagrad and we try to implement the solution in the way Bishop explained in the original article, one can easily observe a very unstable behaviour (rapidly the training loss tends to NaN if we change a little the data set or when we increase a little bit the difficulty of the problem to solve).

Therefore, our first aim will be to collect a set of techniques that will allow us to solve a lot of problems in which we want to predict a probability distribution by using Mixture Density Network avoiding NaN's problem and other derived problems of the model proposed by Bishop [6].

2.2 Is it the variance of the Mixture Density Network a confidence factor?

The second question we asked ourselves is if the variance estimation we can obtain by using the Mixture Density Network could be used for a confidence factor of the prediction. As Lakshminarayanan et al. [29] explains, the quantification of predictive uncertainty in neural networks is a challenging and still unsolved problem.

Consequently, we are going to deepen the concept of uncertainty and we will see to what extent we are able to propose a solution by using neural network models for the different aspects that includes such concept.

2.3 Real application of a Mixture Density Network

Finally, we will try to solve a real application problem where Mixture Density Networks provides new information compared to the standard neural network models. There are many problems for which we may be tempted to use neural networks directly as we know them nowadays to make predictions, but if the decision that has to be taken is of high risk (e.g., control of a nuclear power plant or prediction of the shares evolution in the market), it is important to look for methods that allow us to add more information about the certainty of those predictions.

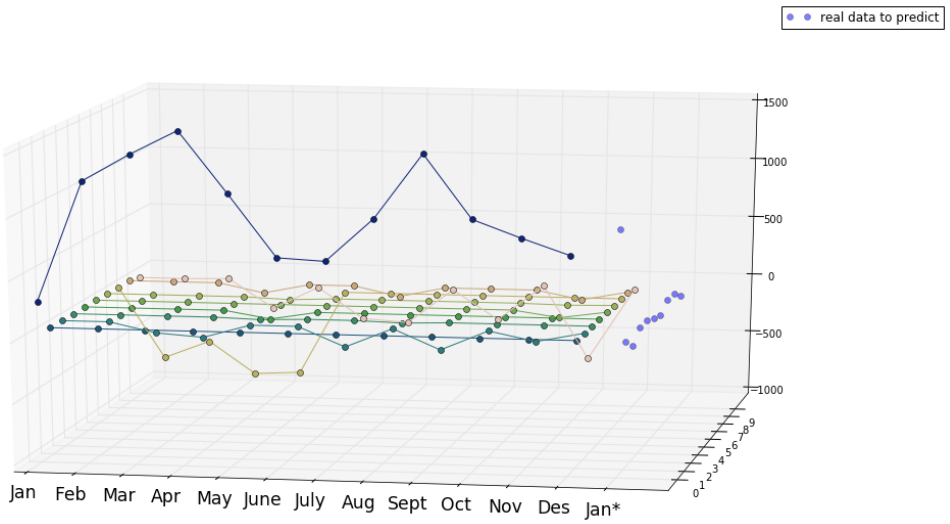


Figure 2.4: Visualisation of the transaction values made during a year every month for 9 accounts and the transaction value for the next January (that is the value we want to predict). If we want to make decisions depending on the predicted value, we are forced to not only know the predicted value but to know how sure we are about that prediction.

The problem we will select will be a financial issue: we only be interested in the prediction of something if we have a minimum certainty about the prediction we made. As we show in Figure 2.4, our dataset will be a collection of data series

of 13 values $(x_1, \ldots, x_{13})$ where each of these x_i values correspond to a financial transaction value for a month of a certain account (where their sign value shows whether the transaction had as ending point that account or it had it outside).

In that case, the assumption is that we consider that the last x_{13} transaction variable depends on the other previous variables $(x_1, \ldots, x_{12})$, i.e. if we know the transaction values of the previous twelve months of a certain account, we can deduct the behaviour of this account the next month. Assuming that this is true, we could build a regression model by using the $(x_1, \ldots, x_{12})$ months information to predict the x_{13} as we can see in the Figure 2.4.

Dealing with this problem gives us the advantage that we have a lot of data. However, the nature of these data involves some challenges that may force us to seek a non-generic solution for many other problems.

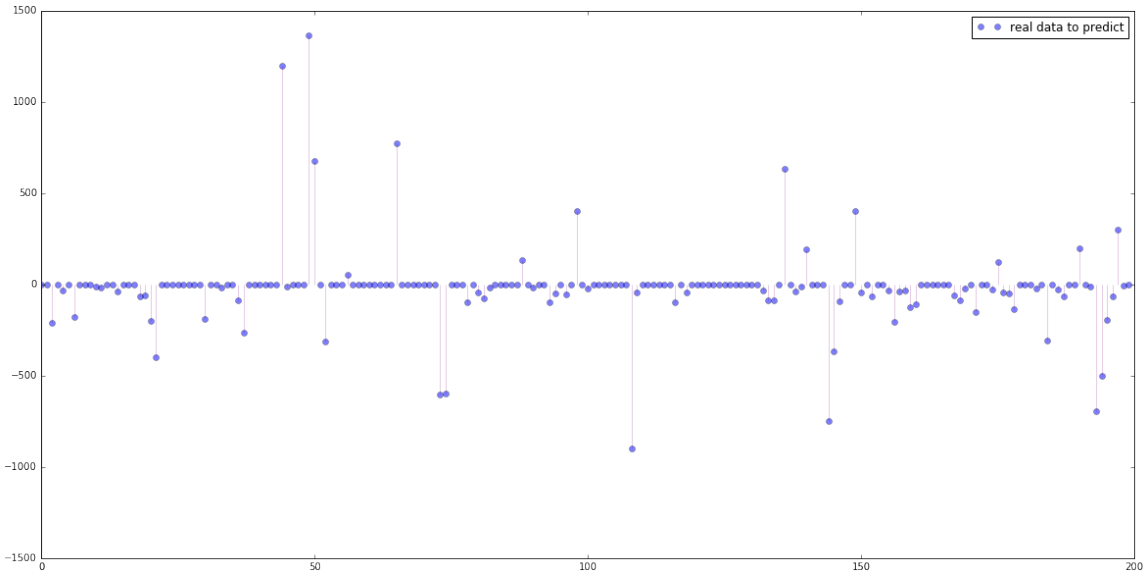


Figure 2.5: Visualisation of 200 transaction values did during a month. This data is what we want to predict, that corresponds to the last month *Jan** of the previous Figure 2.4.

- One of these challenges could be that the amount of external factors that can lead an specific user to realise transactions in a month of a certain value may not be directly correlated with the transaction values of the previous

months. But it is possible that if we collect all the information from many accounts we can find patterns for "non-extreme" cases.

- Another challenge to overcome is that, as we can see in Figure 2.5, the majority of accounts in a given month will not spend anything and therefore, we will have a dataset full of zeros. Thus, it is possible that predicting *always zero* could be a good (and useless) prediction. In that case, we will have to look for models that improve this prediction.

What is certain is that the same transaction values for twelve months $(x_1, \ldots, x_{12})$ do not have to give the same transaction value in the next month x_{13}. Therefore, we are approaching a density function. That is why Mixture Density Networks may represent a good solution to this problem.

Chapter 3

State of the art

3.1 Mixture Density Networks for Galaxy distance determination

A challenging astronomy problem is to determine redshifts[1] of distant galaxies. As we can see in Figure 3.1 and Bonnett [8] explains, the photometric redshift problem happens to be an inverse problem with heteroscedastic noise properties: Heteroscedastic in the sense that the noise properties of the data are not constant. This is an inverse problem due to the fact there is more than one likely solution to the problem.

The problem that Bonnett [8] concerns the estimation of the probability density function of the redshift for the galaxies in order to capture the galaxies that are closer and far away without confusing them. In that case, the use of a Mixture Density Network could be a possible solution for this problem.

As Bonnett explains [8], from a machine learning point of view, this problem could be summarised in this way: we have a large data set with 5 magnitudes (g, r, i, z, Y) and their respective errors. Furthermore, a subset of these galaxies has been observed with a spectrograph and, hence, we know the exact redshift.

[1] Redshift is an astronomical phenomenon related with electromagnetic radiations. According to Bonnett [8] we will refer to the distance of a galaxy as redshift. It is not necessary to understand this concept to follow the explanation but for a deeper understanding ([8],[30]) recommend to read https://en.wikipedia.org/wiki/Redshift

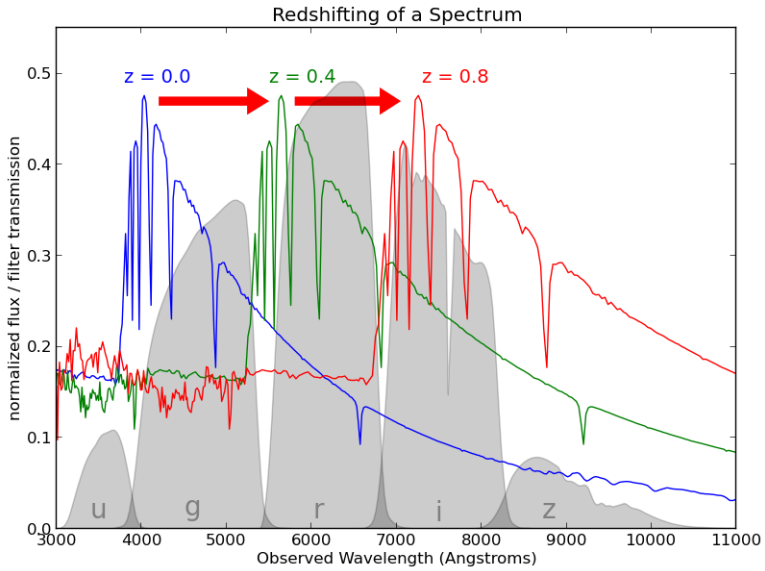


Figure 3.1: The spectrum of the star Vega (α-Lyr) at three different redshifts. The SDSS ugriz filters are shown in gray for reference. Image extracted firstly from [8] and originally was from [30].

Thus, this is a regression problem where the author tried to approximate the full Probability Density Function (PDF) in the following way:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{i=1}^{m} \pi_i(\boldsymbol{x}) Beta(\boldsymbol{y}|\alpha_i(\boldsymbol{x}), \beta_i(\boldsymbol{x}))$$

where $\boldsymbol{y}$ is the redshift, $\boldsymbol{x}$ is the measured features with its corresponding errors, π_i is the mixing coefficient that we explained in 1.2 but, in this case, we changed the notation in order to maintain the notation of the parameters of the *Beta* distribution. According to Bonnett [8], the change in the distribution used has been made because this new one suits the purpose of the problem. However, in order to use the mixture of Beta distributions, Bonnett [8] had to scale the redshift to the $(0, 1)$ range due to the fact that Beta distribution has its support in this interval.

To solve the problem, Bonnett [8] made his implementation in TensorFlow. He decided to use a mixture of 5 Beta distributions to model the PDF. The neural network he used was a 3 consecutive dense layers with $20, 20$ and 10 neurons

in each layer respectively and *tanh* activation function with $L2$ regularisation technique, as he described. The reason why he selected this activation function and not another one was because Bonnett [8] did not find a way to avoid a NaN loss result when he changed the activation functions for $ReLU$.

Finally, in the Figure 3.2, we can see the distributions learnt of an ensemble of galaxies. According to Bonnett [8], this figure shows the estimated distribution in 3 redshift bins by selecting galaxies on the mean of the PDF and, following the explanation Bonnett [8] did, it is a good result.

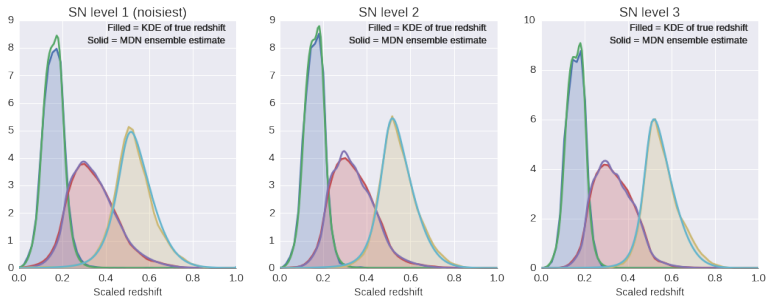


Figure 3.2: Results obtained from estimated distribution in 3 redshift bins by selecting galaxies on the mean of the PDF. The MDN is obtained by stacking the PDF's and the truth is that KDE estimate of the true redshifts. Image extracted from Bonnett [8].

3.2 Mixture Density Networks for handwriting synthesis

One of the most known and recent uses of the Mixture Density Networks can be found in the article of Graves [22] where the author combined a Mixture Density Network with a LSTM neural network to generate complex sequences with long-range structure, simply by predicting one data point at a time. The approach was demonstrated by using discrete and real-valued data. In particular, the aim of that part that used Mixture Density Networks was to extended to handwriting synthesis by allowing the network to condition its predictions on a text sequence generating highly realistic cursive handwriting in a wide variety of styles as we

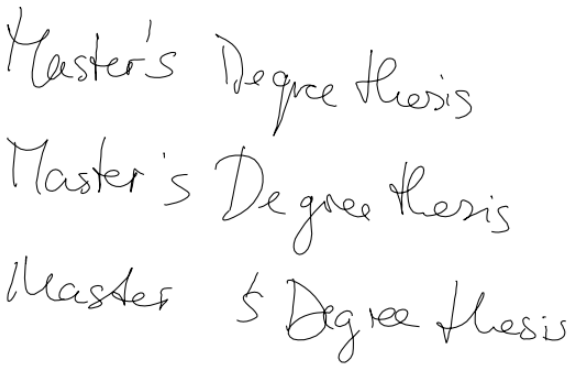


Figure 3.3: Message generated by using the implementation proposed by [22] with different writing styles. Image generated by using [3].

show in Figure 3.3.

In this case, the mixture of distributions he wanted to learn is a bit more complicated than the one used by Bishop [6] and Bonnett [8]:

First of all, a mixture of bivariate Gaussians was used to predict the coordinates of the next point while, secondly, a Bernoulli distribution was used for the prediction of a binary variable that indicates if the vector ends a stroke (that is, according to Graves [22], if the pen was lifted off the board before the next vector was recorded).

As in the original implementation of Bishop [6], Gaussian parameters are obtained by using the same restrictions explained in Section 1.4. The new information is that for the calculation of the Bernoulli distribution parameters from the neural network output, we have to use a *tanh* function to force this output to be within the interval $(-1, 1)$.

One of the results he obtained was the density map we can show in Figure 3.4 where, according to Graves [22], small blobs that spell out the letters are the predictions as the strokes are being written. The three large blobs we can see in the Figure 3.4 corresponds to the predictions at the ending parts of the stroke for the first point in the next stroke. Furthermore, following Graves [22], the bottom heatmap is a visualisation of the mixture components weights during the same sequence.

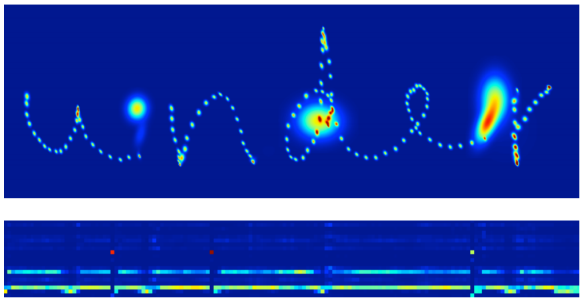


Figure 3.4: According to Graves [22], the top heatmap shows the predictive distributions for the pen locations, the bottom heatmap shows the mixture component weights. The densities for successive predictions are added together, giving high values where the distributions overlap. Image extracted from [22].

The implementation he performed was in TensorFlow and was one of the first implementations in TensorFlow that helped other authors, like Bonnett [8].

3.3 Mixture Density Networks for predictive Uncertainty Estimation

Much more recently, in the Thirtieth Annual Conference on Neural Information Processing Systems (NIPS) that this time took place in Barcelona last December of 2016, 3 authors working for the DeepMind research group of London introduced a new method [29] that allows to estimate predictive uncertainty.

This method could be an important starting point of many new ideas because it neither requires significant modifications to the training procedure nor it is computationally expensive compared to standard neural networks and it can be a simple and readily parallelisable way to estimate predictive uncertainty. In contrast, other Bayesian methods like Bayesian neural networks, which learn a distribution over weights, that are currently the state-of-the-art for estimating predictive uncertainty, have these limitations.

According to Lakshminarayanan et al. [29], they verified that this method is able to express higher degree of uncertainty even for unknown classes.

The summary of the simple method proposed by Lakshminarayanan et al.

[29] is as follows:

- Firstly, they propose to select a correct training criterion by using a proper scoring rule. This statistic concept measures the quality of predictive uncertainty.
- Secondly, they recommend to use *adversarial training* to smooth the predictive distribution.
- Finally, they recommend to train an ensemble of Mixture Density Networks and predict the parameters of the Probability Density Function by combining the parameters of the different Mixture Density Networks.

Their impressive results can be seen in Figure 3.5 where the tails of the function, which they have not been seen for the neural network, the uncertainty grows as it should.

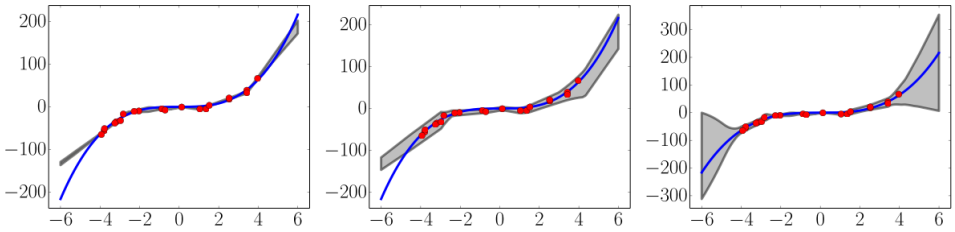


Figure 3.5: Results of a regression problem to points generated by $y = x^3 + \epsilon$ where *epsilon* $\sim \mathcal{N}(0, 3^2)$. The blue line is the *ground truth* curve, the red dots are observed noisy training data points and the gray lines correspond to the predicted mean along with three standard deviations. Left plot corresponds to empirical variance of 5 networks, middle and right plots show the effect of learning variance using a single net and 5 networks respectively. Image extracted from [29].

Or even more impressive, when this process is performed in a classification problem (like MNIST database) and they obtain that new images that the neural network has never seen (from another dataset) give a high levels of uncertainty as we can see in Figure 3.6.

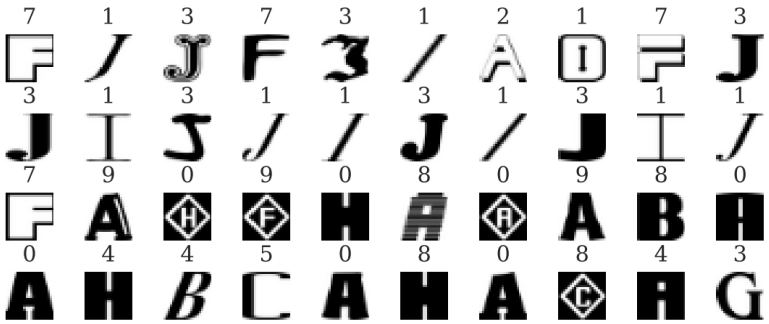


Figure 3.6: Network trained on MNIST and tested on the NotMNIST dataset containing unseen classes: Top two rows denote the test examples with the least disagreement and the bottom two rows denote the test examples with the most disagreement. Image extracted from [29].

Chapter 4

Methodology

4.1 The software library used

One of the things that we can say about the Deep Learning field is that it is in a vertiginous and constant evolution. This can make difficult, or even impossible, to be up to date with the new developments that are taking place in it. As an example, in Figure 4.1 we see four basic software libraries that could currently be the main reference software libraries to develop in the field of Deep Learning but we must keep in mind that these changes go very quickly. Without going more into detail, the library with more stars in GitHub is TensorFlow and its first realise was in November 9th, 2015.

4.1.1 TensorFlow

According to official TensorFlow webpage [2], TensorFlow is an open source software library for numerical computation using data flows graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data array (tensors) communicated between them. As it was explained before, TensorFlow could be used in many other models of Machine Learning outside of Deep Learning techniques but TensorFlow also has a complete part of its library for support these techniques. As we can see in Figure 4.1, one of the more important advantages in front of other software libraries is

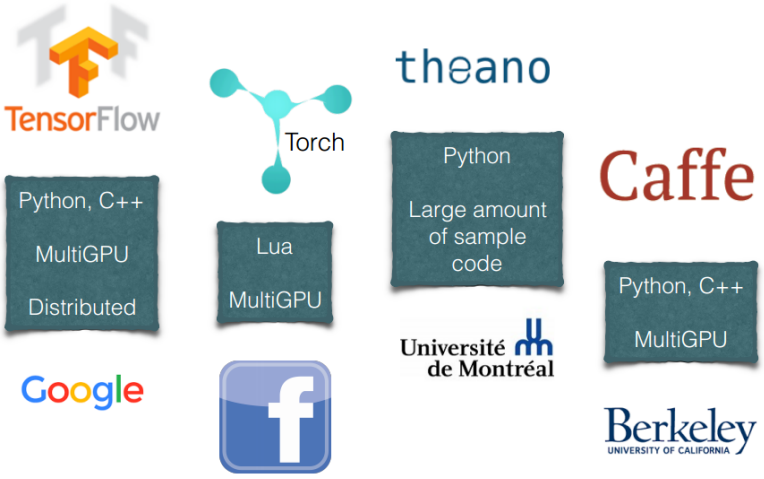


Figure 4.1: Slide extracted from the introduction presentation of the *Deep Learning from Scratch* course [26].

that TensorFlow has a flexible architecture that allows the developer to deploy computation to one or more CPUs or GPUs with a single API.

In our case, we decided to use TensorFlow in order to allow our code to be reused in the future taking into account the huge difference of popularity between other software libraries and TensorFlow.

However, as we can see in Figure 4.2, there are many libraries that runs on top with TensorFlow or Theano. One of the most special (and not coincidentally) libraries that has more stars in GitHub than even Theano is Keras [11].

4.1.2 Keras

According to the main website of Keras project [11], Keras is a high-level neural network library, written in Python and capable of running on top of either TensorFlow or Theano as we can see in Figure 4.2.

Over the past months, in the field of "Deep Learning software libraries", Keras has been widely supported by a large part of the developer community, although it was not strictly tied to any company, organisation o institution (which is true if we do not consider that it was developed as part of the research effort of the

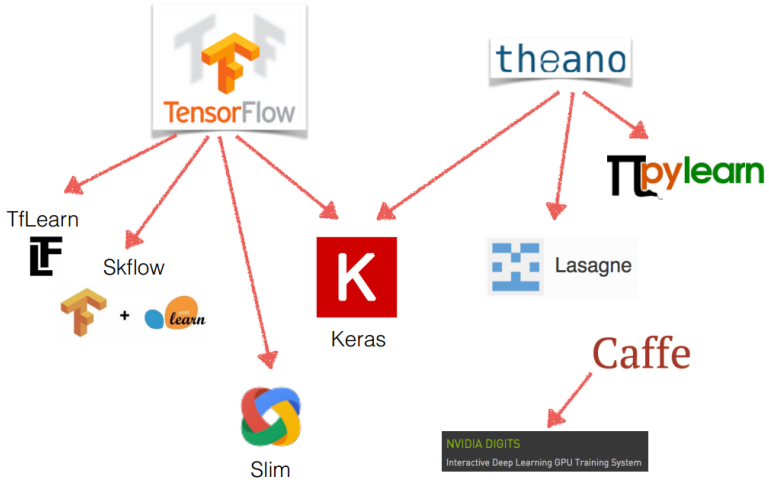


Figure 4.2: Slide extracted from the introduction presentation of the *Deep Learning from Scratch* course [26].

project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), as it is explained in the original webpage of Keras [11]). However, the primary author and maintainer of Keras (François Chollet) (that works as a Deep learning researcher at Google) announced the integration of Keras into TensorFlow on January 15th, 2017. It means that it is very likely that the support in the near future of Keras will be much greater; as we said, this field is constantly changing.

The reason why to select Keras was because we full-filled all the advantages that we obtain by using this wrapper:

- We wanted to do easy and fast prototyping (through total modularity, minimalism and extensibility).
- We wanted to choose between different types of neural networks. We wanted to implement[9] especial neural networks models like Mixture Density Networks and other ones that we will describe in the following chapters.
- Keras [11] supports arbitrary connectivity scheme: An important point for a fast prototyping of our models.

- Like most of the other software libraries, but that it still being a key point, Keras [11] runs seamlessly on CPU than GPU.

4.2 The data set used

Two types of data sets have been used to perform the study carried out in this Master's thesis.

One the one hand, to test the different implementations and tricks of the Mixture Density Networks we have generated data sets according to some equations (and we will usually add a noise to that equation) in the same way that it is shown in the example already explained in Section 2.1.

On the other hand, as we have introduced in Section 2.3, we will try to provide a solution to a real application problem where Mixture Density Networks can be an useful alternative. We will select a private financial data set to tackle the challenge of determining if by using a Mixture Density Network we could obtain a confidence factor that gives us more information that allows us to rely on the prediction of a neural network. Following the notation of Section 2.3, after several attempts, we considered that it would be better to normalise the $(x_1, \ldots, x_{12})$ values indistinctly among the months to improve the training process. Additionally, we have considered as outliers the series that the value to predict was above the $2,500$ and we have directly eliminated it from the data set. This measure has allowed us to focus on the cases that were interesting for this problem and to avoid extreme type of cases that are not interesting for our current problem.

In order to give an idea of the dimensions of the data set to the reader, the data set used contains about three million series $(x_1, \ldots, x_{12}, y)$ and we will use the first one million and a half to train set (and 10% of that to validate during training) and we will use the rest to test. The measure to eliminate the outliers series affected about two hundred thousand series of the initial data set.

4.3 The validation process

Given the complex results of the evaluation process depending on what models we will be exposing during the development part of the work, specific methods will be required depending on what model we want to evaluate. Concerning the models used to only obtain a predictions that try to be similar to the real value, we will use different metrics such as Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE) to evaluate this type of models. On the other hand, concerning models that try to obtain a factor of confidence in their predictions, other techniques closer to visualisation methods will be used to verify if the data is being predicted in a proper way, as we will explain in the following chapters.

Chapter 5

Development of the proposals

5.1 A generic implementation of the Mixture Density Network

As many other authors have found (f.e. Bonnett [8], Mujjingun [19], TensorCruncher [14] and Amjad [4]) performing a Mixture Density Network implementation that is stable to different types of data sets is not a simple task. That is why we decided to make a compilation of the different techniques that we have found that allowed us to use the Mixture Density Networks and we will propose some more to avoid NaNs and other derived problems. As a result, we will end up having a "generic implementation" that will allow us to add or not more tricks depending on the difficulties to solve the problem.

5.1.1 The underflow problem

We refer to underflow situations when we are trying to represent a number in a computer system and that number is considered too small for our computer to be represented. The problem is that there are lots of situations when we obtain a very small number and the computer interprets it as an integer or absolute zero. In the common situation, this error is imperceptible but when there is a logarithm function between operations this is not the case.

When we implement a Mixture Density Network there is a similar problem as we can find in the *forward algorithm* in hidden Markov models: Our batch size in MDNs is a big value and we are summing lot of things together inside a logarithm function following the error function that we explained in Equation 1.4. The problem is that if we extend the whole expression we obtain

$$\log \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}) = -\log\left(p(\boldsymbol{y} \mid \boldsymbol{x})\right) = -\log\left(\sum_{i=0}^{m} \frac{\alpha_i(\boldsymbol{x})}{(2\pi)^{c/2}\sigma_i(\boldsymbol{x})^c} \exp\left\{-\frac{\| \boldsymbol{y} - \boldsymbol{\mu}_i(\boldsymbol{x}) \|^2}{2\sigma_i(\boldsymbol{x})^2}\right\}\right) \tag{5.1}$$

If we look at the last expression, we are doing an exponential (and then a logarithm) of things which may tend to be very small. Then, we could have an underflow problem. A common solution for this type of situations, where we have a logarithm of several summed values, is to use the *log-sum-exp trick* which helps us to eliminate the numerically unstable behaviour of a logarithm of a sum of exponential expressions

$$log \sum_{i=1}^{n} e^{x_i} = \max_i x_i + log \sum_{i=1}^{n} e^{x_i - \max_i x_i}$$

Therefore, if we extend the exponential function within the logarithm of the Equation 5.1 we will obtain

$$\log \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}) = -\log\left(\sum_{i=0}^{m} \exp\left\{\log\left(\alpha_i(\boldsymbol{x})\right) - \frac{c}{2}\log\left(2\pi\sigma_i(\boldsymbol{x})\right) - \frac{\| \boldsymbol{y} - \boldsymbol{\mu}_i(\boldsymbol{x}) \|^2}{2\sigma_i(\boldsymbol{x})^2}\right\}\right) \tag{5.2}$$

We can apply the *log-sum-exp* smooth approximation trick as other authors like Mujjingun [19], TensorCruncher [14] or Amjad [4] did.

5.1.2 The persistent NaN problem

Despite the fact that we had extended the expression as Equation 5.1 and applied the *log-sum-exp trick* to avoid the underflow problem, we saw that the value of

the loss during training was often NaN. Usually a NaN can appear due to three possible scenarios:

1. We have a logarithm of a value which is very close to zero.
2. We have a fraction where the denominator is very close to zero.
3. We have an exponential of a big enough value to be NaN.

If we analyse in depth the Equation 5.1, we can see that the case (1) could exist whenever there is a very small $\alpha(\boldsymbol{x})$ or $\sigma(\boldsymbol{x})$ value. As for what concerns $\alpha(\boldsymbol{x})$, this situation would not be so strange since only one of the m distributions of the mixture has a very small value with respect the others $m-1$ distributions in a particular point this scenario could exist. Therefore, we will seek a solution to this problem. On the other hand, a very small $\sigma(\boldsymbol{x})$ value could be problematic for the cases (1), (2) and even for (3). Finally, an extreme case (but that in our financial data set is very common (because we only eliminated examples above $2,500$); this happens when the difference between the prediction and the real value is very large and the sigma is very small. This extreme case could produce a problem of type (3).

After testing many options, we have considered that there are certain tricks that can help this last problem not to happen so often:

- Applying a **Gradient Clipping** technique when we are training the neural network to try to keep the value small and if there is a very large gradient point, to cut it off.
- Clipping of the $\alpha(\boldsymbol{x})$ and $\sigma(\boldsymbol{x})$ parameters values can be an effective solution but not a successful one (except in the $\alpha(\boldsymbol{x})$ case that we will discuss next).
- Clipping of the real or estimated value could solve the problem of an extreme point but clipping too many times can lead to a less generic solution.
- Applying weight regularisation technique to the respective weights of the $\sigma(\boldsymbol{x})$ part we have verified that tends to give more stable solutions.

- Applying Batch normalisation technique [24] to the output of the layers.

All these variants have been implemented and they will be available in the implementation of the work[9]. However, techniques that gives us a noticeable improvement are described below.

5.1.3 The problem of using too many distributions in a mixture

The situation in which during training the $\alpha(x_i)$ value at some point has a very small value is not so rare if the number of distributions in the mixture m is higher than 1. It is only necessary that any of the distributions is not used at all for that i-th point. In this case, the distribution will have a very small $\alpha(x_i)$ value and when we will apply the Softmax activation function the value will continue to be very small. In that case, we would have a problem and an appropriate solution could be to clip the $\alpha(\boldsymbol{x})$ value defining a minimum value for it.

Making this change we obtained less NaN values in the loss function during training but it was not enough.

5.1.4 Proposal to replace the activation function

As we explained in Section 1.4, in the article that was introduced the Mixture Density Network model [6] it is recommended to represent the variance parameter in terms of the exponential of the corresponding network output. This decision has strengths: for instance, negative network output values tend in positive way toward zero because of the behaviour of the exponential function, but theoretically they will never end to zero. This idea has practical limitations as the exponential function grows very fast and this can lead to a more unstable behaviour. Furthermore, in data sets that are similar to ours there is $\sigma(\boldsymbol{x})$ that must be very big values (since there will be cases where there will be a lot of variability of different real values for the same history of the previous 12 months). This makes us think about another type of representation of the variance by using another function. We wanted a function that had the advantage of the negative

part of an exponential function but that the growth on the positive side was not exponential. Therefore, we decided to define a function to parts that, later, we confirm that there is a very similar definition that, by change, it is proposed as a activation function for neural networks [12]. The original name of the function is *Exponential Linear Unit* (ELU). We can show its shape in Figure 5.1 and it follows the next equation

$$ELU(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

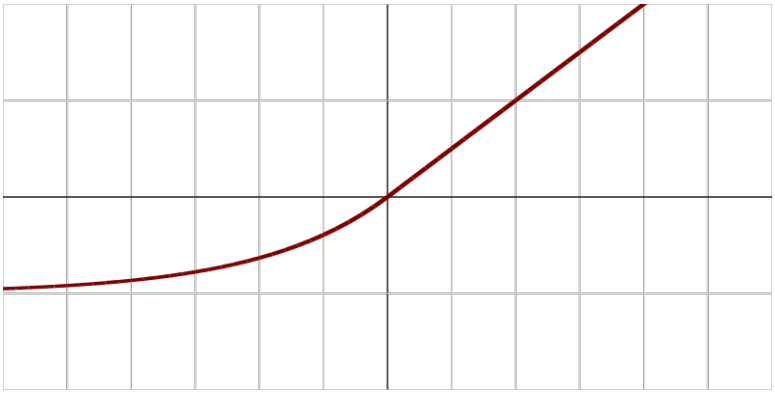


Figure 5.1: Exponential Linear Unit (ELU) function shape. Image extracted from [41].

Therefore, to obtain the desired behaviour $g(x)$, what we had to do was to shift 1 up the value of the ELU's output, i.e. $g(x) = ELU(1, x) + 1$.

5.1.5 Proposal to simplify the distribution function

The changes explained above were not sufficient to avoid in most cases the NaN problem. That is why, from the idea extracted of [8] to use another distribution function, we consider appropriate to review if some other known distributions could be better adapted to our main problem than the Gaussian distribution. Using a Beta distribution function as [8] did, it would not still be a good alternative since the support domain of our data set has higher values than 1.

When we tried to obtain good results from only predicting mean values (i.e. without a Mixture Density Network model) by using different types of neural networks models we could verify that, for our problem, using the Minimum Absolute Error (MAE) as a loss function allowed us to get better results. This gave us the clue to test with the distribution function of Laplace, which has very similar properties with respect to the Gaussian distribution but which also provided us with other good properties.

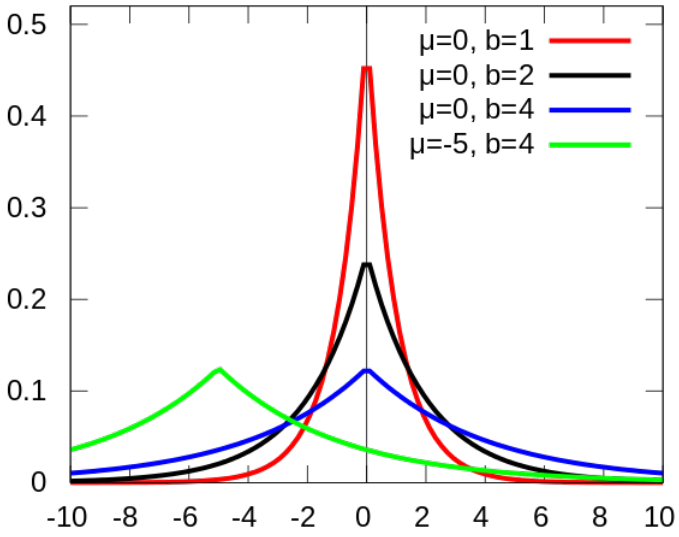


Figure 5.2: Probability density function of the Laplace distribution. Image extracted from [42].

As we can see in Figure 5.2, Laplace distribution has two parameters that perform a similar purpose than the two Gaussian parameters; μ is a location parameter and $b > 0$ is a scale parameter. The probability density function is defined as follows:

$$f(x \mid \mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

As we can see and it is explained in [42], the Laplace distribution is also a reminiscent of the Gaussian distribution; however, whereas the Gaussian distribution is expressed in terms of the squared difference from the mean μ, Laplace

density is expressed in terms of the absolute difference from the mean. Moreover, if we look in the denominator inside the exponential function the scale parameters this time is not square. Therefore, the problems we had with big values of scale parameter can be corrected more easily. Additionally, the way to compute the standard deviation in the Laplace distribution case it only requires to multiply by $\sqrt{2}$ the scale value obtained.

Therefore, we can consider, as we did from Section 1.4 on, the calculation of a mixture of m independent distributions and we can repeat the same algebraic procedure we did in Equation 5.2, i.e.

$$f(x_1, \ldots, x_k \mid \boldsymbol{\mu}, \boldsymbol{b}) = \prod_{i=1}^{m} f(x_i \mid \mu_i, b_i)$$

And, then, we can calculate its corresponding log-likelihood function tu use it as a loss function of our Mixture Density Network

$$\log \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}) = -\log\left(p(\boldsymbol{y} \mid \boldsymbol{x})\right) = -\log\left(\sum_{i=1}^{m} \frac{\alpha_i(\boldsymbol{x})}{(2b_i(\boldsymbol{x}))^c} \exp\left\{-\frac{\mid \boldsymbol{y} - \boldsymbol{\mu}_i(\boldsymbol{x}) \mid}{b_i(\boldsymbol{x})}\right\}\right)$$

$$= -\log\left(\sum_{i=0}^{m} \exp\left\{\log\left(\alpha_i(\boldsymbol{x})\right) - c\log\left(2b_i(\boldsymbol{x})\right) - \frac{\mid \boldsymbol{y} - \boldsymbol{\mu}_i(\boldsymbol{x}) \mid}{b_i(\boldsymbol{x})}\right\}\right)$$

Note: Since the number of parameters of the Gaussian or Laplace distribution are equal and, in addition, they have the same purpose, we will continue with the notation of σ for the scale parameter regardless in the following explanations.

5.1.6 The visualisation problem

To conclude this section, it is important to address the problem of how to visualise the results of a trained Mixture Density Network.

We will start with the simple case where the number of distributions used in the Mixture Density Network is 1, i.e. $m = 1$. In that case, following Figure

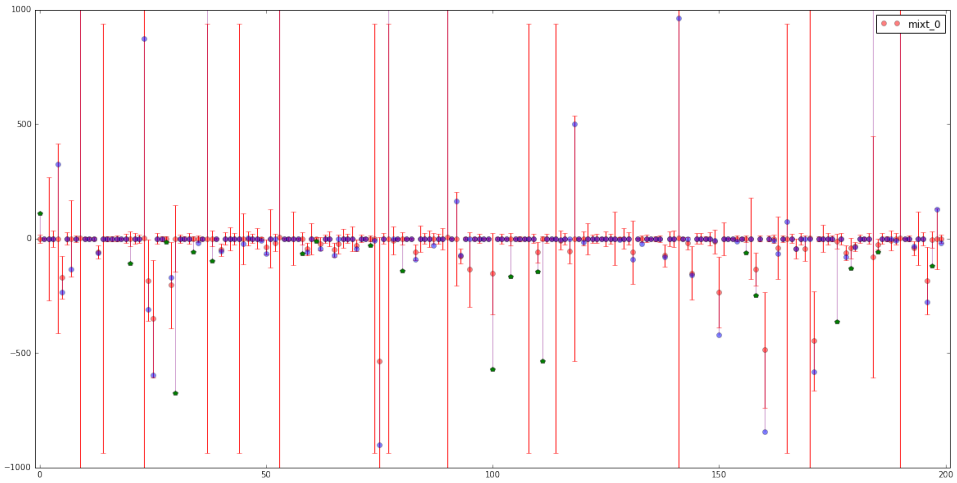


Figure 5.3: Two dimensional visualisation of a Mixture Density Network with $m = 1$.

1.9, we will have three output parameters: α_1, μ_1, and σ_1. In this situation we could easily plot the true value point, the predicted point and even the variability by using the scale parameter to calculate the standard deviation as we can show in Figure 5.3. Additionlly, in this visualisation, that we will use in the future chapters, we will plot the true value point in a different color and shape in case that it is not within the predicted range of variability.

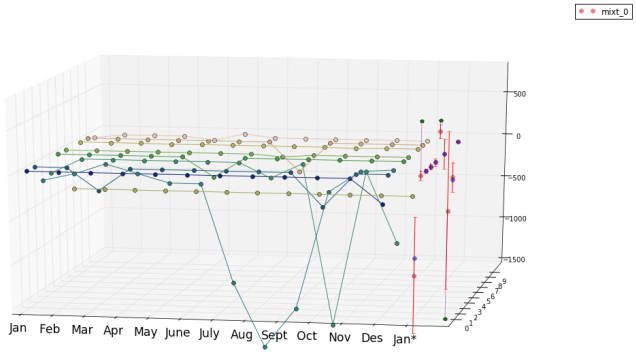


Figure 5.4: Three dimensional visualisation of a Mixture Density Network with $m = 1$.

Moreover, as we have information from the previous 12 months in our financial

data set, we find convenient to perform a visualisation of the points in three dimensions as it can be seen in Figure 5.4.

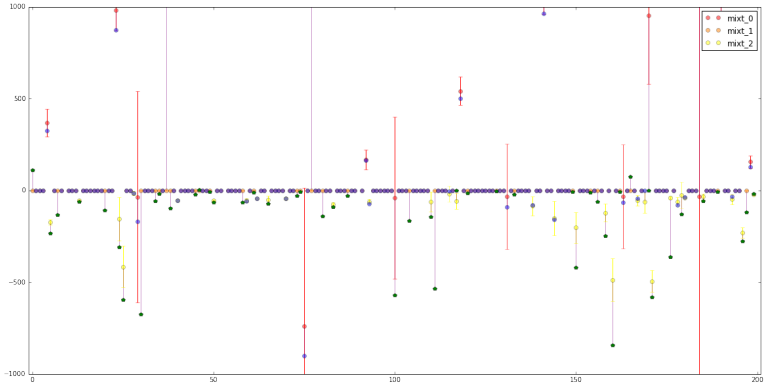


Figure 5.5: Two dimensional visualisation of a Mixture Density Network with $m = 3$.

However, if we perform the visualisation of a Mixture Density Network with several distributions $m > 1$ using the same visualisation as before, as we can see in Figure 5.5, it causes the loss of α (the mixing coefficient) information and, thus, we do not correctly visualise the entire probability density function expressed by the mixture density model.

For these cases, we thought that the best representation would be through a heat graph, as shown in Figure 5.6, where the reddest colours would be considered as a higher probability that the true point is there and the more yellowish colours would be the opposite ones.

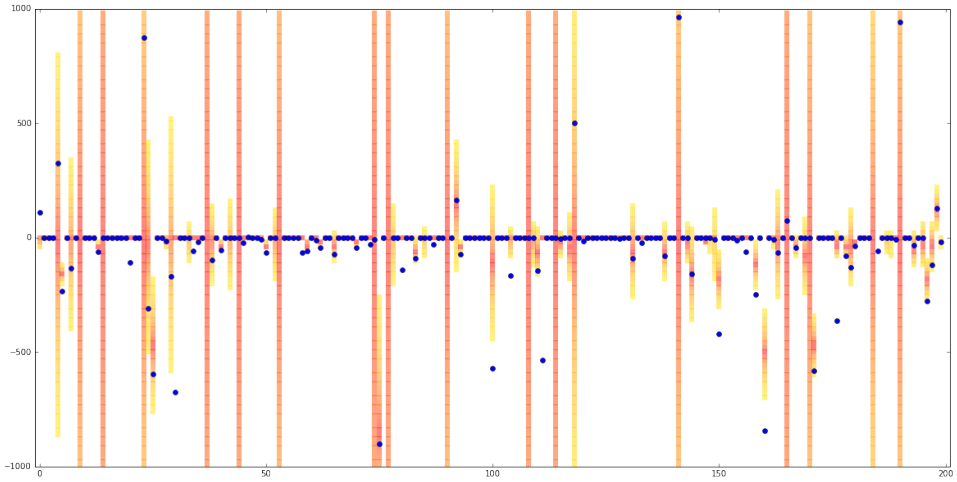


Figure 5.6: Two dimensional heat visualisation of a Mixture Density Network with $m = 3$.

Chapter 6

Evaluation of the proposals

In this chapter we will perform an evaluation of the implemented[9] models and we will analyses the uncertainty related concepts, that will allow us to focus our models to obtain a certain type of confidence factor for the predictions by using Mixture Density Networks.

6.1 MDN 2D

First of all, we will start with a simple example that will allow us to finish tie up loose ends with respect to how the Mixture Density Network works. The Figure 6.1 shows the density function we want to approximate, that is to say, the generated one by using the equation $y = \pm x^2 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, x)$.

To illustrate the effect of using a smaller number of distribution than the optimal one and to see the difference between the use of the Gaussian or Laplacian distribution type, we have considered appropriate to show Figures 6.2 and 6.3 where we can see the evolution of the process of the two types of distributions to try, with a single distribution, to cover the data of $y = \pm x^2 + \epsilon$. The architecture below the Mixture Density part of the neural network we used was a single dense layer of 8 neurons (with a ReLU activation function and a 0.25 dropout).

Comparing Figure 6.2 to Figure 6.3, we can observe that for these simple cases the results of the two cases of distributions are very similar, so we will focus on

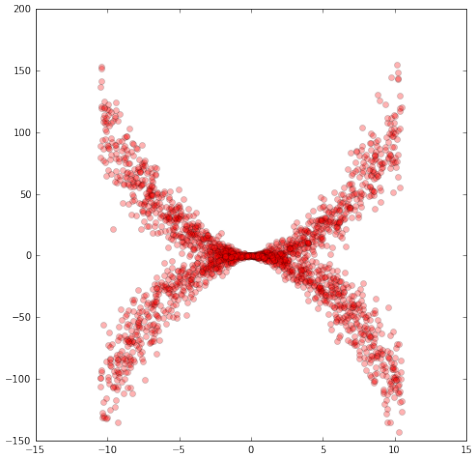
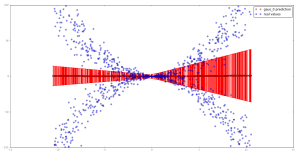
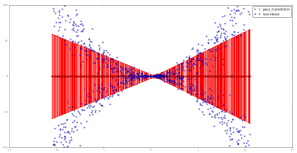


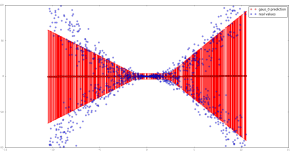
Figure 6.1: $1,000$ points generated by the equation $y = \pm x^2 + \epsilon$



(a) Result obtained after 60 epochs .

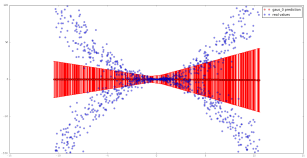


(b) Result obtained after 180 epochs.

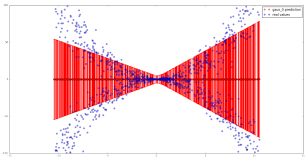


(c) Result obtained after 360 epochs.

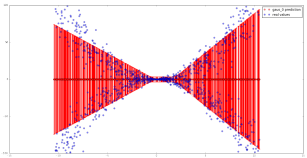
Figure 6.2: Visualisation of the evolution of the Gaussian case for the equation $y = \pm x^2 + \epsilon$ by using 1 million of points of that equation as training set.



(a) Result obtained after 60 epochs .



(b) Result obtained after 180 epochs.



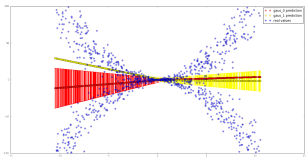
(c) Result obtained after 360 epochs.

Figure 6.3: Visualisation of the evolution of the Gaussian case for the $y = \pm x^2 + \epsilon$ equation by using 1 million of points of that equation as training set.

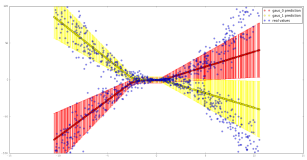
one of the two.

Taking into account the behaviour of the $y = \pm x^2 + \epsilon$ equation, with two distributions it would be enough to be able to approximate it in an appropriate way. In either Laplacian or Gaussian case, if we change the number of distribu-

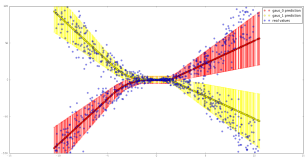
tions parameter in our implementation[9], i.e. $m = 2$, we can show the results in Figure 6.4.



(a) Result obtained after 60 epochs .



(b) Result obtained after 240 epochs.

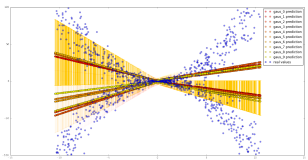


(c) Result obtained after 360 epochs.

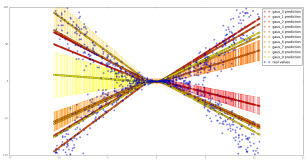
Figure 6.4: Visualisation of the evolution of a mixture of 2 Laplacian distributions for the $y = \pm x^2 + \epsilon$ equation by using 1 million of points of that equation as training set.

It is important to note that all these results were performed using the σ constraint $g(x) = ELU(1, x) + 1$ proposed function. For these problems, that are so simple, the use of an exponential function instead of $g(x)$ would probably give us the advantage of approaching to a suitable solution first. However, we considered that it would be better to visualise the results of the changes proposed in this work using the restriction function $g(x)$ we have proposed.

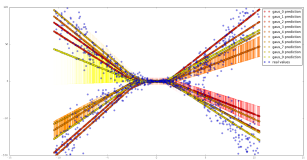
To finish this section, we would like to analyse what would happen if we used a mixture of too many distributions. For example, if we train the same architecture as before but with 10 mixtures, $m = 10$, we can see the result in Figure 6.5.



(a) Result obtained after 60 epochs .



(b) Result obtained after 240 epochs.



(c) Result obtained after 360 epochs.

Figure 6.5: Visualisation of the evolution of a mixture of 2 Laplacian distributions for the $y = \pm x^2 + \epsilon$ equation by using 20 million of points of that equation as training set.

To avoid large errors, in the previous implementation of the visualisation we already weigh the transparency of the variance painted according to the corresponding mixing coefficient α_i of the distribution that we were painting. However,

in the visualisation of the Figure 6.5 it is not enough to be able to verify if the mixture of 20 distributions is approaching well to the equation $y = \pm x^2 + \epsilon$ and, for that reason, we considered appropriately to realise a visualisation with the graph of heat previously proposed in Subsection 5.1.6. The results can be observed in Figure 6.6 where, in fact, the mixture of distributions is approaching the equation in a good way but it could still continue to do so more with more iterations.

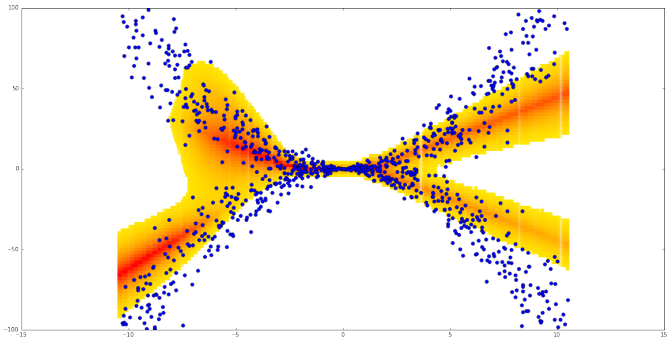


Figure 6.6: Heat visualisation of a mixture of 20 Laplacian distributions for the $y = \pm x^2 + \epsilon$ equation by using 1 million of points of that equation as training set.

6.2 MDN 3D

In this section we will slightly increase the complexity of the issue. Following the problem proposed by TensorCruncher [14], we are going to perform a regression process to a 3-dimensional surface with only the information of one of the coordinates. The code that we developed is ready to have as many inputs and outputs as necessary. Unlike in the previous case, we simply have to modify the output variable c to $c = 2$. A visualisation of the result can be seen in Figure 6.7.

The visualisation that we have chosen more suitable, which we can see in Figure 6.7, is to show the surface $z = x^2 - y^2$ as a wire-frame and visualise the mean of the distribution of the mixture through a sampling process. As we can see, the means overlap the figure, therefore, the result is the expected.

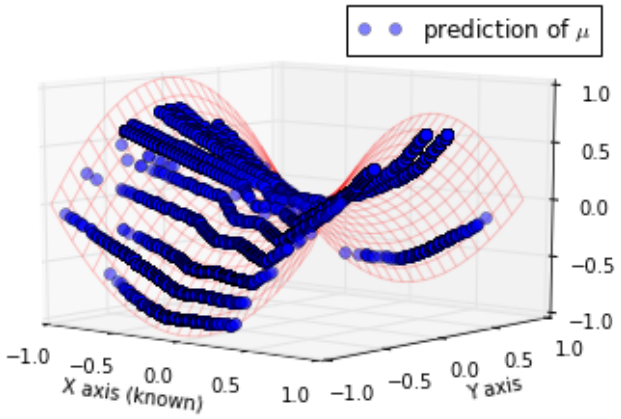


Figure 6.7: Visualisation of the sampling of the averages of 24 distributions of the mixture placed on a wire-frame generated by following $10,000$ points of the equation $z = x^2 - y^2$, where $x, y \in [-1, 1]$

6.3 LSTM time series

As a recurring architecture, the LSTM would be the first choice to solve a more complex problem than problems like the previous ones. In particular, the prediction of the financial transaction value for the next month y given the 12 previous months $(x_1, \ldots, x_{12})$ it is a problem that seems to work much better the recurrent network LSTM model than a dense one. However, after a lot of experimentation trying with different types of layers and methods, we have found a configuration that, although it does not use LSTM, achieves a very small and worse regression error results than LSTM type of neural networks but with the addition that gives us a confidence measure in the prediction it makes. This is why we will start by explaining the LSTM-based architectures tried.

The three LSTM-based architectures we used have:

1. A single input, 2 stacked 128-neuron LSTM layers (with $66,560$ and $131,584$ parameters resp.), a dropout after each of the LSTM layers, a 1 128-dense layer (with $16,512$ parameters) and a dense layer that makes a single output to a value prediction (with 129 parameters).

2. A single input, 3 stacked 128-neuron LSTM layers (with $66,560$, $131,584$

and 131,584 parameters resp.), a dropout after each of the LSTM layers, a 128-dense layer (with 16,512 parameters) and a dense layer that makes a single output to a value prediction (with 129 parameters).

3. A single input, 4 stacked 256-neuron LSTM layers (with 264,192, 525,312, 525,312 and 525,312 parameters resp.), a dropout after each of the LSTM layers, a 256-dense layer (with 65,792 parameters) and a dense layer that makes a single output to a value prediction (with 257 parameters).

The respective results we obtained by using different metrics were:

1. MAE: 110.40, MSE: 9,053,896.50 and RMSE: 3,008.96.
2. MAE: 109.98, MSE: 9,053,640.58 and RMSE: 3,008.92.
3. MAE: 109.93, MSE: 9,053,751.38 and RMSE: 3,008,94.

Where MAE is the Mean Absolute Error, MSE is the Mean Squared Error and RMSE is the Root Mean Squared Error[1].

We must keep in mind that these results are illustrative. To be meticulous it would be necessary to repeat this process several times and to use a procedure like cross-validation. However, due to it is not the main objective of the work to get the best prediction result in this private data set, we consider appropriate to leave this line of development.

As we had introduced in Section 2.3, this data set is so special that even "predicting" always zero gives better results than the average or some similar statistics applied to the previous 12 months (or even using the result of the 12th month). Therefore, we used the results of the different metrics for a prediction of always zero to limit when we were obtaining a good or a bad result.

The results of Always Zero prediction by using different metrics were:

- MAE: 160.14, MSE: 9,104,905.15 and RMSE: 3,017.43

[1] $MAE = \frac{1}{n}\sum_{i=1}^{n} \mid f_i - y_i \mid$, $MSE = \frac{1}{n}\sum_{i=1}^{n}(f_i - y_i)^2$ and $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(f_i - y_i)^2}$ where f_i and y_i are the predicted and real values resp.

Our goal was to get as close as possible to the quality of LSTM prediction and never worsen the always zero prediction.

After verifying that a good neural network architecture (taking into account training time and results obtained) could be to use the (2) LSTM model, we were ableto add the Mixture Density block on top of that recurrent neural network. To start, we trained the a single Gaussian (i.e. $m = 1$) mixture distribution network and these were the results:

- MAE: 174.06, MSE: $9,102,559.05$ and RMSE: $3,017.04$

As we can see, this architecture worsens the result of an always zero prediction. Therefore, we were forced to look for an alternative. That alternative was to use the Laplace distribution function proposed in Subsection 5.1.5 and these were the results:

- MAE: 114.28, MSE: $9,056,219.58$ and RMSE: $3,009.35$

In this case, the results were much more similar to those of just using the architecture of (2).

During that time we performed many tests that led us to the conclusion that, for this problem, the number of distributions in the mixture we found optimal were 1. For example, using the same architecture previously used but with 3 Laplace distribution in the mixture the results were as follows:

- MAE: 134.43, MSE: $9,078,183.87$ and RMSE: $3,013.00$

A visualisation of what we can obtain with the best Mixture Density Network explained before it is shown in Figure 6.8 and in Figure 6.9.

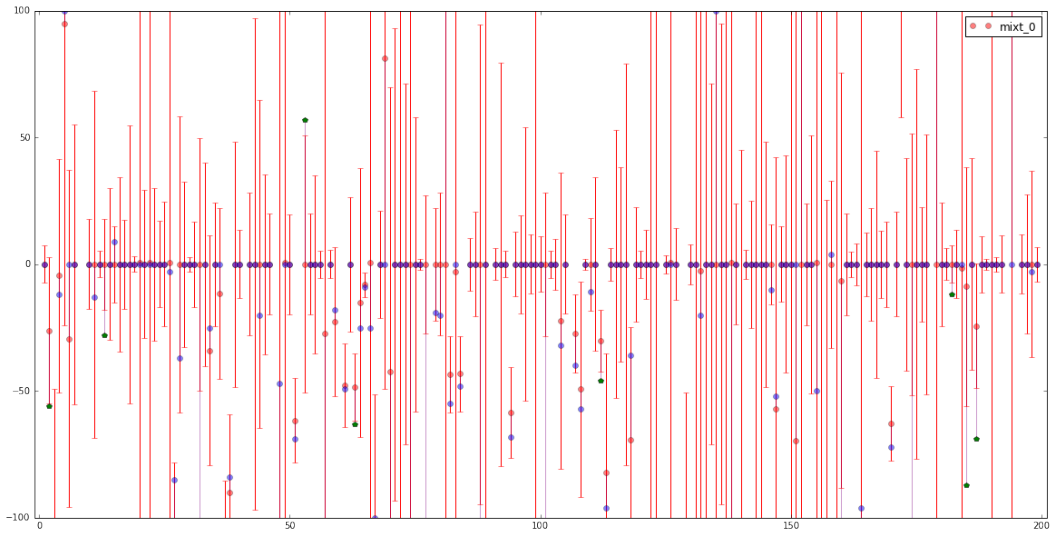


Figure 6.8: Two dimensional visualisation of 200 points of the LSTM-based architecture (2) stacked with a one Laplace distribution using Mixture Density Network model.

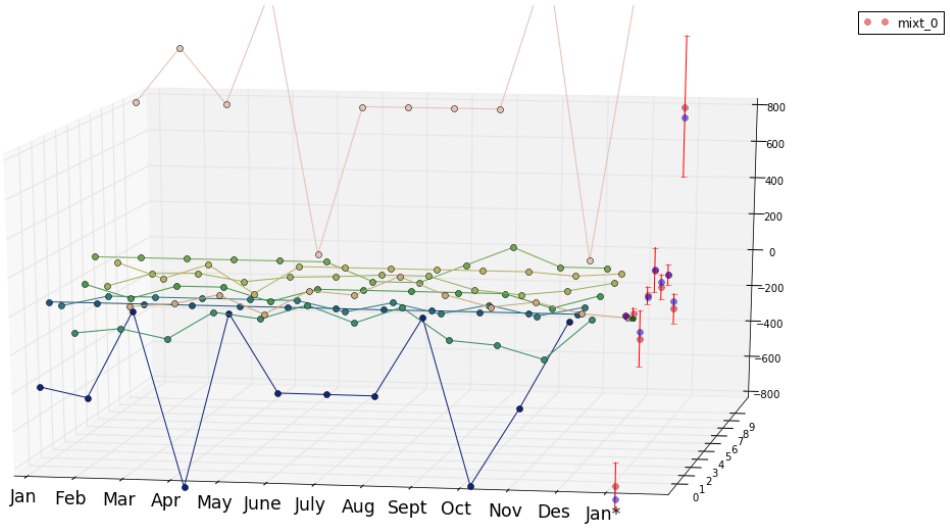


Figure 6.9: Three dimensional visualisation of 200 points of the LSTM-based architecture (2) stacked with a one Laplace distribution using Mixture Density Network model.

6.4 DNN time series

Until the last moment, the complete dense neural network was not the favourite option to be chosen as the best solution to this problem but, as we will explain below, this changed when we made this network deeper and added some more tricks.

In the same way as in the recurrent case, we will show the results that we obtained from different types of complete dense networks that we have proven:

1. 12 inputs, 3 stacked 250-neuron dense layers (with $3,250$, $62,750$ and $62,750$ parameters resp.), a dropout after each of the dense layers and a dense layer that makes a single output to a value prediction (with 251 parameters).

2. 12 inputs, 4 stacked 500-neuron dense layers (with $6,500$, $250,500$, $250,500$ and $250,500$ parameters resp.), a dropout after each of the dense layers and a dense layer that makes a single output to a value prediction (with 501 parameters).

It should be noted that one of the best advantages of training a totally dense network with respect to the LSTM is the low amount of parameters that the dense network has. As well as the training time is reduced in a very noticeable way.

The respective results we obtained by using the same metrics that in LSTM case were:

1. MAE: 113.01, MSE: $8,650,182.88$ and RMSE: $2,941.11$.

2. MAE: 112.69, MSE: $8,650,128.79$ and RMSE: $2,941.11$.

Although the difference between the case (1) to (2) was supposed to triple the training time, the training time of case (2) was totally manageable compared to the LSTMs and, since it was a result more closer to the LSTM, we decided to work with the architecture (2).

6.4.1 The addition of adversarial gradient

According to Nøkland [34], a challenge in machine learning concerns the creation of models that generalise to new data samples not seen in the training data. Recently, some researchers in machine learning as Goodfellow et al. [21], Nøkland [34] or Szegedy et al. [39]) verified that small perturbations added to the input data lead to consistent mis-classification of data samples. This samples that easily misled the model are called *adversarial examples*. One way to generate adversarial examples is to use the *fast gradient sign method* (FGSM) (also called in Lakshminarayanan et al. [29] adversarial training). According to Nøkland [34], in the FGSM a perturbation is added to the original data sample, and this perturbation is proportional to the sign of the gradient back-propagated from the output to the input layer[1]. Therefore, in a simplified notation, our loss function would become:

$$\mathcal{L}_{adv}(\boldsymbol{y} \mid \boldsymbol{x}) = \lambda \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}) + \lambda \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x} + \boldsymbol{\omega})$$

Where $\mathcal{L}_{adv}$ is the new adversarial loss, $\mathcal{L}$ is the original loss function, $\lambda \in [0, 1]$ is a coefficient to give more or less importance to the adversarial part and $\boldsymbol{\omega} = \epsilon \operatorname{sign}(-\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{y} \mid \boldsymbol{x}))$ i.e. is the ϵ-weighted sign of the gradient of the loss function with respect to the input.

As we already introduced in Section 3.3, in the Thirtieth Annual Conference on Neural Information Processing Systems (NIPS) that this time took place in Barcelona last December (2016), 3 authors working for the DeepMind research group of London introduced a new method Lakshminarayanan et al. [29] that they stated that allows to estimate predictive uncertainty. The second step we explained about his method was to use adversarial training to smooth the predictive distribution. This is what we did, we tried to train the completely dense neural networks models but this time using adversarial training and the results were:

2. MAE: 110.62, MSE: $8,648,857.65$ and RMSE: $2,940.89$.

[1] It is important to not to be confused with different the concept of Generative Adversarial Network

(Note: We will fix the enumeration index corresponding to the deepest fully connected neural network type to avoid confusion with the first unused type).

On the other hand, Nøkland [34] proposes a simple variation of the adversarial training performing two times the back-propagation process modifying the weights; the first when we calculate the ω and the second time when the standard back-propagation is applied. We have also implemented [9] this variation of the adversarial training but we have not obtained better results as we can show below:

2. MAE: 112.78, MSE: $8,650,023.04$ and RMSE: $2,941.09$.

In our case we did not prove any improvement so we continued with the adversarial training initially proposed.

Then, we added the single Laplacian Mixture Density model and adversarial training. The results were as follows:

2. MAE: 113.28, MSE: $9,055,498.91$ and RMSE: 3009.23.

At first sight this may seem to be poor results, but if we compared them to those obtained in the case of the LSTM also with the Laplacian Mixture Density Network, this can be seen as an improvement for our results (and above all in training time).

Before continuing, it would be important to go deeper into the idea that [29] propose in their article.

6.4.2 A simple example to verify Deep Ensemble

We will take a step back and we will define a simpler problem. We define 5 Mixture Density Networks of 1 Gaussian distribution with only one 8-neuron fully-connected layer below the Mixture Density block. Subsequently we tried to solve the regression problem shown in Figure 6.10 with each of the 5 Mixture Density Networks but randomly shuffling the data in a different way for each of the networks.

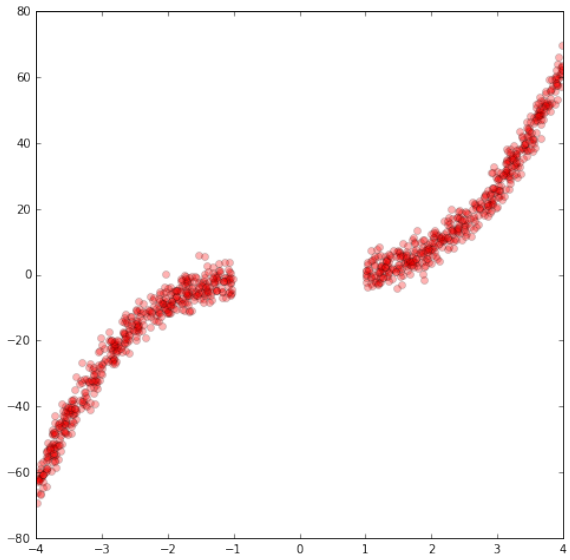


Figure 6.10: Data generated by using the equation $y = x^3 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 3)$ and $x \sim \mathcal{U}((-4, -1) \cup (1, 4))$.

It seems to be a simple problem, but there are two difficulties in the problem represented in Figure 6.10. On the one hand, in the interval $(-1, 1)$ we will not have data and, therefore, our predictions in that interval will be in points of an unknown space. On the other hand, the same problem will happen in the interval $(-\infty, -4) \cup (4, +\infty)$. According to Lakshminarayanan et al. [29], an ensemble of 5 networks of this type would have to be able to give us a behaviour of increase of the variance in the two cases. The result in the first case can be seen in Figure 6.11. It is true that we can observe a growth behaviour but it is not significant. On the other hand, in Figure 6.12 we can observe a behaviour more specific to what we would like to obtain.

Our latest results to arrive at a confidence factor will be based on the results proposed in the paper of Lakshminarayanan et al. [29].

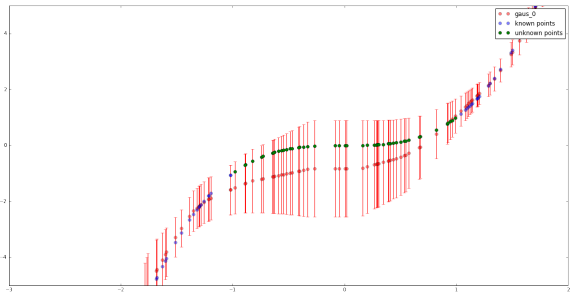


Figure 6.11: (blue) Points obtained following the equation $y = x^3$ where $x \in ((-4, -1) \cup (1, 4))$. (green) Unknown points obtained following the same equation but $x \in (-1, 1)$. (red) Points and variance predicted by the mixture of the 5 Mixture Density Networks.

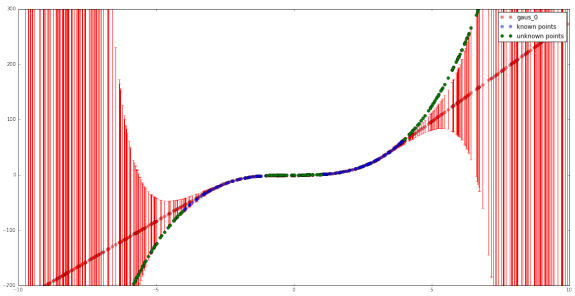


Figure 6.12: (blue) Points obtained following the equation $y = x^3$ where $x \in ((-4, -1) \cup (1, 4))$. (green) Unknown points obtained following the same equation but $x \in (-1, 1)$. (red) Points and variance predicted by the mixture of the 5 Mixture Density Networks.

6.4.3 Application of the results and the adversarial data set definition concept

From the results proposed by Lakshminarayanan et al. [29] and verified in the previous subsection, we repeat the same procedure with our financial data set and 5 always fully-connected Mixture Density Networks with a Laplacian distribution as we finally proposed. If we calculate the same metrics in the same way as in previous cases but, following Lakshminarayanan et al. [29], considering that $\mu_*(\boldsymbol{x}) = \frac{1}{m}\sum_{i=1}^{m}\mu_i(\boldsymbol{x})$ and $\sigma_*^2(\boldsymbol{x}) = \frac{1}{m}\sum_{i=1}^{m}(\sigma_m^2(\boldsymbol{x}) + \mu_i^2(\boldsymbol{x})) - \mu_*^2(\boldsymbol{x})$, the results of the metrics are:

2. MAE: 113.22, MSE: $9,055,350.57$ and RMSE: $3,009.21$.

Now we need a way to verify if our σ_* gives us information about the confidence of the μ. As a conclusion, we thought that a way to assess *if points were more "unknown" then we obtain a greater* σ was to create a data set more and more adversarial in time, what we call an Adversarial data set. This data set history we have been collecting and we have evaluated the σ_i for each of the data set i points. Our goal was that the sum of all σs had an increasing behaviour, and so it was (as we can see in Figure 6.13).

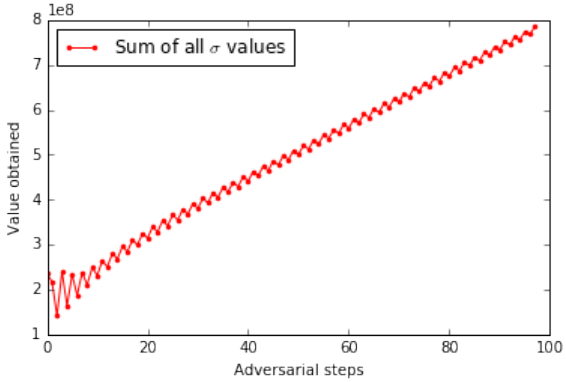


Figure 6.13: Sum of all σs of all the points of the data set during the different 100 adversarial steps.

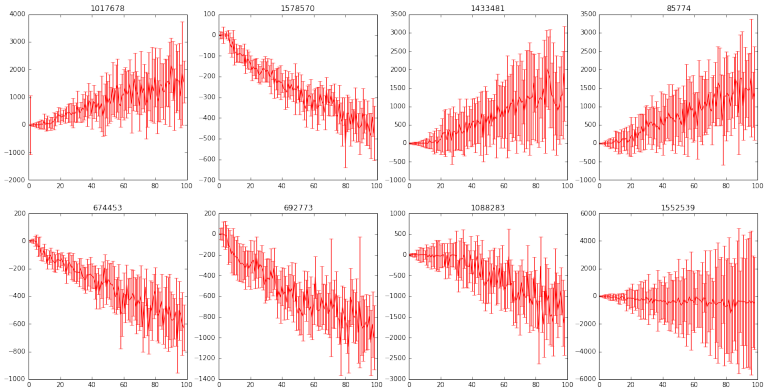


Figure 6.14: Evolution of the σ during the adversarial 100 steps of 8 points selected randomly in the data set.

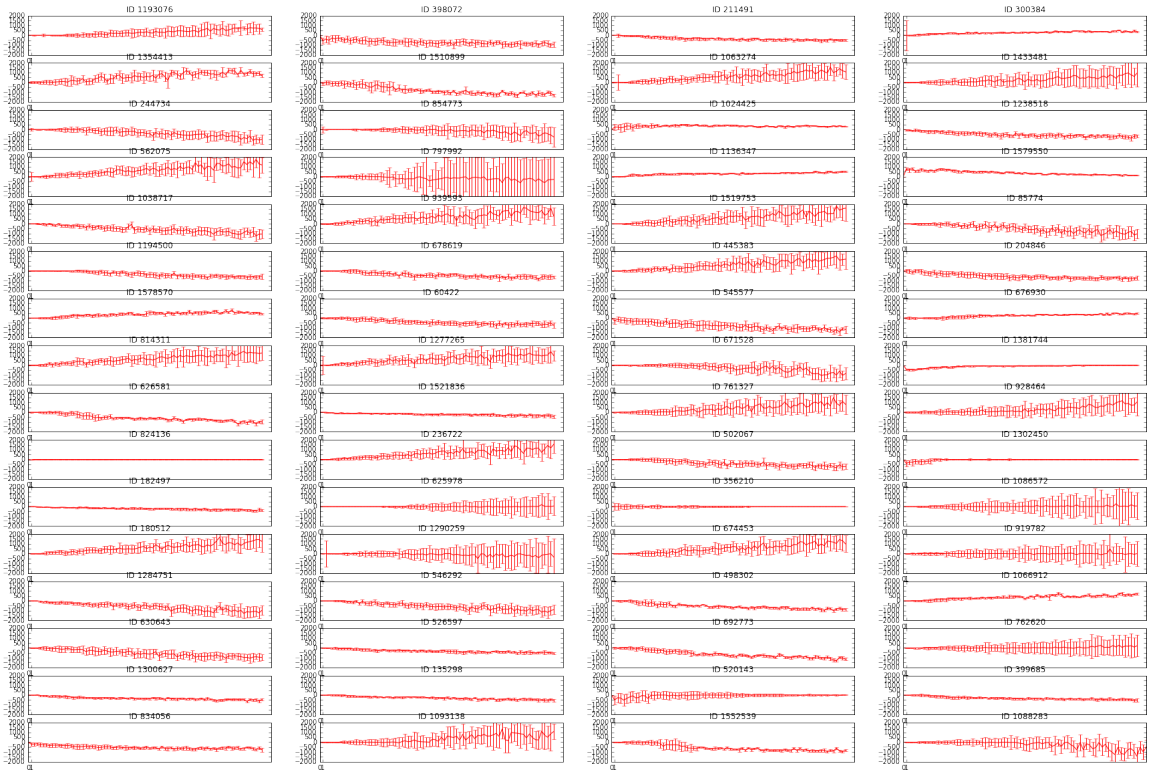


Figure 6.15: Evolution of the σ during the adversarial 100 steps of 64 points selected randomly in the data set.

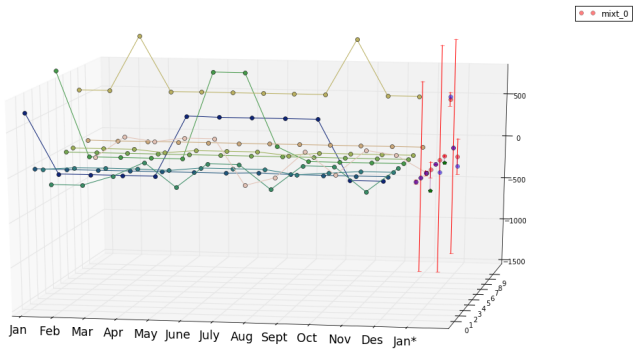


Figure 6.16: Three dimensional visualisation of the predictions of the Ensemble of 5 fully-connected Mixture Density Networks with a Laplacian distribution and trained by using Adversarial training.

Chapter 7

Conclusions

This Master's Thesis has had three important parts. In the first one, we analysed the state-of-the-art regarding the Mixture Density Network models and we developed a reference implementation [9] giving solutions for many of the numerical stability problems that characterise this type of models. A deep analysis of the current situation of Mixture Density Networks has led us to the second part. The second important part of the work was to realise an understanding and proof of current literature methods to measure uncertainty through Mixture Density Networks. And finally, the third part was a proposal of the use of an increasingly adversarial data set to obtain some reliability regarding the predicted uncertainty. This proposal is closely linked to the fact that the hypotheses written by Lakshminarayanan et al. [29] in their article gave good results in our experiments. However, this is not a closed topic since it would be necessary to check more situations to consider that the uncertainty could be well estimated in this way.

We can consider that the stated objectives of the project were satisfied since we achieved a stable implementation of Mixture Density Networks for all the problems that we raised (even for a problem as peculiar as the financial data set we used, in which most of the values were absolute zeros and a good prediction could always be predicted to be zero. Above all, the result to be remarked would be that we obtained a first implementation in a complex case by using the Mixture Density Networks to provide, not only a prediction value, but also an estimate of the confidence in such prediction.

To sum up, we obtained perceptible result. In order to continue with this work, we could consider three levels of possible improvements: Firstly, the continuation of the research regarding a more adequate architecture for the problem we face; secondly,the proposal of new ways of proving whether the σ could be a good confidence estimator; and, finally, we could try to find a way of obtaining other estimators of confidence that came from Mixture Density Networks.

Therefore, the concrete lines to follow could be:

- The application of Adversarial training for recurrent architectures.
- Findings regarding a more suitable number of mixtures to solve our problem.
- The continuation of the idea regarding the creation of another set of data that we can later ensure that it is an "unknown" set for the trained neural network and verify that the σ is high.
- The calculation of the statistics for Ensemble prediction using other relationships between the different mixtures.
- Finding concerning the point of connection between this method and other Bayesian methods that currently give good estimates of uncertainty.

References

[1] *Anyone can learn to code an lstm-rnn in python (part 1: Rnn), iamtrask blog.* http://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/. Accessed: 2016-12-10. 5

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org. 33

[3] U. o. T. Alex Graves, *Recurrent neural network handwriting generation.* http://www.cs.toronto.edu/~graves/handwriting.html. Accessed: 2017-1-5. 29

[4] A. Almahairi, *Mixture density networks.* https://amjadmahayri.wordpress.com/2014/04/30/mixture-density-networks/. Accessed: 2017-1-5. 38, 39

[5] Y. Bengio, P. Simard, and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult*, Neural Networks, IEEE Transactions on, 5 (1994), pp. 157–166. 7

[6] C. M. Bishop, *Mixture density networks*, (1994). 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 29, 41

[7] ——, *Pattern recognition*, Machine Learning, 128 (2006). 18

[8] C. Bonnett, *Mixture density networks for galaxy distance determination in tensorflow*. http://cbonnett.github.io/MDN.html. Accessed: 2017-1-5. 26, 27, 28, 29, 30, 38, 42

[9] A. Brando, *Mixture density networks (mdn) for distribution and uncertainty estimation*, 2017. GitHub repository with a collection of Jupyter notebooks intended to solve a lot of problems related to MDN. ii, 1, 35, 41, 48, 50, 58, 64

[10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, arXiv preprint arXiv:1406.1078, (2014). 9

[11] F. Chollet, *keras*. https://github.com/fchollet/keras, 2015. 34, 35, 36

[12] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, *Fast and accurate deep network learning by exponential linear units (elus)*, arXiv preprint arXiv:1511.07289, (2015). 42

[13] Colah, *Understanding lstm networks, colah's blog*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2016-12-10. 4, 5, 7, 8, 9

[14] T. Cruncher, *Tensor cruncher blog*. https://tensorcruncher.wordpress.com/2016/09/07/mdnmixture-density-network-implementation-in-theano/. Accessed: 2017-1-5. 38, 39, 51

[15] B. C. Csáji, *Approximation with artificial neural networks*, Faculty of Sciences, Etvs Lornd University, Hungary, 24 (2001), p. 48. 3

[16] D. D. Dougal Maclaurin and M. Johnson, *Autograd: Efficiently computes derivatives of numpy code.* https://github.com/HIPS/autograd. Accessed: 2017-1-5. 18

[17] F. Gers, J. Schmidhuber, et al., *Recurrent nets that time and count*, in Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on, vol. 3, IEEE, 2000, pp. 189–194. 9

[18] F. A. Gers, J. Schmidhuber, and F. Cummins, *Learning to forget: Continual prediction with lstm*, Neural computation, 12 (2000), pp. 2451–2471. 8

[19] GitHub, *Mixture density network #1061.* https://gist.github.com/mujjingun/7403363ff0249e5b8bbe9d5490e5da80. Accessed: 2017-1-5. 38, 39

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016. http://www.deeplearningbook.org. 18

[21] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, arXiv preprint arXiv:1412.6572, (2014). 57

[22] A. Graves, *Generating sequences with recurrent neural networks*, arXiv preprint arXiv:1308.0850, (2013). 28, 29, 30

[23] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural computation, 9 (1997), pp. 1735–1780. 7, 8

[24] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv preprint arXiv:1502.03167, (2015). 41

[25] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, *Adaptive mixtures of local experts*, Neural computation, 3 (1991), pp. 79–87. 16

[26] D. Jordi Vitrià, *Deep learning from scratch: A short course.* https://github.com/DataScienceUB/DeepLearningfromScratch/blob/master/slides/DeepLearning%20ShortCourse.pdf, 2016. 34, 35

[27] Karpathy, *The unreasonable effectiveness of recurrent neural networks, andrej karpathy blog.* http://karpathy.github.io/2015/05/21/rnn-effectiveness/. Accessed: 2016-12-10. 6

[28] M. lab from University of Montreal, *Multilayer perceptron.* http://deeplearning.net/tutorial/mlp.html. Accessed: 2017-1-5. 1, 2

[29] B. Lakshminarayanan, A. Pritzel, and C. Blundell, *Simple and scalable predictive uncertainty estimation using deep ensembles*, arXiv preprint arXiv:1612.01474, (2016). 22, 30, 31, 32, 57, 58, 59, 60, 64

[30] A. S. learn, *Regression: Photometric redshifts of galaxies.* http://www.astroml.org/sklearn_tutorial/regression.html. Accessed: 2017-1-5. 26, 27

[31] K. E. B. McLachlan G. J, *Mixture models: Inference and applications to clustering*, (1988). 14

[32] J. Miller, *Machine learning videos of mathematicalmonk's youtube channel.* 9

[33] M. A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015. 1, 18

[34] A. Nøkland, *Improving back-propagation by adding an adversarial gradient*, arXiv preprint arXiv:1510.04189, (2015). 57, 58

[35] S. J. Nowlan and G. E. Hinton, *Simplifying neural networks by soft weight-sharing*, Neural computation, 4 (1992), pp. 473–493. 16

[36] S. Otoro, *Mixture density networks with tensorflow.* http://blog.otoro.net/2015/11/24/mixture-density-networks-with-tensorflow/. Accessed: 2017-1-5. 19, 21

[37] F. Rosenblatt, *The perceptron: a probabilistic model for information storage and organization in the brain.*, Psychological review, 65 (1958), p. 386. 2

[38] S. Ruder, *An overview of gradient descent optimization algorithms.* http://sebastianruder.com/optimizing-gradient-descent/. Accessed: 2017-1-5. 18

[39] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing properties of neural networks*, arXiv preprint arXiv:1312.6199, (2013). 57

[40] T. Weinlein, *Diplomarbeit im fach informatik.* 7

[41] Wikipedia, *Activation function.* https://en.wikipedia.org/wiki/Activation_function. Accessed: 2017-1-5. 42

[42] ——, *Laplace distribution.* https://en.wikipedia.org/wiki/Laplace_distribution. Accessed: 2017-1-5. 43

[43] K. Yao, T. Cohn, K. Vylomova, K. Duh, and C. Dyer, *Depth-gated recurrent neural networks*, arXiv preprint arXiv:1508.03790, (2015). 9