Neural Networks in the Emulation of Probability Distributions for Massive Computational Acceleration with Applications in Mathematical Biology

Alexander John Pinches

October 15, 2020

Abstract

The generation of the probability distributions of stochastic models such as those in Biology are computationally expensive time consuming relying on methods such as the Gillespie Algorithm for stochastically correct trajectories or slightly faster, but approximate, methods such as the Tau-Leaping Algorithm. To explore the large multi-dimensional parameter spaces of stochastic reaction networks present in biology we employ neural networks to emulate the probability distributions of the sampled trajectories much faster than using the Gillespie Algorithm. We explore the application of Long Short Term Memory (LSTM) arrays and another new approach in the form of an approximating Mixture Model whose parameters are generated by a feed forward neural network. Our source code is publicly available at https://github.com/Parply/PDFEmulators.

I Introduction

Biological systems are inherently stochastic in nature from the production of proteins from a regulating gene to the interaction of enzymes, or the growth of a population of cells and their observed heterogeneity. The stochasticity of these models create a distribution of members of each species in a model that is determined by the reaction rates. Distributions of these species at the steady state can vary greatly or remain similar with the change of reaction rates exhibiting multimodal distributions. Many models of interest cannot be solved analytically except for all but the simplest models and they rely on computationally expensive algorithms such as the Gillespie Algorithm *Székely and Burrage* [33]. This motivates the need for a computationally inexpensive method to emulate the solutions to such models to accelerate the exploration of their parameter space.

Simulated data from a mixture of 3 Gaussians are used to explore the predictive capability of the model and choose appropriate hyperparameters. In addition to providing a setting where the model will have to learn to emulate distributions of varying modality between 1 and 3. We demonstrate the model in a real world, simplistic and well explored model of a biological process we using a regulating gene producing mRNA and then a Protein. Such a reaction network exhibts heterogeneity due to the stochasticity of the process shown in *Shahrezaei and Swain* [30]. Protein production forms the basis of many biological systems and is stochastic in nature with the decomposition and production of mRNA and proteins and the switching of the promoter from active to inactive states all being stochastic. Seeking to emulate the distribution of proteins given some reaction rates which is not obtainable analytically we then expand this to the joint distribution of mRNA and Proteins with the Mixture Model to explore its capability in higher dimensions.

In pursuit of a faster method to emulate these systems we seek to apply neural networks which have become a popular method for emulating computationally complex problems *Cranmer et al.* [11]. Firstly in the use of LSTM arrays which have been used effectively in the prediction of time series *Siami-Namini and Namin* [31], text prediction *Guo et al.* [18] and even in complex video games with Google's DeepMind AI *Vinyals et al.* [35]. These problems rely on the problem to represented as a series of related points in time. In our case representing the values of the bins as a series. They were used by *Wang et al.* [36] in the solutions of differential equations motivating our exploration of such neural networks in our problem. Secondly, we also apply a new method in the use of feed forward neural networks in the prediction of the parameters of a Mixture Model. Models of this type have seen applications for some problems but are not similar to the problem here as in *Bishop* [7] in estimating highly non-linear problems. Examples given were the kinematics of a simple 2-link robot arm or in the mapping $x = t + 0.3sin(2\pi t) + \epsilon$ predicting x from a given t with ϵ white noise where a Multivariate Gaussian

Mixture Model is predicted for each x. Methods not utilising neural networks exist in Bayesian methods shown by *Beaumont* [5]. The complexity of such methods can get prohibitively large and rely on the computation of many random variables.

An issue unique to this problem compared to the previously mentioned is common loss functions such as the Mean Square Error $MSE = \frac{1}{T+1}\sum_{i=0}^{T}(y_i - x_i)^2$ have small gradients for $|y-x| \in (0,1)$. Which we would expect when emulating probability distributions by their definition. To counteract this we propose the use of divergence measures as loss functions to prevent vanishing gradients caused by some loss functions.

II Mathematical Methods

Here we explain the mathematical methods employed in our models and in the generation of our data.

II.A Multilayer Perceptron

A multilayer perceptron or feedfoward neural network is built up of layers of neurons *Murtagh* [23], *Trevor Hastie and Friedman* [34]. A neuron has inputs $X \in \mathbb{R}^n$ which can be the data or from another neuron, it has weights $W \in \mathbb{R}^n$ and a bias $\in \mathbb{R}$ It computes the activation as a function to the inputs, weights and bias $W^T X + b$. Then it applies an activation function $\sigma(\cdot)$ to the activation to give the output of the neuron as

$$y = \sigma(W^T X + b)$$

.

We can build multiple layers of multiple neurons that feed the output of the previous layer to the next with differing activations and sizes until we reach the output layer. We can define this neural network with $L+1$ layers $z^{(0)}, \ldots, z^{(L)}$ where $z^{(i)} \in \mathbb{R}^{n_i}$ with n_i the number of neurons in layer i. Input $x = z^{(0)} \in \mathbb{R}^d$ and output $\hat{y} = z^{(L)}$ We can then define for $l = 0, \ldots, L-1$ the forward propagation

$$z^{(l+1)} = \sigma\left(W^{(l)} z^{(l)} + b^l\right)$$

Where $W^{(l)} \in \mathbb{R}^{n_{l+1} x n_l}$, $b^{(l)} \in \mathbb{R}^{n_{l+1}}$ and the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ applied elementwise. We want to find the weights and biases such that we maximise the log-likelihood. To do this we must define a loss function for example we select the mean squared error with p the dimensions of the output.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{p}(y_j^{(i)} - \hat{y}_j^{(i)})^2$$

A feed forward neural network with the Identity Function $f(x) = x$ as activation simplifies to a Linear Model which follows from the matrix form of the layers of the neural network by using other activations however we can capture non-linear effects.

II.B Long Short Term Memory (LSTM)

LSTM cells have the ability to learn patterns in series based on an input and the previous cell state. This has lead to them being used extensively in areas such as text prediction and in modelling time series data where the autocovariance function demonstrates the need for a model that can remember long patterns. The ability to learn patterns has been proposed as a way for predicting functions in *Gers et al.* [15] as we believe that for a function with support $\mathbb{X}$ and points along its support $\{x_1, \ldots, x_n : x_i \in \mathbb{X}\ \forall i = 1, \ldots n\}$ it's value at $f(x_i)$ contains information about its value at $f(x_{i+1})$ and similarly $f(x_i)$ contains information about its value at $f(x_{i-1})$. The difference between points needs to be sufficiently small for this information to exist in our case the bins must be sufficiently small to capture the information. Using LSTM's we can stack arrays on top of one another and/or introduce Bidirectionality *Schuster and Paliwal* [29] to improve the models predictive ability. We achieve this by having one set of LSTM arrays going in one direction prediction forwards in time $1 \to T$ and another going in the opposite direction $T \to 1$. Then concatenating or summing their outputs. This has been proposed as a way for the network to learn more context about each point as it has predictions from both directions in text prediction and in our case, we expect it to improve the model's ability to predict in cases with long tails.

The update of the weights in the cells follow from the case of the feed forward neural network as the gates can be viewed as layers in feed forward neural network when we unfold and back propagate through time *Gers et al.* [15].

II.C Standard LSTM

A standard LSTM cell with forget gates as proposed in *Gers et al.* [15] of a given encoder size s and input dimension d consists of a the Cell Memory C_{t-1} this is the long term memory of the cell and the short term memory H_{t-1} which is the output of the cell. These are then fed through the cell's gates. The Forget Gate f_t that controls the forgetting of information stored in the cell memory C_{t-1}, the cell gate g_t which proposes new candidates to be added to the cell memory and the input gate i_t which controls the weighting of these inputs and finally the output gate o_t which controls the output of the cell. We also define $H_0 = 0$ and $C_0 = 0$. A graphical representation of this cell is shown in figure 1.

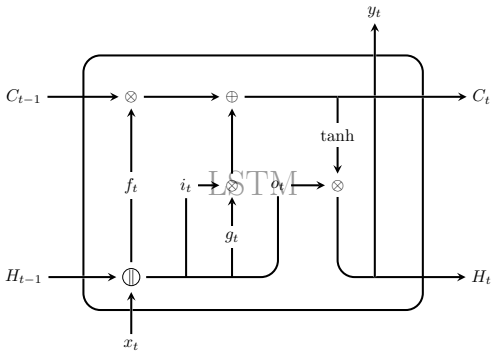

Figure 1: Standard LSTM Cell

Variables:

$$
\begin{aligned}
x_t &\in \mathbb{R}^d : \text{Input Vector} \\
C_t &\in \mathbb{R}^s : \text{Cell Memory} \\
H_t &\in \mathbb{R}^s : \text{Cell Output} \\
f_t &\in \mathbb{R}^s : \text{Forget Gate} \\
i_t &\in \mathbb{R}^s : \text{Input Gate} \\
g_t &\in \mathbb{R}^s : \text{Cell Gate} \\
o_t &\in \mathbb{R}^s : \text{Output Gate} \\
W_i &\in \mathbb{R}^{s \times d} : \text{Weight Matrices} \\
W_h &\in \mathbb{R}^{s \times s} : \text{Weight Matrices} \\
b_i &\in \mathbb{R}^s : \text{Bias Vectors}
\end{aligned}
$$

Operators and functions:

$$
\begin{aligned}
\otimes &= \text{Hadamard Product} \\
\oplus &= \text{Element-wise Addition} \\
\Vert &= \text{Concatenate} \\
\sigma(x) &= \frac{1}{1+\exp(-x)} : \text{Sigmoid Function} \\
\tanh(x) &= \frac{\exp(2x)-1}{\exp(2x)+1} : \text{Hyperbolic Tangent Function}
\end{aligned}
$$

Gate functions:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}H_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}H_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}H_{t-1} + b_{hg}) \\
i_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}H_{t-1} + b_{ho}) \\
C_t &= f_t \otimes C_{t-1} + i_t \otimes g_t \\
H_t &= o_t \otimes \tanh(C_t)
\end{aligned}
$$

II.D Peephole LSTM

The Peephole LSTM adds extra Peephole connections to feed the cell memory into the gates as proposed in *Gers and Schmidhuber* [14] and the gate functions are modified to use the C_i instead of H_i. We represent the additional connections by the blue dotted line in the graphical representation shown in figure 2. It's use in learning relationships over longer periods was shown in *Gers and Schmidhuber* [14] which could be useful in our model especially for a large number of bins.

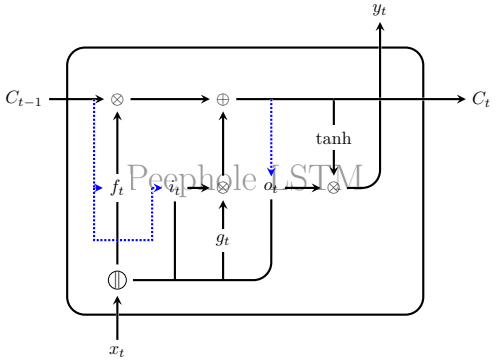

Figure 2: Peephole LSTM Cell

Gate functions:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}C_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}C_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}C_{t-1} + b_{ho}) \\
C_t &= f_t \otimes C_{t-1} + i_t \otimes g_t \\
H_t &= o_t \otimes \tanh(C_t)
\end{aligned}
$$

II.E Activation functions

Example activation functions we could use include the identity function as previously mentioned. We ideally want easily differentiable activation functions and for activations on the final output layers to map to the space of what we are predicting.

$$identity(x) = x$$

A commonly used activation function that was used in *Wang et al.* [36] with their LSTM model was the rectified linear unit (ReLU) function. Its primary weakness being is as it is zero and has zero gradient for negative values so it cannot backpropagate on these values.

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

This leads to variations of ReLU that overcome this problem. One such alternative is exponential linear units (ELU) which also has an extra parameter to optimise from *Clevert et al.* [10]. Where it was shown to obtain higher classification accuracy than ReLU.

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

In our application a PMF must have non negative values less than or equal to one and sum to one. An activation function that is commonly used for this is the standard unit Softmax function.

$$\sigma(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)} \text{ for } i = 1, \ldots, K$$

The non-negativity comes from the exponential function and the summing to one is obviously true.

$$\sum_{i=1}^{K} \sigma(x)_i = \frac{\sum_{i=1}^{K} \exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)} = 1$$

As it is non-negative and sums to unity then all $\sigma(x)_i \in [0, 1]$. We can obviously select from a large variety of activation functions both those available to us already in packages and we can create new activation functions. We here however limit ourselves to the activation functions available in Pytorch as of 1.5.0 in the exploration of improvements they can offer.

II.F Optimisers

In any neural network given a loss we need an algorithm to determine how we change the weights of the layers in a manner that will lead to convergence to a solution. Our choice of algorithm and its hyperparameters can determine if and how fast it converges to an optimal solution. Knowing which algorithm and the hyperparameters to use rely on testing and using performance on similar models as an indicator to determine which to use. The size of each update, the learning rate we denote by η.

II.F.1 Batching

Shown universally across different models is the increase in performance resulting from updating across a batch of data rather than individually for each sample or across the whole training dataset *LeCun et al.* [21]. The intuition behind this is that by using a batch we can help the model learn general features rather than fitting noise in the data and also train a model faster. Too large a batch however can lead to over generalisation a problem which is shown later in the LSTM model.

II.F.2 Gradient Descent

A commonly applied optimiser is that of Gradient Descent *LeCun et al.* [21]. Intuitively we would like to adjust the weights by moving in the direction that reduces the loss function. As we assume the samples from the data

are independent and identically distributed the loss is a summed across each sample for the model parameters θ for N samples and loss function L.

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i, \theta)$$
$$\implies \nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta L(x_i, y_i, \theta)$$

This can be expensive for large datasets and lead to over fitting so instead we use Stochastic Gradient Descent *LeCun et al.* [21]. We approximate the expected gradient by using a random minibatch of the data $(x^{n_i}, y^{n_i})_{i=1}^{N'}, N' \ll N$ This means the estimated gradient is

$$g = \frac{1}{N'}\sum_{i=1}^{N'} \nabla_\theta L(x^i, y_i, \theta)$$

We the adjust the parameter with learning rate $0 < \eta \leq 1$

$$\theta \leftarrow \theta - \eta g$$

This does introduce two training parameters the size of the minibatch and the learning rate. A general rule is to lower the learning rate as you increase the size of the batch. For example with the loss function as the MSE and $\sigma_{(l)}$ representing the activation function of layer l.

We can define the loss

$$L = \frac{1}{N'}\sum_{i=1}^{N'} (z_i^{(L)} - y_i)^2$$

We then take the derivative with the respect to the output of layer L $z^{(L)}$

$$\frac{\partial L}{\partial z^{(L)}} = \frac{2}{N'}\sum_{i=1}^{N'} (y_i - z_i^{(L-1)})\sigma_L'(z_i^{(L)})$$

We can then apply the chain rule to get

$$\frac{\partial L}{\partial z^{(l)}} = \prod_{i=l}^{L-1} \frac{\partial L}{\partial z^{(i+1)}} \frac{\partial z^{i+1}}{\partial z^{(i)}}^T$$

Then we can calculate the gradients

$$\begin{aligned} g(w^{(l)}) &= \frac{\partial L}{\partial z^{(l)}} \frac{\partial z_{(l)}}{\partial w^{(l)}}^T \\ &= \frac{\partial L}{\partial z^{(l)}} (z^{(l-1)})^T \\ g(b^{(l)}) &= \frac{\partial L}{\partial z^{(l)}} \frac{\partial z_{(l)}}{\partial b^{(l)}}^T \\ &= \frac{\partial L}{\partial z^{(l)}} \end{aligned}$$

Using that $\frac{\partial z_{(l)}}{\partial b^{(l)}}$ is the identity and $\frac{\partial z_{(l)}}{\partial w^{(l)}} = z_{(l-1)}$. We then use the gradients to update each of the parameters as explained above. Repeating this for each batch. We then loop over new random samples of batches of the data for each training epoch updating the weights.

SGD guarantees convergence to a minimum but this can be a local minimum and the speed at which it converges can be long *LeCun et al.* [21]. Extensions and variations of the SGD exist in the form of adding momentum *LeCun et al.* [21] and in adaptive learning rate gradient decent (Adagrad) *Duchi et al.* [12] but they were not applied to our models.

II.F.3 Adam

A now very popular optimisation algorithm Adam has seen wide spread use originally proposed in *Kingma and Ba* [19]. Using the first and second moments to control the learning rate of each parameter adaptively. Controlled by 3 parameters α the stepsize, β_1 controls the decay of the moving average of the first moment and β_2 the second moment.Some variations upon this algorithm can be seen in *Reddi et al.* [27] but we have not applied them to our models. We use as parameters the Pytorch defaults $\alpha = \eta, \beta_1 = 0.9, \beta_2 = 0.999$. In the paper α is the learning rate η, $0 < \epsilon$ is a small term to prevent division by zero usually 10^{-8} and g is as defined before for Gradient Descent $g = \nabla_\theta J(\theta)$. This gives us the simple update rule

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where we have

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$m_t = (1 - \beta_1) g_t + \beta_1 m_{t-1}$$
$$m_0 = 0$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$v_t = (1 - \beta_2) g_t^2 + \beta_2 v_{t-1}$$
$$v_0 = 0$$

m_t and v_t are exponential moving averages that are biased estimates for the first and second moments and $\hat{m}_t$,$\hat{v}_t$ are the bias corrected estimates. We can see this by rewriting the formula of each and taking expectations.

$$\begin{aligned} m_t &= (1 - \beta_1) g_t + \beta_1 m_{t-1} \\ &= (1 - \beta_1) \sum_{i=0}^{t} \beta_1 g_i \\ E[m_t] &= (1 - \beta_1) \sum_{i=0}^{t} \beta_1 E[g_i] \\ &= E[g_t](1 - \beta_1^t) \\ \implies E[\hat{m}_t] &= E[g_t] \end{aligned}$$

$$\begin{aligned} v_t &= (1 - \beta_2) g_t^2 + \beta_2 m_{t-1} \\ &= (1 - \beta_2) \sum_{i=0}^{t} \beta_2 g_i^2 \\ E[v_t] &= (1 - \beta_2) \sum_{i=0}^{t} \beta_2 E[g_i^2] \\ &= E[g_t^2](1 - \beta_2^t) \\ \implies E[\hat{v}_t] &= E[g_t^2] \end{aligned}$$

This leads to a key property of Adam in that the learning rate is scaled for each parameter. The Adam optimiser was also used in the source code provided with the LSTM model in *Wang et al.* [36].

II.G Mixture Models

A commonly used technique to represent distributions is a Mixture Model *Trevor Hastie and Friedman* [34]. Mixture models use the sum of a chosen number of individually weighted independent distributions.

Definition II.1. Mixture Model A mixture model with $n \geq 1$ distributions ϕ defined by a parameter vector $\boldsymbol{\theta}$ and weights $\alpha_i \in [0, 1]$ such that $\sum_{i=1}^{n} \alpha_i$ has probability mass function

$$f(x; \boldsymbol{\alpha}, \boldsymbol{\theta}, n) = \sum_{i=1}^{n} \alpha_i \phi(x; \boldsymbol{\theta}_i)$$

It has been shown that mixtures can get arbitrarily close to a given distribution for a finite number of distributions in the continuous case *Nguyen et al.* [26],*Nguyen and McLachlan* [25]. It is immediately obvious for the bounds on the number of distributions is at least as many as the modality of the output divided by the modality of our chosen distribution and at most the number of bins in the discrete case. The lower bound comes from the fact

that our mixture can have at modality of at most the number of mixtures with the used distributions being of modality 1. Our upper bound comes from the used distributions having location and scale parameters we can change independently. This leads to the obvious exact model consisting distributions with means inside each bin and weight α_i the probability of that bin and a variance of almost zero.

$$var(X) = 0 = \mathbb{E}((X - \mathbb{E}(X))^2) \tag{1}$$

$$(X - \mathbb{E}(X))^2 \geq 0 \implies \mathbb{P}((X - \mathbb{E}(X))^2 \neq 0) = 0 \tag{2}$$

$$\implies X = \mathbb{E}(X) \tag{3}$$

This outlines our ability to get the exact distribution if we have variance 0 in the discrete case. A model like this however, will likely overfit the data so in practice you wouldn't want to use this many distributions in the Mixture Model.

II.G.1 Gaussian Mixture Model

Where Mixture Models are applied the most commonly used distribution is the Normal distribution $\mathcal{N}(\mu, \sigma)$ *Trevor Hastie and Friedman* [34]. The attractiveness of using a Gaussian mixture are its use on many problems shows it to be effective and it's easily interpreted. We do however, as the Gaussian distribution is continuous have to discretise by using the value of the CDF across each bin. As the support of this distribution is $x \in \mathbb{R}$ we ignore the probability outside the range covered by the bins. This can cause problems in the faith we have in divergence metrics that aren't symmetrical such as the Kullback-Lieber Divergence as the model doesn't necessarily sum to 1 across the range covered by the bins. However, as the model converges towards its solutions and distance metrics such as the L1 Norm, MSE and Hellinger Distance reduce we can have more faith in the values of non-symmetric divergence measures. This also motivates the use of discrete distributions that can also have their mean and variance varied as alternative distributions in our Mixture Model. We show below an example of this Mixture Model with 3 distributions 3.

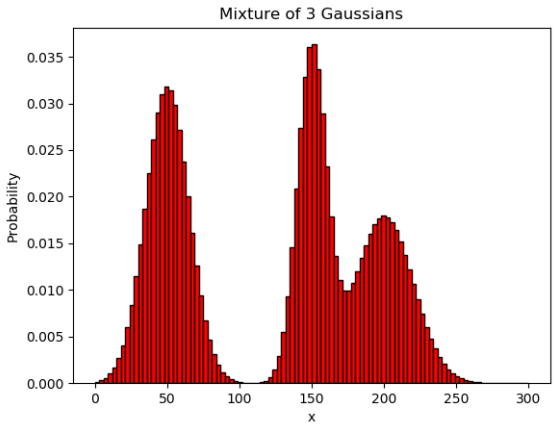

Figure 3: A mixture of 3 Gaussians with parameters $\mu = (50, 150, 200), \sigma = (15, 10, 20)$ and weights $(0.4, 0.3, 0.3)$ for bins of width 3 from 0 to 300

II.G.2 Gamma-Poisson Mixture Model

The Gamma-Poisson distribution has become an attractive distribution due to its ability to account for overdispersion from a Poisson distribution originally developed by *Bates and Neyman* [3] and has seen been applied to a regulating gene by *Amrhein et al.* [2] and mixtures to RNA-Seq and DNA methylation data by *Li et al.* [22]. The intuition behind the Gamma-Poisson model is to have a Poisson distribution $X|\lambda \sim Poi(\lambda)$with it's rate parameter Gamma distributed $\lambda \sim Gamma(\alpha, \beta)$ with $X \in \mathbb{N}_0, \alpha > 0, \beta > 0$. Allowing us to change the variance and the mean of the model independently unlike with a Poisson distribution.The support is exactly the values that the distribution from our regulating gene model can take and most models in biology can take being $\mathbb{N}_0$.

We can derive the PMF and CDF simply.

$$\begin{aligned}
f(x;\alpha,\beta) &= \int_0^\infty P(X=x|\lambda)P(\lambda|\alpha,\beta)d\lambda \\
&= \int_0^\infty \frac{\lambda^x e^{-\lambda}}{x!}\frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda}d\lambda \\
&= \frac{\beta^\alpha}{\Gamma(\alpha)\Gamma(x+1)}\int_0^\infty \lambda^{x+\alpha-1}e^{-\lambda(1+\beta)}d\lambda \\
&= \frac{\Gamma(\alpha+x)\beta^\alpha}{\Gamma(\alpha)\Gamma(x+1)(1+\beta)^{\alpha+x}} \\
F(x;\alpha,\beta) &= \int_0^\infty P(X\leq x|\lambda)P(\lambda|\alpha,\beta)d\lambda \\
&= \int_0^\infty \sum_{k=0}^{\lfloor x\rfloor}\frac{\lambda^k e^{-\lambda}}{k!}\frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda}d\lambda \\
&= \sum_{k=0}^{\lfloor x\rfloor}\frac{\beta^\alpha}{\Gamma(\alpha)\Gamma(k+1)}\int_0^\infty \lambda^{k+\alpha-1}e^{-\lambda(1+\beta)}d\lambda \\
&= \sum_{k=0}^{\lfloor x\rfloor}\frac{\Gamma(\alpha+k)\beta^\alpha}{\Gamma(\alpha)\Gamma(k+1)(1+\beta)^{\alpha+k}}
\end{aligned}$$

The integration comes from a substitution and applying the definition of the Gamma function, we use the identity $\Gamma(x) = (x-1)!$ for positive integers, the interchanging of the sum and the integral comes from Tonelli's theorem the requirements of which are satisfied by all distributions.

Definition II.2. Gamma-Poisson We define the Gamma-Poisson distribution with $\alpha \in (0,\infty), \beta \in (0,\infty)$ and support $x \in \mathbb{N}_0$

$$f(x;\alpha,\beta) = \frac{\Gamma(\alpha+x)\beta^\alpha}{\Gamma(\alpha)\Gamma(x+1)(1+\beta)^{\alpha+x}}$$

To find the moments to show the variance isn't a function of the mean like Poisson we construct the moment generating function using that $X|\lambda \sim Poi(\lambda)$ and $\lambda \sim Gamma(\alpha,\beta)$ their known moment generating functions.

$$\begin{aligned}
M_{X|\lambda}(t) &= \exp\left(\lambda\left(e^t-1\right)\right) \\
M_X(t) &= E(M_{X|\lambda}(t)) \\
\implies M_X(t) &= E\left(\exp\left(\lambda\left(e^t-1\right)\right)\right) \\
&= M_\lambda\left(e^t-1\right) \\
&= \left(1-\frac{e^t-1}{\beta}\right)^{-\alpha}
\end{aligned}$$

We can use this to find the moments and show that we can control vary both the mean and variance with the parameters.

$$\begin{aligned}
E(X) &= \frac{\alpha}{\beta} \\
E(X^2) &= \frac{\alpha(\beta+1)+1}{\beta^2}
\end{aligned}$$

It is also important to note that this distribution is equivalent to the Negative Binomial distribution with a change of variables $\alpha = r, \beta = \frac{1-p}{p}$. Negative Binomial Mixture Models have seen use with some success.

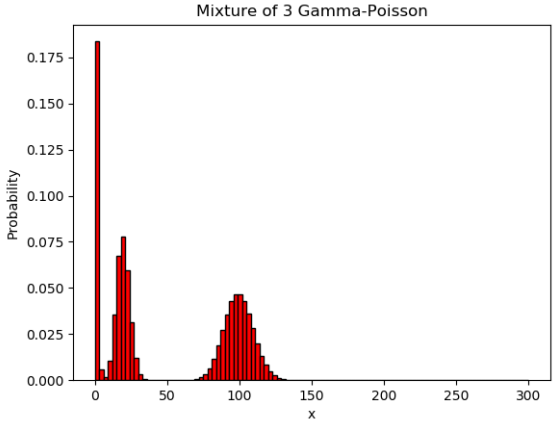

Figure 4: A mixture of 3 Gamma-Poissons with model parameters $\alpha = (5000, 300, 1000), \beta = (50, 300, 50)$ and weights $(0.4, 0.3, 0.3)$ for bins of width 3 from 0 to 300

II.H Bivariate Gamma-Poisson Mixture Model

We can extend this to the bivariate case by using the product of two Gamma-Poisson distributions with their weight parameters weighted by p_1, p_2 as suggested in *Nelson* [24]. This leads to the probability distribution.

$$\begin{aligned} f(X_{12} = (i,j); \alpha, \beta, p) &= \int_0^\infty P(X_1 = i|p_1\lambda)P(X_2 = j|p_2\lambda)P(\lambda|\alpha,\beta)d\lambda \\ &= \frac{\Gamma(i+j+\alpha)\beta^\alpha p_1^i p_2^j}{\Gamma(\alpha)\Gamma(i+1)\Gamma(j+1)(1+\beta)^{\alpha-(i+j)}} \end{aligned}$$

Definition II.3. Bivariate Gamma-Poisson We define the Bivariate Gamma-Poisson distribution with $\alpha \in (0,\infty), \beta \in (0,\infty), p_1 \in (0,1], p_2 \in (0,1]$ with $p_1 + p_2 = 1$ and support $i, j \in \mathbb{N}_0$

$$f(i,j;\alpha,\beta,p) = \frac{\Gamma(i+j+\alpha)\beta^\alpha p_1^i p_2^j}{\Gamma(\alpha)\Gamma(i+1)\Gamma(j+1)(1+\beta)^{\alpha-(i+j)}}$$

A suggested extension to this model from *Nelson* [24] that we use is the Dirichlet-Multinomial Gamma-Poisson Model. We model the weights p_1, p_2 with the Dirichlet distribution with parameters θ_1, θ_2 this leads to the the probability distribution. A graphical plot of this distribution is shown in figure 5.

$$\begin{aligned} f(X_{12} = (i,j); \alpha, \beta, \theta) &= \int_0^\infty \int_0^1 P(X_1 = i|p_1\lambda)P(X_2 = j|p_2\lambda)P(\lambda|\alpha,\beta)P(P|\theta)dPd\lambda \\ &= \frac{\Gamma(i+j+\alpha)\Gamma(\theta_1+\theta_2)\Gamma(\theta_1+i)\Gamma(\theta_2+j)\Gamma(i+1)\Gamma(j+1)\beta^\alpha}{\Gamma(\alpha)\Gamma(\theta_1+\theta_2+i+j)\Gamma(\theta_1)\Gamma(\theta_2)(1+\beta)^{\alpha-(i+j)}} \end{aligned}$$

Definition II.4. Dirichlet-Multinomial Gamma-Poisson We define the Dirichlet-Multinomial Gamma-Poisson distribution with $\alpha \in (0,\infty), \beta \in (0,\infty), \theta_1 \in (0,\infty), \theta_2 \in (0,\infty)$ and support $i, j \in \mathbb{N}_0$

$$f(i,j;\alpha,\beta,\theta) = \frac{\Gamma(i+j+\alpha)\Gamma(\theta_1+\theta_2)\Gamma(\theta_1+i)\Gamma(\theta_2+j)\Gamma(i+1)\Gamma(j+1)\beta^\alpha}{\Gamma(\alpha)\Gamma(\theta_1+\theta_2+i+j)\Gamma(\theta_1)\Gamma(\theta_2)(1+\beta)^{\alpha-(i+j)}}$$

An important feature we would like is the ability to model the correlation between the two variates. We apply the law of total covariance to get.

$$\begin{aligned} COV(X_1, X_2) &= \mathbb{E}(COV(X_1, X_2|p_1\lambda, p_2\lambda)) + COV(\mathbb{E}(X_1|p_1\lambda), \mathbb{E}(X_2|p_2\lambda)) \\ &= \mathbb{E}(\mathbb{E}(X_1X_2|p_1\lambda, p_2\lambda) - \mathbb{E}(X_1|p_1\lambda)\mathbb{E}(X_2|p_2\lambda)) + COV(p_1\lambda, p_2\lambda) \\ &= 0 + \mathbb{E}(\lambda^2)COV(p_1, p_2) \\ &= -\frac{\alpha(1+\beta)}{\beta^2}\frac{\theta_1\theta_2}{(\theta_1+\theta_2)^2(\theta_1+\theta_2+1)} \end{aligned}$$

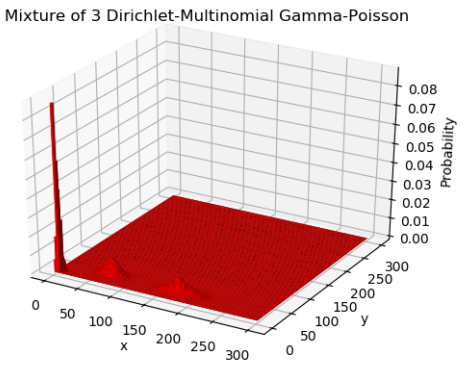

Figure 5: A mixture of 3 Dirichlet-Multinomial Gamma-Poissons with model parameters $\alpha = (1000, 5000, 60000), \beta = (5, 50, 6000), \theta_1 = (25171, 36770, 133977), \theta_2 = (179284, 90339, 127516)$ and weights $(0.4, 0.3, 0.3)$ for bins of width 3 from 0 to 300 in both dimensions

II.I Poisson Log-Normal Mixture Model

A distribution that has seen use in modelling in biology with species abundance *Bulmer* [9], microbial counts in food *Gonzales-Barron and Butler* [17] and other settings is the Poisson Log-Normal distribution. Proposed by *AITCHISON and HO* [1] as a way to better model variation and correlation structures when applied to higher dimensions. We model the rate parameter λ of the Poisson distribution by a Log-Normal distribution. A graphical plot of a mixture of Poisson Log-Normals is shown in figure 6.

Definition II.5. Poisson Log-Normal We define the Poisson Log-Normal distribution with μ, σ and support $x \in \mathbb{N}_0$ where $P(\lambda|\mu, \sigma)$ is the PDF of the Log-Normal distribution as

$$f(x; \mu, \sigma) = \int_{\mathbb{R}^+} P(X = x|\lambda) P(\lambda|\mu, \sigma) d\lambda$$

Definition II.6. Log-Normal We define the Log-Normal distribution with μ, σ and support $x \in \mathbb{R}^+$

$$f(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\log(x) - \mu)^2}{2\sigma^2}\right)$$

Unfortunately this integral has no analytical solution so we rely on numerical methods to evaluate the PDF. An easy and effective method that works in the univariate case is quadrature based on the quartiles of the Log-Normal distribution. Sadly this doesn't work in higher dimensions due to the inverse CDF in the multivariate case having more than one solution. As the Log-Normal distribution is a transform of the Normal distribution we can define the inverse CDF as the inverse transform of the the inverse normal CDF.

$$\begin{aligned} \Phi_{LN}^{-1}(p) &= \exp(\Phi_N^{-1}(p)) \\ &= \exp(\mu + \sigma\sqrt{2} erf^{-1}(2p - 1)) \end{aligned}$$

We then for n quadrature points take n+1 points p_i in (0,1) we don't take p=0,1 as this can lead to inf or NaN values. We evaluate the LN distribution at these values and take the pairwise average to give λ_i. For higher dimensions as suggested in *AITCHISON and HO* [1] multivariate Hermitian integration can be used.

$$f(x; \mu, \sigma) \approx \sum_{i=1}^{n} \frac{1}{n} f_{Po}(x; \lambda_i)$$

We can find its nth moment easily by noticing if X is Log-Normal then $Y = \log(X)$ is normal

$$\mathbb{E}(X^n) = \mathbb{E}(\exp(nY)) = \int_{\mathbb{R}} \exp(ny) \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{(y - \mu)^2}{2\sigma^2}\right) dy$$

This is the moment generating function of the normal distribution which we know to be

$$\mathbb{E}(X^n) = \exp\left(n\mu + \frac{1}{2} n^2 \sigma^2\right)$$

The expectation and variance of this distribution is calculated easily as

$$\mathbb{E}(X) = \exp\left(\mu + \frac{1}{2}\sigma^2\right)$$

$$\begin{aligned} var(X) &= \exp\left(2\mu + 2\sigma^2\right) + \left(\exp\left(\mu + \frac{1}{2}\sigma^2\right)\right)^2 \\ &= \exp(2\mu + \sigma^2)\left(\exp(sigma^2) + 1\right) \end{aligned}$$

Allowing us to change the variance and mean independently.

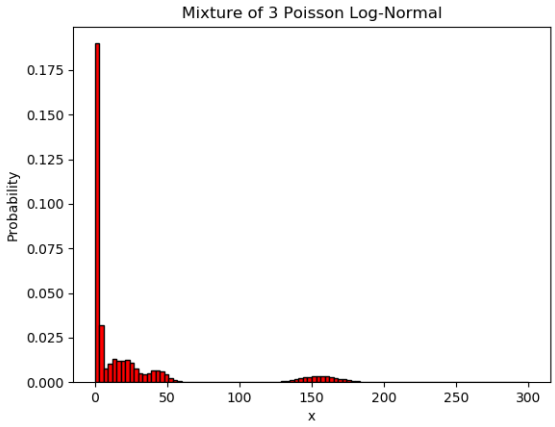

Figure 6: A mixture of 3 Poisson Log-Normal distributions with model parameters $\mu = (-1, 4, 2), \sigma = (3, 3, 1)$ and weights $(0.4, 0.3, 0.3)$ for bins of width 3 from 0 to 300

II.J Reaction Networks

Definition II.7. Reaction Network A reaction network consisting of a set of species $X_1, \ldots, X_N$ with abundances $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ that interact via a set of R reactions of the form

$$\nu_{1,r}^- X_1 + \cdots + \nu_{N,r}^- X_N \xrightarrow{w_r(x)} \nu_{1,r}^+ X_1 + \cdots + \nu_{N,r}^+ X_N$$

where $r = 1, \ldots, R$, and the stoichiometric coefficients $\left(\nu_{ir}^{\pm}\right)_{i=1,\ldots,N}$ are non-negative integers and $w_r(x) \geq 0$ is the reaction rate.

This leads us to a system of ODEs.

Definition II.8. Deterministic rate equation The deterministically modelled reaction network is given by the system of ODEs

$$\frac{dx_i}{dt} = \sum_{r=1}^{R} (\nu_{ir}^+ - \nu_{ir}^-) w_r(x) \text{ for } i = 1, \ldots, N$$

called the rate equations which are solved subject to the initial condition $x(0) = x_0$

An important definition we will use in the derivation of analytical solutions is

Definition II.9. Mass-action Kinetics The vector of reaction rates $\mathbf{w}(x)$ using mass action kinetics are given by

$$w_r(x) = k_r \prod_{i=1}^{N} x_i^{\nu_{ir}^-} \text{ for } r = 1, \ldots, R$$

where k_r is the reaction rate constant

Not all systems are solvable analytically such as the regulating gene model and thus if we want a deterministic solution we would be required to use a numerical ODE solver.

II.J.1 Deterministic

We can think of a solution to our problem in a deterministic way without any stochasticity. An example where we can find an analytical deterministic solution is population explosion. Consider the two sexes (male and female) with equal rates of reproduction b and mortality d

$$X + Y \xrightarrow{b} X + 2Y,\ X + Y \xrightarrow{b} 2X + Y,\ X \xrightarrow{d} \emptyset,\ Y \xrightarrow{d} \emptyset$$

Using Mass-action Kinetics this forms the system of ODEs

$$\frac{dx}{dt} = bxy - dx,\ \frac{dy}{dt} = bxy - dy$$

If we chose the initial conditions $x(0) = y(0)$, clearly $x(t) = y(t) \forall t$ and there is no distinction between female and male populations. Thus

$$\frac{dx}{dt} = bx^2 - dx$$

which is a Bernoulli differential equation which has known exact solutions. In this case it is

$$x(t) = \frac{dx(0)}{(d - bx(0))e^{dt} + bx(0)}$$

However, this solution does not capture any of the stochastic nature of these systems which we would expect to exist. If we compared our analytical solution to real world data we would likely see heterogeneity that isn't accounted for by the deterministic setting.

II.J.2 Stochastic

We can seek to model the reactions in a more realistic stochastic formulation. As the number of events is necessarily discrete this is a counting process.

Definition II.10. Counting process $Y(t)$ is a counting process if $Y(t)$ is constant except for jumps of size ± 1, whose paths are cadlag, i.e., they are right-continuous with left-limits $Y(t^-) = \lim_{s \to t} Y(s)$.

The simplest counting process is the Poisson Process and we can often interpret simple reaction networks as Poisson Processes.

Definition II.11. Poisson Process A counting process is a Poisson Process with rate λ if for $t < s$, it holds that

$$Pr(Y(s) - Y(t) = k) = \lambda^k \frac{(s-t)^k}{k!} e^{-\lambda(s-t)}$$

We also make a modification to the law of Mass-action for the stochastic case and give the Chemical Master Equation the proof of which comes from Dykin's formula.

Definition II.12. Mass-action Kinetics for stochastic reaction networks The vector of reaction rates $\mathbf{w}(x)$ using mass action kinetics are given by

$$w_r(x) = k_r \prod_{i=1}^{N} \frac{x_i!}{(x_i - \nu_{ir}^-)!} \text{ for } r = 1, \ldots, R$$

where k_r is the reaction rate constant

Definition II.13. Chemical Master Equation The time-dependent probability of the stochastic reaction network $p(z,t) = Pr(x(t) = z)$ satisfies the Chemical Master Equation:

$$\frac{dp(z,t)}{dt} = \sum_{r=1}^{R} [w_r(z - \nu_r)p(z - \nu_r) - w_r(z)p(z)]$$

The Chemical Master Equation is often very difficult to solve analytically if it is even possible. An example that exhibits a closed form solution is a Linear birth-death process under the detailed balance equation, thus giving us the stationary distribution.

$$\emptyset \xrightarrow{a} x,\ x \xrightarrow{b} \emptyset,\ w_+(x) = a,\ w_-(x) = bx$$

Where $w_+(x)$ and $w_-(x)$ are the birth and death rates respectively giving us the Chemical Master Equation:

$$\frac{dp(x,t)}{dt} = w_+(x-1)p(x-1) - w_+(x)p(x) + w_-(x+1)p(x+1) - w_-(x)p(x)$$

Assuming stationarity and using the detailed balance condition we get

$$w_+(x-1)p(x-1) = w_-(x)p(x)$$

Giving us

$$p(x) = p(0)\prod_{i=0}^{x}\frac{w_+(i-1)}{w_-(i)}$$

Which exists if

$$\frac{1}{p(0)} = \sum_{x=0}^{\infty}\prod_{i=0}^{x}\frac{w_+(i-1)}{w_-(i)} < \infty$$

Substituting in for the reaction rates we get

$$p(x) = \exp\left(-\frac{a}{b}\right)\frac{1}{x!}\left(\frac{a}{b}\right)^x$$

Which is Poisson with rate $\frac{a}{b}$. The solution if we looked at the Linear birth-death process as deterministic would be.

$$\begin{aligned}\frac{dx}{dt} &= a - bx \\ \implies x_t &= e^{-bt}x_0 + \frac{a}{b}(1 - e^{-bt})\end{aligned}$$

Although, in general solutions like this aren't obtainable analytically such as in our case of a self-regulating gene and thus this leads us to the need for a simulation algorithm.

II.J.3 Gillespie Algorithm

A exact simulation algorithm exists in the Gillespie Allgorithm from *Gibson and Bruck* [16]. The algorithm exists in two equivalent formulations below we describe the algorithm for the Direct Method which calculates explicitly which reaction occurs next and when it occurs. We consider a system of r reactions each with an associated rate constant $k_1, \ldots, k_r$ these can be a true constant or have time variance. The system is in exactly one state at each time step. This leads that at a given time step the system has r possible transitions represented by each reaction from its state. Using a random number generator with the correct distribution we can choose the next state from the r possible transitions. We specify for the next reaction μ and its occurrence time τ a probability density $P(\mu, \tau)$. This can be shown to be

$$P(\mu,\tau)d\tau = \alpha_\mu \exp\left(-\tau\sum_j \alpha_j\right)d\tau$$

Where α_i is the propensity function of reaction i derived from the law of mass action. We can obtain the the probability of each reaction by integrating over 0 to ∞

$$Pr(\text{Reaction} = \mu) = \frac{\alpha_\mu}{\sum_j \alpha_j}$$

The distribution for the times can be found by summing over all μ

$$Pr(\tau)d\tau = \left(\sum_j \alpha_j\right)\exp\left(-\tau\sum_j \alpha_j\right)d\tau \sim Exp\left(\sum_j \alpha_j\right)$$

We formulate this into the following algorithm

Algorithm 1 Direct Method Gillespie

Initialise Set initial numbers of species, stop time t_{MAX} and $t = 0$
while $t < t_{MAX}$ **do**
  Calculate the propensity $\alpha_i \ \forall i = 1, \ldots, r$
  Choose the next reaction μ according to the distribution $Pr(\text{Reaction} = \mu)$
  Choose the next timestep τ according to the distribution $Exp\left(\sum_j \alpha_j\right)$
  Change the number of molecules to reflect reaction μ occuring and $t \leftarrow t + \tau$
end while

The equivalent and faster method of the Gillespie First Reaction Method using the previously defined for the Direct Method.

Algorithm 2 First Reaction Method Gillespie

Initialise Set initial numbers of species, stop time t_{MAX} and $t = 0$
while $t < t_{MAX}$ **do**
  Calculate the propensity $\alpha_i \ \forall i = 1, \ldots, r$
  for $i = 1, \ldots, r$ **do**
    Generate a putative time τ_i according to $Exp\left(\sum_j \alpha_j\right)$
  end for
  Choose $\mu = \text{argmin}_{i \in \{1,\ldots,r\}}(\tau_i)$ set $\tau = \tau_\mu$
  Change the number of molecules to reflect reaction μ occurring and $t \leftarrow t + \tau$
end while

This allows us to generate samples from the distribution. Even for small reaction networks such as a regulating gene the generation of a useful number of independent samples is time consuming and for larger networks or for greater exploration of the reaction rate space it can be prohibitive. Faster non-exact algorithms exist such as tau-leaping shown in *Gibson and Bruck* [16] but these methods although faster still require a large amount of time to explore the reaction rate space of large reaction networks.

II.K Ensemble Models

We suggest a few possible ensembles as efforts to improve upon our models individual performance. An important requirement to build an ensemble with additional predictive power over the individual models is variation in the predictions of the submodels. This is rather obvious as if all our models predicted distributions that are arbitrarily close to each other there would be no need for an ensemble. As neural networks are stochastic in nature with random initial weights and using random batches to train on there exists variation between our trained models which is how an ensemble was constructed in *Wang et al.* [36] but in an effort to increase this variation we use bagging proposed by *Breiman* [8]. Bagging splits the dataset into a training data set and test data set as before but we train k submodels on a different random sample with replacement of the training dataset. This increases variation between the models by increasing and decreasing at random the occurrence of different patterns in the dataset. We then use a dense layer or multiple layers to take the predictions of each submodel and form a new prediction. Ensemble methods have frequently been shown to increase the predictive ability of an individual model provided there exists variation between the models as shown in *Wang et al.* [36] and *Breiman* [8].

III Alternative Emulation Methods

Alternative methods exist in the Bayesian setting with approximate Bayesian computation *Beaumont* [5]. These methods involve sampling from a parameter prior distribution then sampling a posterior distribution based on the prior and accepting or rejecting the simulated posterior based on summary statistics of the distribution or the full distribution. These methods require a large number of random samples which are often computationally expensive and the complexity increases greatly as dimensionality increases. Improvements upon these methods using Markov Chain Monte-Carlo *Beaumont* [4] and sequential Monte-Carlo *Sisson et al.* [32] techniques have been proposed but still require a large number of time consuming model simulations. Further suggested improvements include the use of regression or Gaussian Process emulation of these models where simulation is difficult *Wilkinson* [37].

The choice of parameters greatly affect the performance of such models. Also the choice of summary statistics can greatly affect performance and there is no guarantee we can find a finite dimensional number of summary statistics that encompass all the information of the distribution exist outside of the exponential family *Robert et al.* [28]. As such these methods can be too complex and time consuming for problems of interest as such an alternative method less effected by dimensionality and require generating random numbers would be useful.

IV Proposed Long Short Term Memory (LSTM) Model

We propose using a model based on *Wang et al.* [36] consisting first a series of 5 dense layers of $64, 128, 256, 128, 64$ neurons to learn general model characteristics from the parameters. Then we split this off into a dense layer for each T time steps which feeds into one or two LSTM layers depending on if it is bidirectional. Taking in the p previous time steps predicted by the LSTM array. This allows the model to predict based off previous points in the curve. The outputs of the LSTM arrays are fed into a dense layer to give the T points. The model structure we use is shown in figure 7. We can use any activation we want for the dense layers at the start but for the final dense layer we use Softmax that gives an output that sums to 1. This model also allows for the use of both the Standard LSTM Cell and the Peephole LSTM Cell.

To form an ensemble of these models we train a given number of models n on random samples of a training dataset and then take their outputs to input into a dense layer with an input dimension of nT and output dimension of T. We use a Softmax activation function as our activation function as in the submodels.

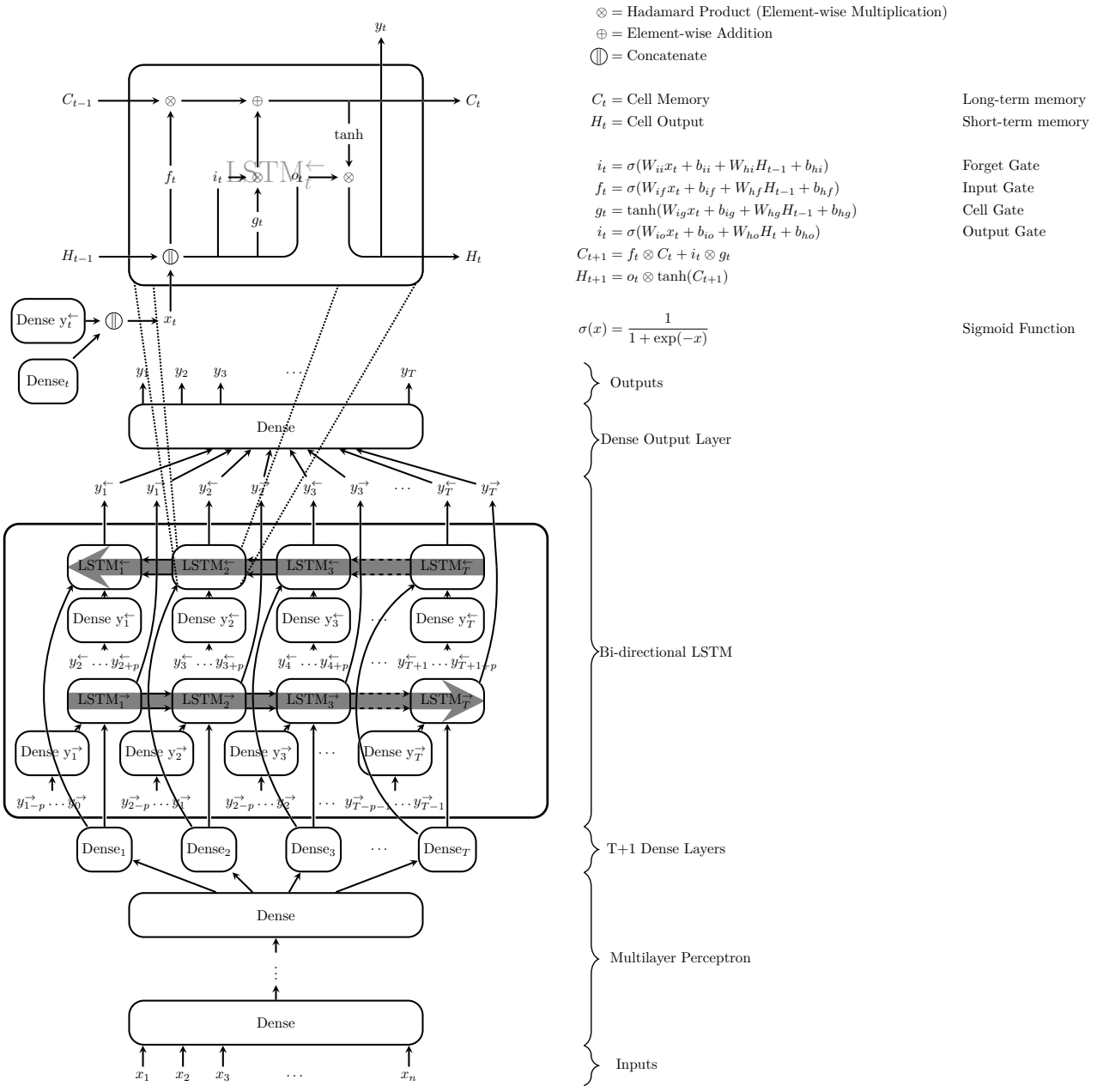

Figure 7: General structure for the Bidirectional LSTM Model

V Proposed Mixture Model

We utilise a feed foward neural network to predict the distribution parameters $\boldsymbol{\theta}$ and the distribution weights α given the input which in the case of the replicating gene model is the reaction rates. We cannot calculate a loss directly from the parameters so we construct our estimate for each bin using the CDF of our chosen distribution $\Phi(x; \boldsymbol{\theta}_i)$. We would most like to choose distributions that have the same or similar support to the distribution we are estimating and at least the ability to change the first and second moments without them being a function of each other that is the case for the proposed distributions.

Say our distribution takes p parameters $(\theta^{(1)} \dots \theta^{(p)} \in \mathbb{R}^n)$ and there are r inputs for the neural network then a general diagram is shown in figure 8. The complexity of model is invariant to the number of bins which is useful if we want to train with lots of small bins. These models also extend to the multivariate case if we use a mixture of multivariate distributions. This would allow for the modelling of the joint distribution of multiple species if we wish with a minimal increase in the model's complexity. The dense layers before the output layer attempt to learn the general mapping from the inputs to the distribution parameters increasing its depth and size can increase the models ability to learn more complex mappings. We would therefore expect more complex reaction networks to require this to be deeper. In our experiments we use 5 layers of $64, 128, 256, 128, 64$ neurons. The number of mixtures controls the maximum complexity of the output mixture thus increasing the number of mixtures may not increase the model's performance if the initial dense layers can't learn a good mapping or if the distribution is not appropriate. The activation of the dense layers can be any that works best on a given problem but the activation functions of the output layers must be chosen carefully to map to the values allowed for that parameter or a subset of those values. The activation for the layer predicting the weights α is Softmax as this enforces the summing to unity and values in the range $[0, 1]$.

Using neural networks as a way to get the parameters of a Gaussian Mixture Model has seen some limited use but on rather different and more simplistic problems such as the mapping of an inversion of a function with noise to obtain the conditional probability of the input of the function given the output *Bishop* [7]. Taking a mean, mode or the mean of the most heavily weighted distribution in the mixture as the output depending in the application. It predicted the inputs of highly non-linear function given the outputs much better than a simple feed forward neural network.

To form ensembles of this model we use a series of dense layers that takes the outputs of all the submodels into a dense layer. This will be of dimension $n(p+1)d$ where d is the sum of the dimensionality of each model parameter. We can feed this through multiple dense layers before as in the individual models we have a dense layer for each parameter with appropriate activation functions.

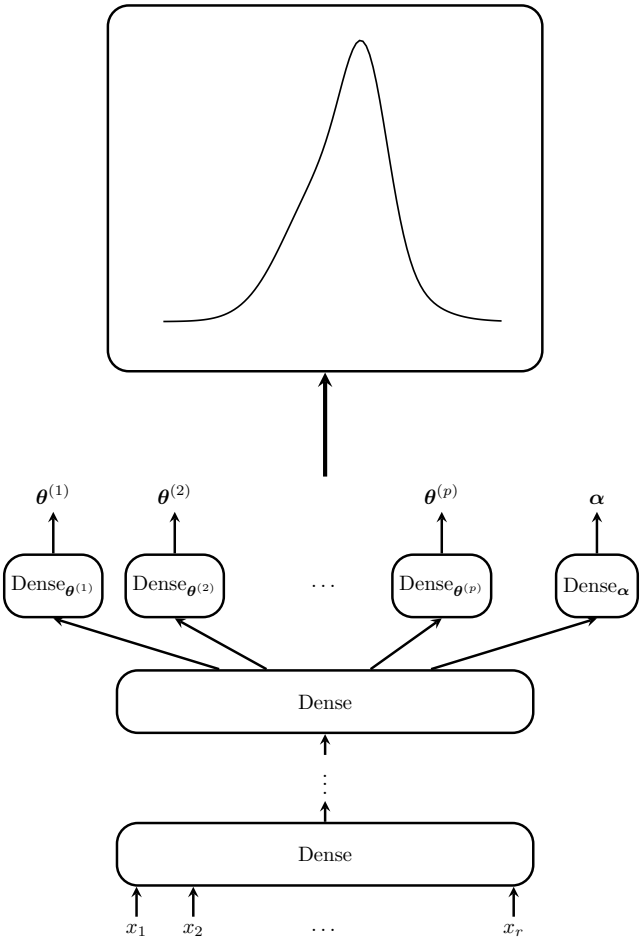

Figure 8: General structure of the Mixture Model

VI Loss Functions

To show the vanishing gradient problem caused by common loss functions and show alternatives which offer larger gradients for our problem we construct 3D surface plots to show their shape in figure 9. We use T points and a batch size of B when defining the loss functions.

VI.A Mean Squared Error

The mean squared error is commonly used in regression problems as it is twice differentiable and the gradient reduces as $x_t \to y_t$.

Definition VI.1. MSE The Mean Squared Error for $X, Y \in \mathbb{R}^{B\text{x}T}$ of B batches and T points

$$MSE(X,Y) = \frac{1}{B}\frac{1}{T}\sum_{i=1}^{B}\sum_{t=1}^{T}(y_t^{(i)} - x_t^{(i)})^2$$

We can see visually that the gradient becomes very small for the values we expect in figure 9. Also remembering most values will be closer to 0 than 1 as we don't expect y_t to be uniformly distributed in its interval.

VI.B L1

One common alternative is the L1 Norm but a disadvantage of this approach is this function is not twice differentiable and thus can lead to the model bouncing around optimal solutions. The plot of this is shown in figure 9.

Definition VI.2. L1 The L1 Norm for $X, Y \in \mathbb{R}^{B\text{x}T}$ of B batches and T points

$$L1 = \frac{1}{B}\frac{1}{T}\sum_{i=1}^{B}\sum_{t=1}^{T}|y_t^{(i)} - x_t^{(i)}|$$

VI.C Squared Hellinger Distance

We propose as an alternative measure of loss the Squared Hellinger Distance. This is a f-divergence measure *Eguchi* [13], it's twice differentiable and is a measure of distance due to it being symmetric. The attractiveness of this measure is it has similar features to the MSE with it's gradient becoming smaller as it approaches its minimum but its gradient is much larger than that of the MSE in the domain of our problems. We do however need to be careful that when the model predicts a 0 as when calculating the gradient during back propagation this leads to NaN due to a division by zero. This is easily resolved by not letting the prediction fall bellow a small $\epsilon > 0$ and also doing this for the true value y. This effectively means if they are both below ϵ we assume they are both equal and zero.

Definition VI.3. H^2 The Squared Hellinger Distance for $X, Y \in \mathbb{R}^{B\text{x}T}$ of B batches and T points

$$H^2(X,Y) = \frac{2}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}\left(\sqrt{y_t^{(i)}} - \sqrt{x_t^{(i)}}\right)^2$$

We can check this is a suitable loss by first checking the differential w.r.t x.

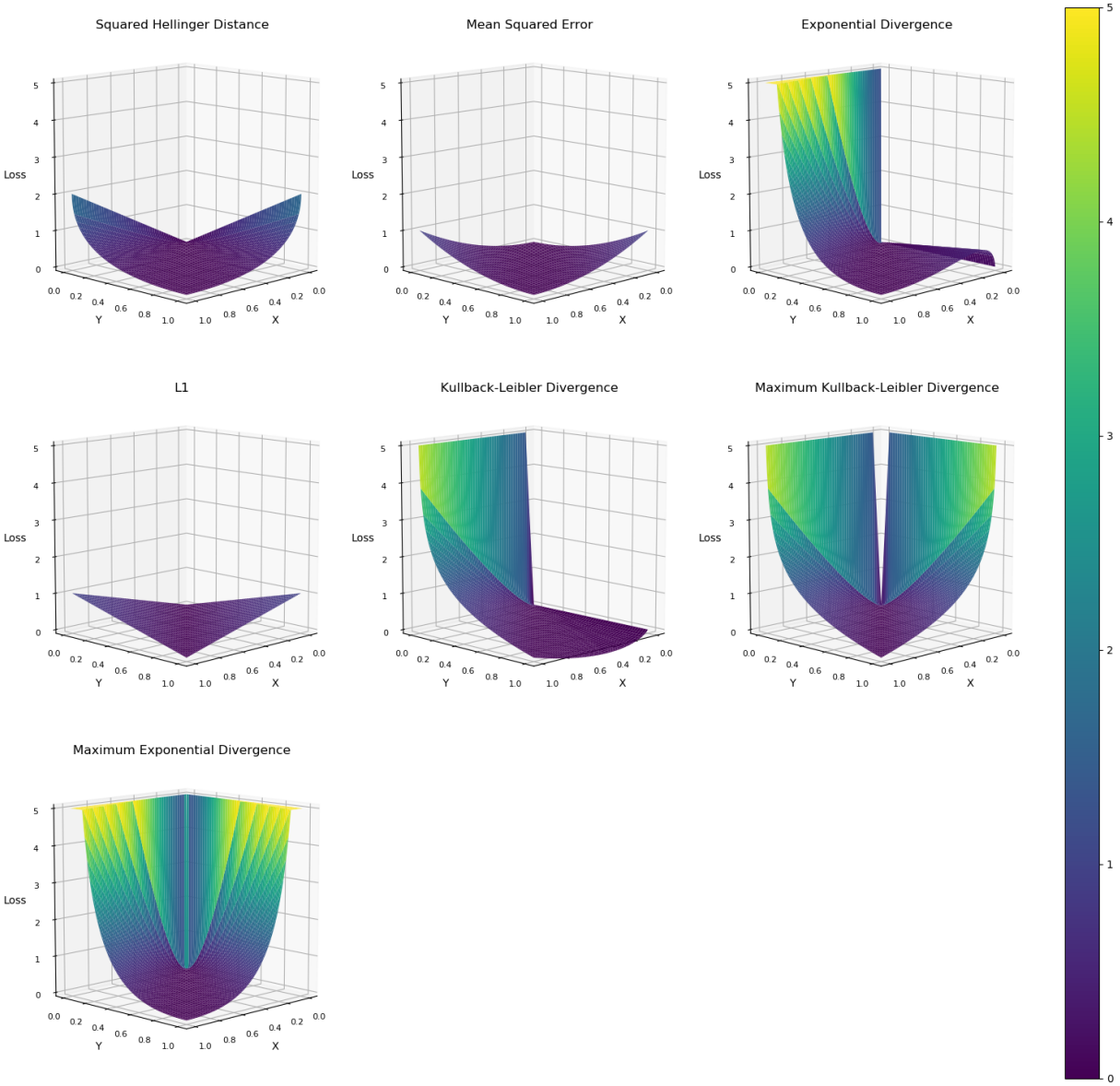

Figure 9: Proposed loss functions plotted for $x \in [0, 1], y \in [0, 1]$ plotted up to a maximum loss of 5. The colour map has the same scale for all the plots to show their differences in terms of magnitude. This also clearly shows the existance of other minimas in the divergence measures and their lack of symmetry.

$$\frac{\partial H^2}{\partial x_t} = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}\left(1 - \frac{\sqrt{y_t^{(i)}}}{\sqrt{x_t^{(i)}}}\right)$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T} y_t^{(i)} = \sum_{i=1}^{B}\sum_{t=1}^{T} x_t^{(i)} \implies \frac{\partial H^2}{\partial x_t} = 0$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T} y_t^{(i)} > \sum_{i=1}^{B}\sum_{t=1}^{T} x_t^{(i)} \implies \frac{\partial H^2}{\partial x_t} < 0$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T} y_t^{(i)} < \sum_{i=1}^{B}\sum_{t=1}^{T} x_t^{(i)} \implies \frac{\partial H^2}{\partial x_t} > 0$$

We also check that the loss function is convex by looking at the 2$^{\text{nd}}$ differential and showing it is greater than or equal to 0 for the entire domain this is also implied by it being an f-divergence measure *Eguchi* [13]. This shows the minimum is a global minimum for the entire domain.

$$\frac{\partial^2 H^2}{\partial x_t} = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=0}^{T}\frac{\sqrt{y_t^{(i)}}}{2(x_t^{(i)})^{\frac{3}{2}}} \geq 0$$

We also show a plot of this loss function in figure 9.

VI.D Kullback-Leibler Divergence

We also propose the use of the KL Divergence. This is a twice differentiable function and as a f-divergence measure it's a useful metric for evaluating the performance of the model and is convex.

Definition VI.4. KLD The Kullback-Leibler Divergence for $X, Y \in \mathbb{R}^{B\text{x}T}$ of B batches and T points

$$KLD(X,Y) = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}(x_t^{(i)})\log\left(\frac{x_t^{(i)}}{y_t^{(i)}}\right)$$

One problem with this loss function is it requires x and y to be from a proper probability distribution or it can give negative values leading to convergence to incorrect minima. To be able to use it as a loss function we must normalise the output of our model so it sums to 1. We also as we are taking logs we use clamp to enforce a minimum to prevent log(0). A plot of this function is shown in figure 9. It is important to note that this function is not symmetric so not a distance this can make its value harder to interpret. The existence of this other minimum makes it not useful as a loss function to give to our optimiser as we can get convergence to the wrong minimum. In the current implementation of Pytorch 1.5.0 there exists overflow errors causing negative values that are attributed to it requiring a log input which have been fixed in the nightly version and will be included in the next release.

VI.E Exponential Divergence

Another f-divergence measure which should offer larger gradients than the previous f-divergence measures is the Exponential Divergence (ED) *Eguchi* [13]. As it hasn't been used as a loss function before we calculate derivatives to check but these results are also implied through definition of an f-divergence measure.

Definition VI.5. ED The Exponential Divergence for $X, Y \in \mathbb{R}^{B\text{x}T}$ of B batches and T points

$$ED(X,Y) = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}(x_t^{(i)})(\log(x_t^{(i)}) - \log(y_t^{(i)}))^2$$

$$\frac{\partial ED}{\partial x_t} = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}\log\left(\frac{x_t^{(i)}}{y_t^{(i)}}\right)\left(\log\left(\frac{x_t^{(i)}}{y_t^{(i)}}\right)+2\right)$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T}y_t^{(i)} = \sum_{i=1}^{B}\sum_{t=1}^{T}x_t^{(i)} \implies \frac{\partial ED}{\partial x_t} = 0$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T}y_t^{(i)} > \sum_{i=1}^{B}\sum_{t=1}^{T}x_t^{(i)} \implies \frac{\partial ED}{\partial x_t} < 0$$

$$\sum_{i=1}^{B}\sum_{t=1}^{T}y_t^{(i)} < \sum_{i=1}^{B}\sum_{t=1}^{T}x_t^{(i)} \implies \frac{\partial ED}{\partial x_t} > 0$$

Looking at the 2nd derivative we get that the function is convex.

$$\frac{\partial ED^2}{\partial x_t^2} = \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}\frac{2\left(\log\left(\frac{x_t^{(i)}}{y_t^{(i)}}\right)+1\right)}{x_t^{(i)}} \geq 0$$

Without the enforcing x and y from a proper probability distribution we get a non-convex function so could have our model converge to the wrong minima. We see this in the plot of the loss function in figure 9. It is important to note that this function is not symmetric like the KLD meaning $ED(X,Y) \neq ED(Y,X)$ for most X, Y because of this like KLD we would not want to give this to the optimiser to update the weights.

VI.F Converting A Divergence To A Distance Measure

We can however construct a symmetric distance measure from the Exponential Divergence and Kullback-Lieber Divergence to make its value more interpretable and also as a side effect eliminate the other minima. One approach is Intrinsic Discrepancy Loss applied to the KL divergence this offers reduced gradients over the normal KL divergence but is more robust than KLD and out performed the MSE *Bernardo et al.* [6]. For discrete distributions P and Q.

$$\delta(P,Q) = \min\left\{\sum_{i=1}^{n}P(x_i)\log\left(\frac{P(x_i)}{Q(x_i)}\right), \sum_{i=1}^{n}Q(x_i)\log\left(\frac{Q(x_i)}{P(x_i)}\right)\right\}$$

This makes the function symmetric however it does allow for incorrect minima if P,Q are not true distributions which we may not always want to force. We suggest a modification of this function in the form of taking the maximum instead of the minimum to avoid this issue and extend this to include the exponential divergence as well as the KL and all f-divergence measures $D(\cdot,\cdot)$ in the Maximum Divergence Distance Loss $\rho(P,Q)$.

$$\rho(P,Q) = \max\{D(P,Q), D(Q,P)\}$$

The proof of symmetry comes from the symmetry of max

$$\rho(P,Q) = \max\{D(P,Q), D(Q,P)\} = \max\{D(Q,P), D(P,Q)\} = \rho(Q,P)$$

It is zero when $P = Q$ using that for all divergence measure by definition $P = Q \implies D(P,Q) = 0$.

$$\rho(P,Q) = \max\{D(P,P), D(P,P)\} = \max\{0,0\} = 0$$

To satisfy the requirements of a loss function we note that the divergence measures are convex and the maximum of two convex functions is also convex so $\rho(X,Y)$ is convex. Proof that $f(x) = \max\{f_1(x),\ldots,f_m(x)\}$ the maximum of convex functions $f_1,\ldots,f_m$ with $dom(f) = dom(f_1)\cap\cdots\cap dom(f_m)$ is also convex. for any

$x, y \in dom(f), \lambda \in [0,1]$ for some $j \in \{1, \ldots, m\}$

$$\begin{aligned} f(\lambda x + (1-\lambda)y) &= f_j(\lambda x + (1-\lambda)y) \\ &\leq \lambda f_j(x) + (1-\lambda) f_j(y) \\ &\leq \lambda\{f_1(x), \ldots, f_m(x)\} + (1-\lambda)\{f_1(y), \ldots, f_m(y)\} \\ &= \lambda f(x) + (1-\lambda) f(y) \end{aligned}$$

Plots of these two loss functions are in figure 9 and offer much steeper gradients than the other loss functions.

VI.G Maximum L1

We use as a loss function for evaluating performance the maximum L1 Norm across each batch.

Definition VI.6. L1 MAX The Maximum L1 Norm for $X, Y \in \mathbb{R}^{B \text{x} T}$ of B batches and T points

$$L1M(X,Y) = \frac{1}{B}\sum_{i=1}^{B} \max_{t \in \{1,\ldots,T\}} |y_t^{(i)} - x_t^{(i)}|$$

This allows us to see the maximum absolute difference between the 2 probability measures and thus giving us an interpretation of the largest deviation we would expect from the true distribution. We would expect this to help us capture when the model produces predictions with lower modality than that of the true distribution.

VII Experiments: Gaussian Mixture

As a simplistic and easy to generate example that we use to test hyper-parameters of the models. We use a mixture of three Gaussians. To be able to form bins we evaluate the CDF across the bin.

$$Y(x) = k_1\Phi(x, \mu_1, \sigma_1) + k_2\Phi(x, \mu_2, \sigma_2) + k_3\Phi(x, \mu_3, \sigma_3)$$

Where,

$$\Phi(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{x-\frac{\delta}{2}}^{x+\frac{\delta}{2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) dx$$

with $\sum_{i=1}^{3} k_i = 1$ and δ representing the difference between each consecutive x. We sampled independently from uniform distribution $\mu \sim U(-3, 3)$, $\sigma \sim U(0.1, 5)$ and we sampled from the Dirichlet distribution their weighting $k \sim Dir(1, 1, 1)$. We used as our bins 100 bins of width 0.3 from -15 to 15. We then normalise the bins to sum to 1 and generate 10^5 random samples. As is common practice we normalise the inputs to the models μ, σ, to have 0 mean and variance 1.

We expect that the Mixture Model with a mixture of Gaussians will be able to perfectly predict the parameters as we use as the input the mean, variance and weighting of each Gaussian used to make the distribution the output. We use this example as a way to easily find the best hyperparameters for each of the models and check the can perform well on a simple experiment.

VII.A LSTM Model

We first explore the LSTM model and the effect of varying batch size on its performance with a unidirectional model, ELU activation for the initial dense layers, Sigmoid activation for the output of the LSTM array and Adam as optimiser with the Squared Hellinger Distance. The LSTM cell was the standard type with a encoder size of 512. We use a random 80/20 split between training and validation data.

We see in figure 10 plots for the KLD, L1 Norm and L1 Max and a table of their minimums of the validation dataset in figure 34. In general a reduction in L1 Loss as we reduce batch size. However, L1 Max appears to be unaffected suggesting possibly that the model struggles to identify multiple modes in all cases. The initial increase and negative KLD do suggest it is not training to an appropriate distribution. We see small batches leading to a slightly lower loss this is likely because the model over generalises in the case of larger batches. This

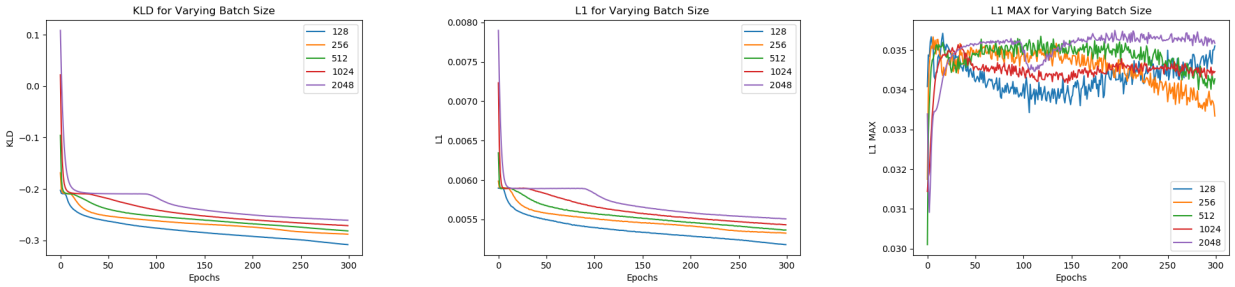

Figure 10: L1, L1 Max and KLD on the validation dataset for different batch sizes on the LSTM Model

is likely why in the source code provided in *Wang et al.* [36] they used a small batch size of 64. Looking in the table we see similar performance across the batches with a batch size of 512 achieving the lowest MSE. We therefore elect to use a batch size of 512 as the improvement for smaller batches seems small and the increase in train time is huge when we reduce the batch size.

We next explore the use of bidirectionality to increase the models predictive ability and perhaps identify modes of the distribution better in figure 11 we present their loss over epochs and in figure 35 their minimum loss across the validation data. We see that bidirectionality does not lead to a lower loss like we would expect although the difference is not significant and the loss over the training time is more consistent with the bidirectional model as the L1 Max and KLD becomes more negative as the unidirectional model trains.

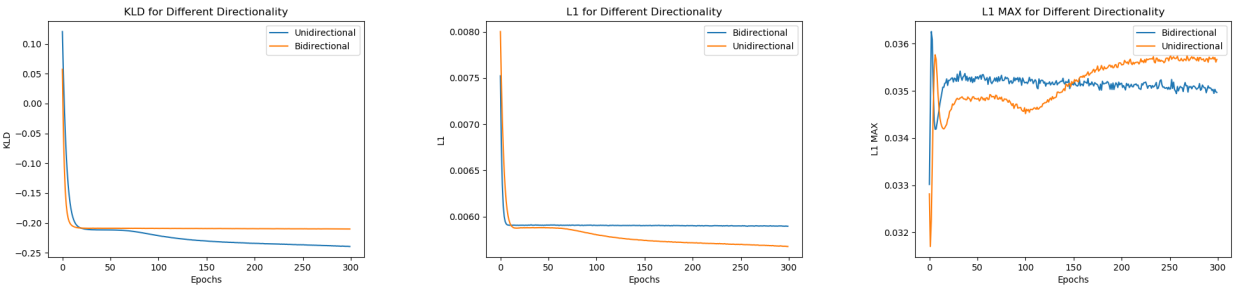

Figure 11: L1, L1 Max and KLD on the validation dataset for Unidirectional and Bidirectional LSTM models

We next test various combinations of loss function and optimiser to see which achieves the lowest loss over a given number of epochs with a unidirectional model. No results are shown for stochastic gradient descent as they all failed to train diverging to NaN in all cases. The validation loss using the Adam optimiser for various loss functions are shown in figure 12 and a table of their minimums in figure 36. They all converged to negative KLD values except in the case of the Maximum Exponential Divergenge Distance Measure this is likely as this loss function is more strict in enforcing the requirement of the output being a proper distribution. Contrary to this performed worst in terms of the L1 and Maximum L1 loss. The MSE performs best in terms of the L1 Max but the L1 followed closely by the Squared Hellinger Distance perform best in terms of the L1 loss. The values of the L1 Max are high enough to suggest that the model is failing to capture some of the modes of the distribution at about 3%. We opt to use the Squared Hellinger Distance from here on but could have argued to use the L1 loss but the L1 loss appeared to be stationary where as H^2 is decreasing. We would suspect this is due to L1 not being twice differentiable so is bouncing around its optimal weights.

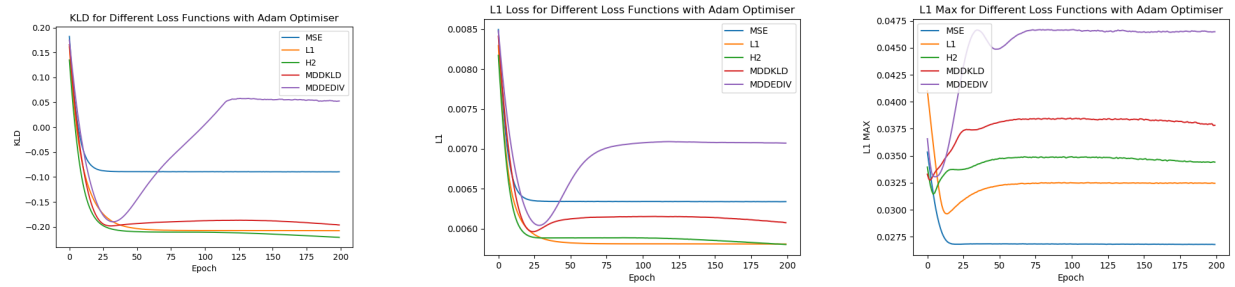

Figure 12: L1, L1 Max and KLD on the validation dataset for different loss functions with the Adam optimiser

Next we test the various choices of activation function for the initial dense layers and the activation function of the output of the LSTM array on a unidirectional LSTM model. The loss across the validation dataset is shown in figure 13 and the minimums it achieved in figure 39. We see a negative KLD again, similar L1 Max for all combinations but in terms of the L1 Loss the ELU activation function for the dense layers and Softplus for the output gives a better L1 loss with the others performing similarly. We elect to use the ELU activation and Softplus activation functions.

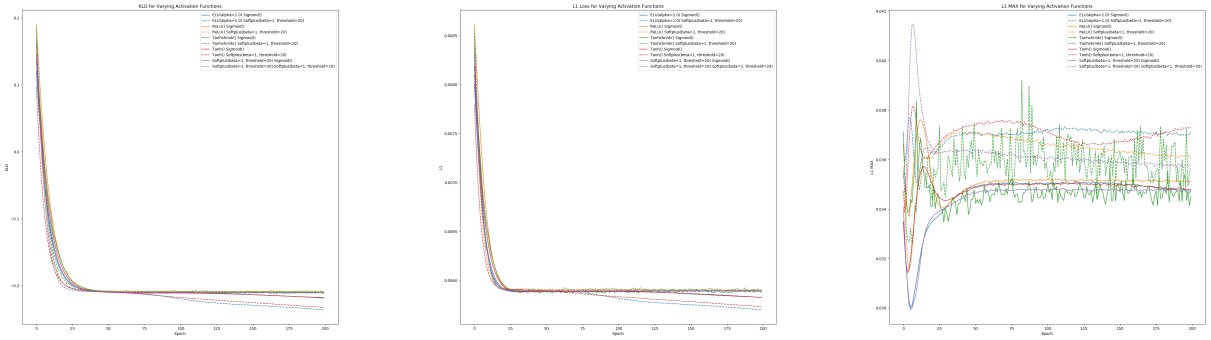

Figure 13: L1, L1 Max and KLD on the validation dataset for different activation functions. The colour and the first name in each lines label represents its initial dense layers activation function and the line type and the second name represents the activation for the output of the LSTM array

We move on to test varying LSTM types either standard or with peephole connection and their sizes. The validation loss is show in figure 14 with a table of minimums in figure 38. The 1024 encoder size standard LSTM would diverge which is why its results are not present. We see similar performance for both types of cell and the tested LSTM encoder sizes with the Peephole LSTM performing slightly better in terms of L1 Loss but worse in terms of L1 Max. The peephole cells also had a more negative KLD indicating convergence to an improper distribution. Although the difference is not very large so we opt to continue using the standard LSTM cell with an encoder size of 512.

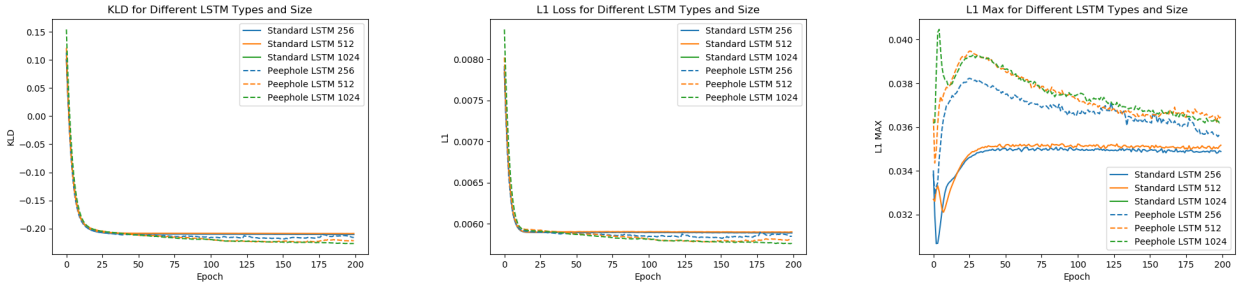

Figure 14: L1, L1 Max and KLD on the validation dataset for different LSTM Types. The colour and the number in each lines label represents its size and the line type and the name represents the type of LSTM array

To see the best performance we can expect we train for 1000 epochs our best performing LSTM model with the Squared Hellinger Distance and Bidirectionality the performance on the validation dataset is shown in figure 15 and the minimas therein achieved in figure 39. We see the model quickly reaches a set of weights such that the KLD remains negative and the L1 remain consistent. We see however the L1 Max decreases slowly as we train showing that if we trained for a longer time we could see some improvement in the predictions. The difference is not massive but the lowest value is still lower than any previously trained model.

To try and improve our performance we next train a ensemble of 5 Bidirectional LSTM models each for 1000 epochs and form an ensemble by training a dense layer to take the outputs of these 5 models and form a new prediction. The performance on the validation dataset is shown in figure 16 and it minimum loss in figure 40. We see a small increase in performance over the individual models although not massive in terms of the L1 Max but not the other 2 metrics. Which suggests that the models have similar predictions for most inputs so we don't gain much extra information from 5 separate models.

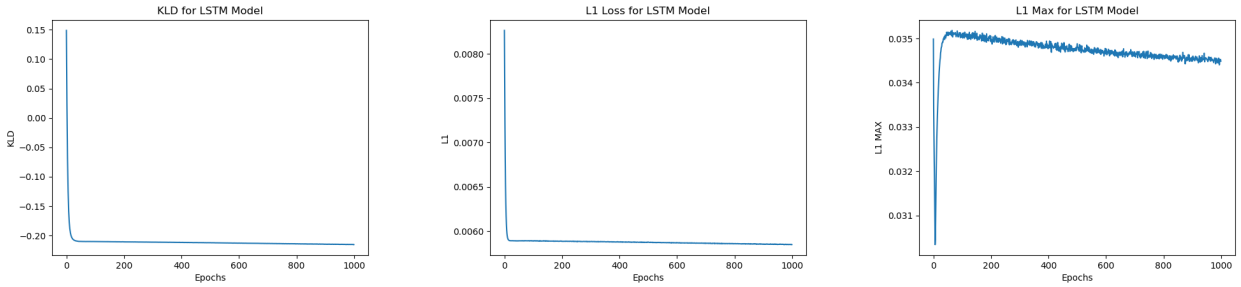

Figure 15: L1, L1 Max and KLD on the validation dataset for the LSTM Model for 1000 epochs

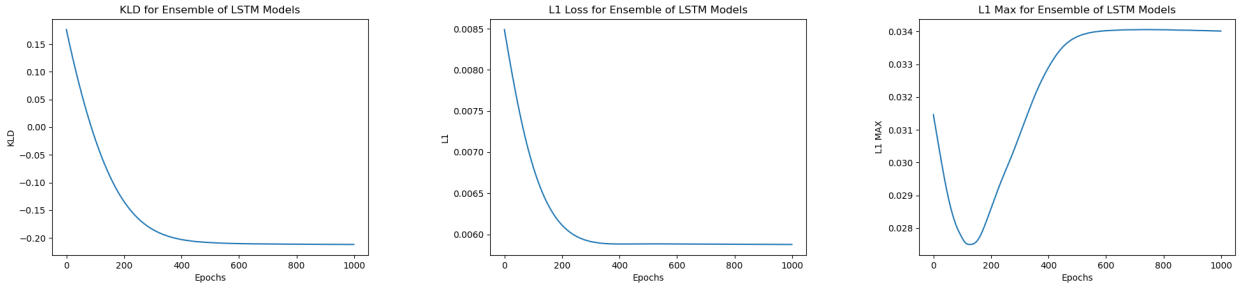

Figure 16: L1, L1 Max and KLD on the validation dataset for an ensemble of 5 LSTM Models

VII.B Mixture Model

We only use the Gaussian Mixture for our distribution for this question as it is the only applicable distribution. We use the Adam optimiser with H^2 as our loss function. As our activation functions we use the identity function for mu as we want to map to any value in $\mathbb{R}$ and for σ we use the exponential function as we need a positive real number but we empose a minimum of 10^{-12} to prevent division by zero errors. Similarly to the LSTM Model we use ELU as the activation function for the initial dense layers. We first explore the optimal number of distributions in our mixture by performing Kfold cross validation with $K = 5$ over 500 epochs for $1, 2, 3, 4, 5, 6, 8, 16, 32, 64$ Gaussians in our mixture. We expect that we should find that the optimal number is 3 or close to 3.

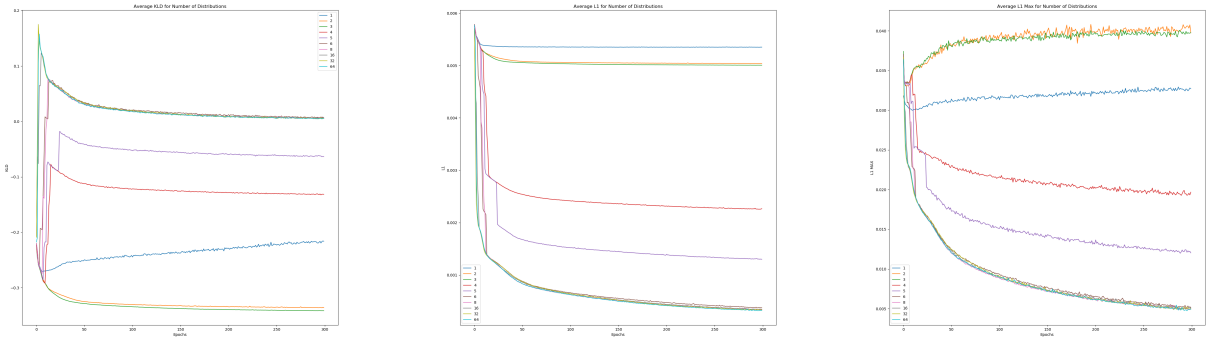

Figure 17: L1, L1 Max and KLD on the validation datasets averaged across 5 folds for different numbers of distributions in the mixture

Plots of the loss for the number of distributions are shown in figure 17 and the minimum loss in figure 41. We see that the optimal value is 6 higher than we might first expect however, this is likely down to the re-scaling of the data as the estimates of the mixture model aren't normalised. This is shown by the negative KLD for the lower numbers of distributions which is negative when the predicted values do not sum to 1. This requirement is not enforced across the bins with this model. We could try to normalise the output of the mixture across the bins however this is likely to induce errors in backpropagation caused by NaN gradients. A potential solution is to use a SoftMax activation for the values given by the mixture but this would change the distribution we predict to no longer be Gaussian. The largest drop occurs at 6 distributions but we see for 16 and 32 the divergence is a

considerable amount lower. The large jumps in minimum loss going from 1 to 2 to 3 to 4 distributions indicate the model is learning to predict the additional modes. From here on we use 6 Gaussians.

We next want to explore the effect of batch size on the models performance we do this simply by training models for different batch sizes. The validation loss for these are shown in figure 18 and their minimas in figure 42. Ideally we would also use kfold cross validation here but for small batch sizes it can take prohibitively long to train the model. We see the model is largely invariant to the size of the batch with all converging but lower losses are observed with smaller batches. Either a batch of 1024 or 512 samples appears to be optimal so we select 1024 from here on.

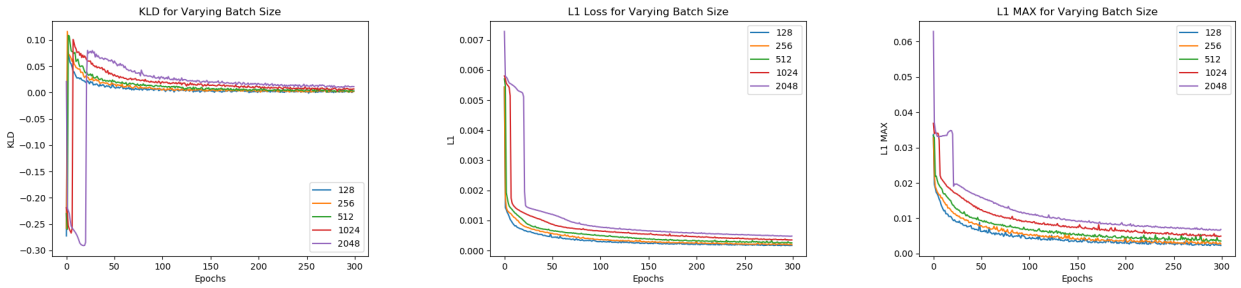

Figure 18: L1, L1 Max and KLD on the validation datasets for different batch sizes with the Mixture Model

We now explore the effects of two optimisers Stocahstic Gradient Descent and Adam with different loss functions. We look at the results on the validation dataset and are shown in 19 and their minimums in figure 43. Similar to the LSTM model in that the Adam optimiser significantly outperforms SGD with all loss functions but SGD does not diverge to NaN like it did with the LSTM Model. The Maximum Exponential Divergence Distance performs the best with all loss functions. Although looking at how it reaches this we see a massive jump at 200 epochs would suggest perhaps this loss in unstable and might not always work best. It could be best used when we have a pretrained model we want to train further to squeeze out more performance. The squared Hellinger Distance has the closest performance closely followed be L2 and is more consistent so from here on we use it with the Adam optimiser for this model.

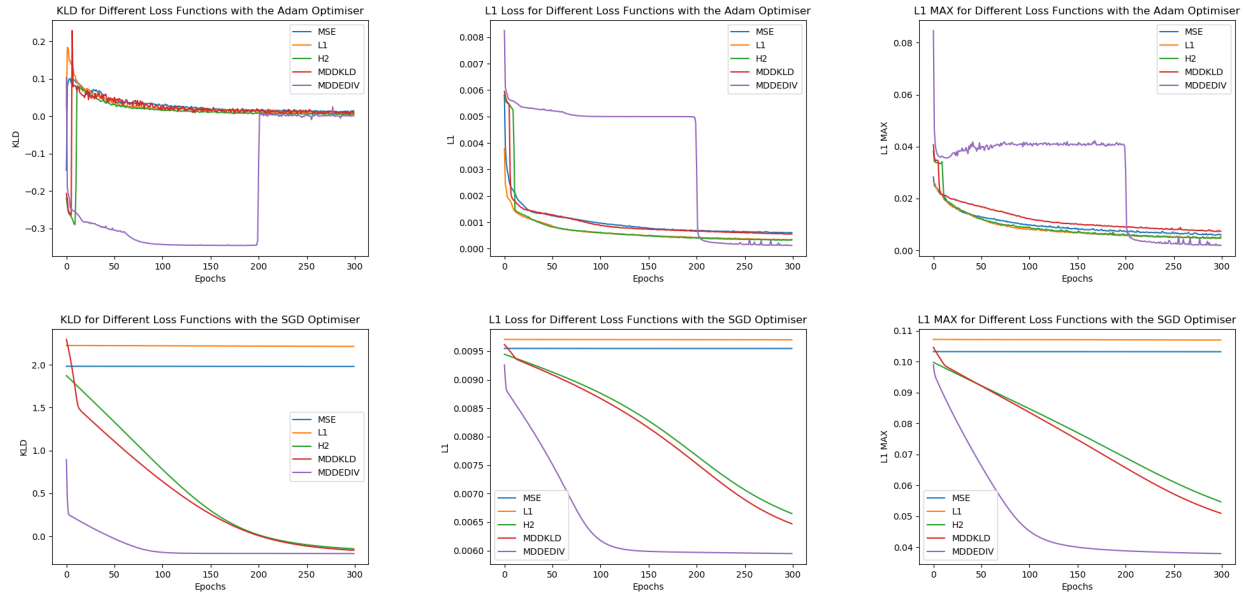

Figure 19: L1, L1 Max and KLD on the validation datasets for different optimisers. Adam was used to train the first row and SGD the second row with each line representing a different loss function as indicated in the legend.

We next want to explore different activation function for the dense layers and the output for σ. The activation function for the weights is fixed as we require it to sum to unity and for μ we use the identity function as we know it can take any value in $\mathbb{R}$. The results on the validation dataset are in figure 20 the line colour represents dense activation and the line style the activation of the σ output layer. The minimum loss achieved across the validation data set is shown in figure 44. The Clamped function represents $ReLU(x) + \epsilon$ where $\epsilon = 10^{-12}$. We see a general trend of the best performing activation for the σ layer being the exponential function and 2 combinations failed to converge Tanhshrink with Exponential and Softplus. Overall the best performing combination in terms of

the KLD,L1 Loss and L1 Max is the hyperbolic tangent function as the activation for the dense layers and the Exponential function on the layer the outputs σ. This is why we elect to use this combination from here on.

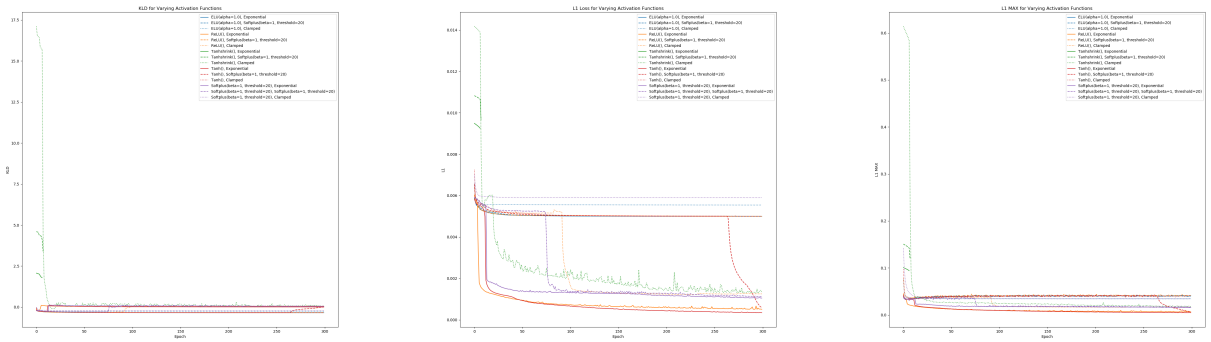

Figure 20: L1 and KLD on the validation datasets for different batch sizes

We now train for a large number of epochs with first the Squared Hellinger Distance as loss function then with the Maximum Exponential Divergence to explore the performance after the jump compared to the next best model. The performance on the validation data set for H^2 is in figure 45 and its minimum values in figure 45 and for the Maximum Exponential Divergence in figure 22 and its minimum values in figure 22. We see the problem we observed before with the Maximum Exponential Divergence with a large jump in loss now this time at 2000 epochs showing despite offering a low loss it is inconsistent perhaps due to the steepness of the loss function causing bouncing around the optimal value. The model trained with the Squared Hellinger Distance achieves very low loss but the Maximum Exponential Divergence still outperforms it with half the L1 and L1 Max. We do choose to continue using H^2 as the large jumps seem to happen randomly and this is not an ideal characteristic.

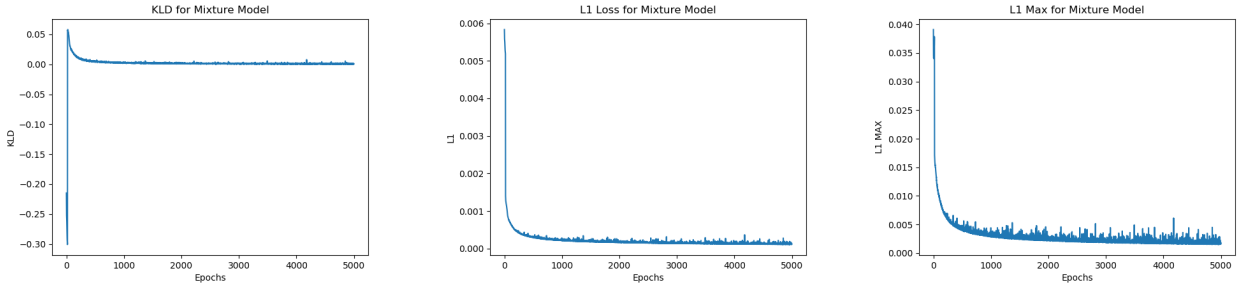

Figure 21: L1, L1 Max and KLD on the validation dataset trained with the Squared Hellinger Distance for 5000 epochs

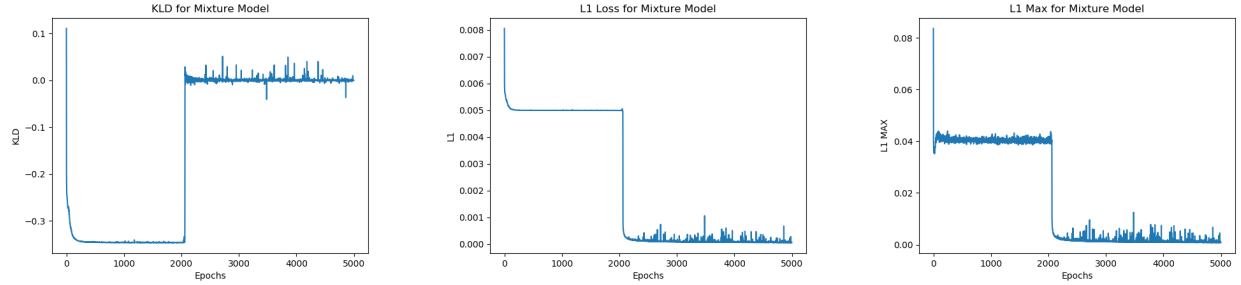

Figure 22: L1, L1 Max and KLD on the validation dataset trained with the Maximum Exponential Divergence for 5000 epochs

Finally we utalise an ensemble method we train 5 Mixture Models on a random sample of the training dataset for 1000 epochs. We then use the predictions of the models that achieved the lowest validation loss on the data to set to produce a Gaussian Mixture from the Mixtures predicted by the 5 Models. The performance on the common validation dataset is shown in figure 23 and the minimum they achieved in figure 47. We achieve slightly worse performance to the model we trained for a large number of epochs but better than models trained in previous tests. We suspect this is due to the submodels requiring longer to train but the difference in loss between the model and the performance of other previous models is small suggesting not much is gained from using an ensemble as the predictions are already very accurate.

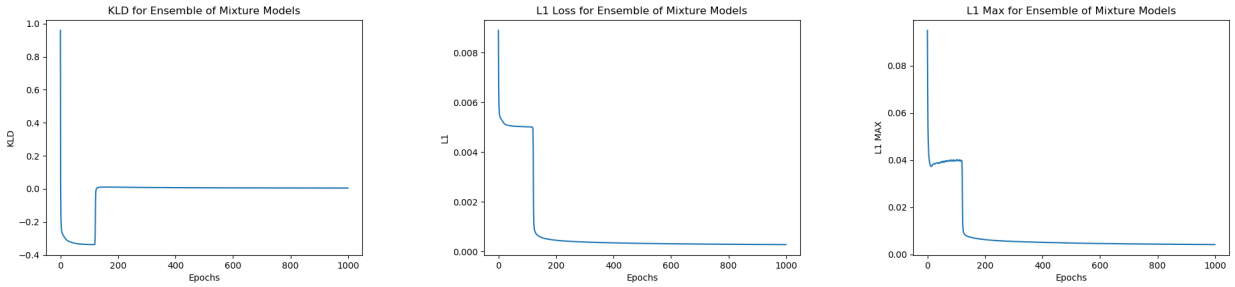

Figure 23: L1, L1 Max and KLD on the validation datasets for Ensemble of 5 Gaussian Mixture Models which were each trained for 1000 epochs on a random sample of the training dataset

VII.C Comparison

The Mixture Model achieved significantly lower loss than that of the LSTM array and due to it not having the LSTM arrays, its complexity is not scaled by the number of bins T and as it is able to use a larger batch size it trains much faster than the LSTM Model. The mixture with this data has very little it needs to learn from its inputs as it is predicting a Gaussian Mixture from the parameters of a Gaussian mixture so its performance on other real world problem could be worse and very dependent on the matching of an appropriate distribution. The LSTM Model does not have this problem and does not need this adjustment but for large numbers of bins the training quickly becomes very long and requires a significant amount of memory to be able to perform back propagation on the 1 or 2 LSTM arrays.

VIII Experiments: Protein Distribution

As previously mentioned a real world example is the production of proteins from a self-regulating gene which is shown in figure 24. We fix the average number of proteins to allow us to capture the distribution with a fixed number of bins of a fixed size with the average number of proteins given by

$$P = \frac{v_1 v_0 k_0}{d_1 d_0 (k_0 + k_1)}$$

We also fix $d_0 = 1$ as we can scale the other rate parameters to have the same effect as if this was a free parameter. We sample k_0, k_1 from the log uniform distribution $LOG_{10}U(0.01, 1)$, d_1, v_0 from the uniform distribution $U(0.01, 1)$ and finally we choose v_1 such that the average number of proteins is 100. Using Gillespie's Algorithm we generate the timeseries of proteins for given reaction rates then take independent samples to get the distribution of proteins. To get independent samples for the rate parameters r we sample every $max(max(1/r), 1)$. The binning procedure is we take a number T of equal size bins on the interval $[0, 300)$ then normalise so they sum to 1. The standard binning functions given in numpy and scipy don't allow for fast vectorised binning so we make some improved binning functions that allow for vectorisation. Even in this simplistic model the binning and generation of the timeseries of proteins is rather long at 45.638 seconds on average per set of parameters. Even using parallelisation across nodes the generation and binning for the 10^5 samples requires a lot of time and access to significant computing resources motivating the need for a faster generation method.

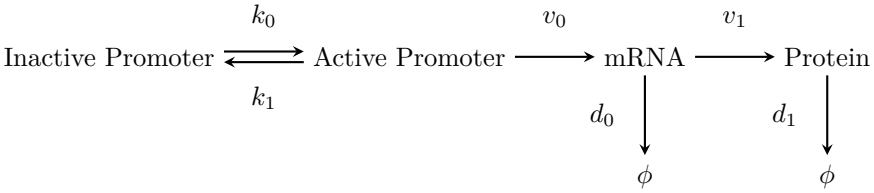

Figure 24: Gene switching model

VIII.A LSTM Model

We first explore the performance of the Bidirectional LSTM model trained with Adam and H^2 on this problem for 1000 epochs. Its performance on the validation dataset is shown in figure 25 and its minimum performance

in figure 48. The KLD here is not negative as it was on the Gaussian data but is high and L1 loss is low and slowly decreasing but the L1 Max is rather large suggesting the model is not predicting significant peaks in the distribution of proteins.

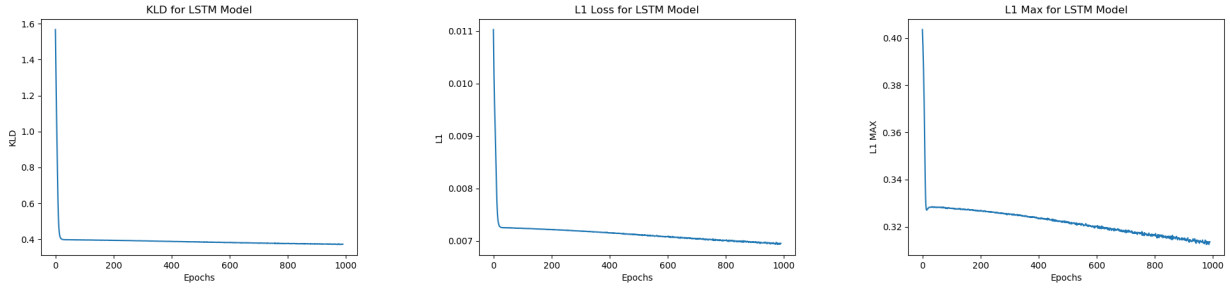

Figure 25: L1,L1 MAX and KLD on the validation dataset for the LSTM model using the distribution of Proteins in a self-regulating gene model

We next see if an ensemble of 5 LSTM models trained for 500 epochs as used before can increase the performance with this data. The performance on the validation dataset is shown in figure 26 and its minimum loss in figure 49. The ensemble has much lower L1 and L1 Max but these values are still much higher than we would like.

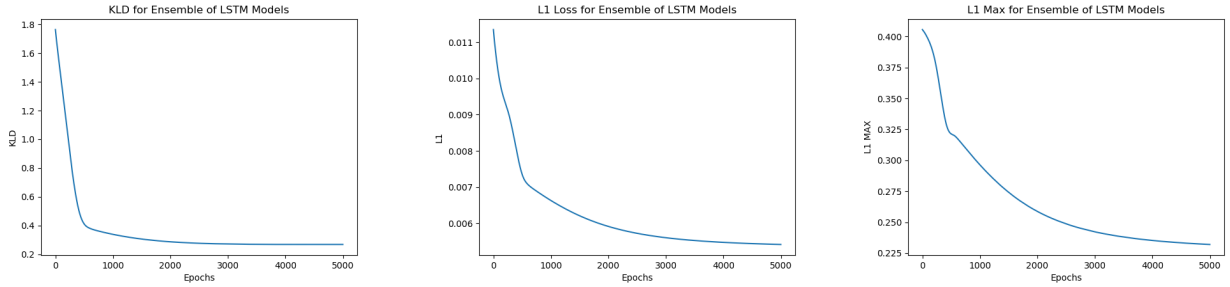

Figure 26: L1, L1 Max and KLD on the validation dataset for an ensemble of 5 LSTM Models

VIII.B Mixture Model

We first explore the possible distributions Gaussian, Gamma-Poisson, Poisson Log-Normal and the number of them first. We used the Adam optimiser and H^2 for activation functions the Gamma-Poisson distribution we used the clamped exponential for both as $\alpha, \beta > 0$ and for the Poisson Log-Normal we used the same activations as with the Gaussian distribution. The Log-Normal distribution we do not display results for as it produced NaNs. This was due to precision errors caused by evaluating the Poisson Distribution at the quadrature points. I tried implementing methods to prevent the NaNs being output such as assigning them as zero but this worked better but still resulted in NaNs. Perhaps a more strict activation function that prevents it evaluating models with very large μ or σ at the risk of hindering the range of distributions we can match. We could use double precision in the evaluation of the CDF but this will result in a large increase in computation time, or casting to a new set of bins that will lead to smaller appropriate parameters. Here this could be done with 100 bins in $[0, 100)$ instead of $[0, 300)$. Lastly perhaps another quadrature method that avoids using Pytorch's Poisson distribution. The results for 5 folds on the validation datasets for a mixture of Gaussians are show in figure 27 with its minimums in figure 27 and for a mixture of Gamma-Poissons in 28 with its minimums in figure 28.

We see that the Gamma-Poisson Model out performs the Gaussian Model on the regulating gene model by a significant margin for the KLD, L1 and L1 Max more so with the KLD and L1 Max. This is what we expected with the support being more similar to the distribution we are trying to emulate. Both mixtures perform significantly better when the number of distributions is greater than 1. The Gaussian Model performed best with 6 distributions with more offering only slightly greater performance. The Gamma-Poisson Model has decreasing loss as we increase the number of distributions however beyond 3/4 distributions the reduction in loss is small. We therefore select for this problem the Gamma-Poisson distribution with 4 distributions in the mixture.

We then train a Mixture Model of 4 Gamma-Poisson distribution for 3000 epochs with Adam and H^2 to see its performance after a large number of epochs. The performance of this model on validation data is shown in figure

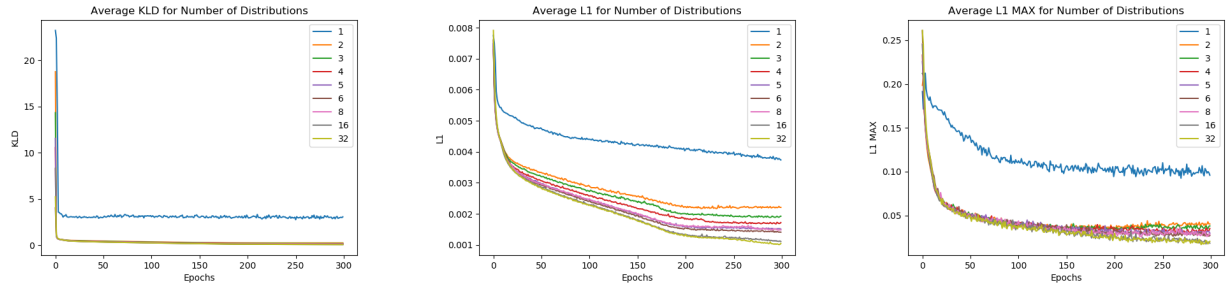

Figure 27: L1,L1 MAX and KLD on the validation datasets averaged across 5 folds for 300 epochs on different numbers of Gaussian distributions on the distribution of Proteins in a self-regulating gene model

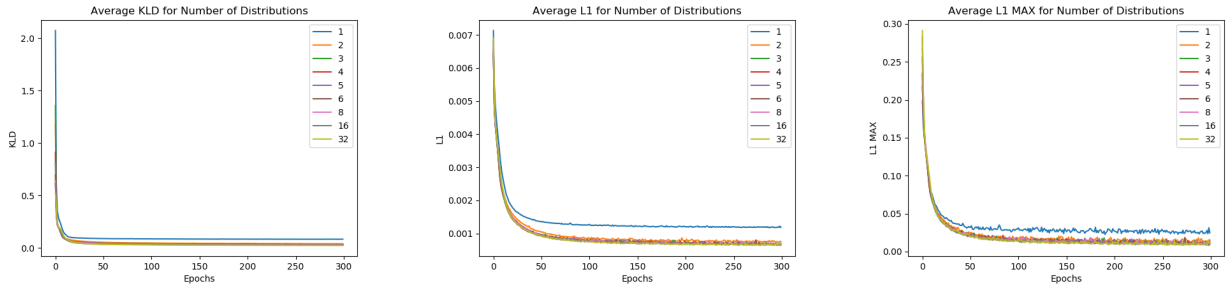

Figure 28: L1,L1 MAX and KLD on the validation datasets averaged across 5 folds for 300 epochs on different numbers of Gamma-Poisson distributions on the distribution of Proteins in a self-regulating gene model

29 and its minimum loss in figure 52. We see its performance after relatively few epochs is rather good but not as good as it was on the Gaussian data as we would expect and we see a reduction in loss coming from the results on the 5 folds.

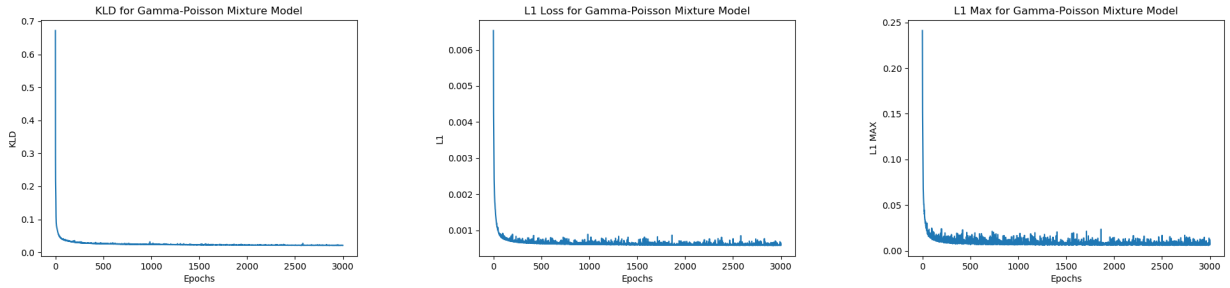

Figure 29: L1,L1 MAX and KLD on the validation dataset for a Mixture Model of 4 Gamma-Poisson distributions on the distribution of Proteins in a self-regulating gene model

As before we create an ensemble of 5 of the previous models trained for 1000 epochs the results on the common validation dataset the results of which are shown in figure 30. We achieve a slight decrease in performance over the individual model.

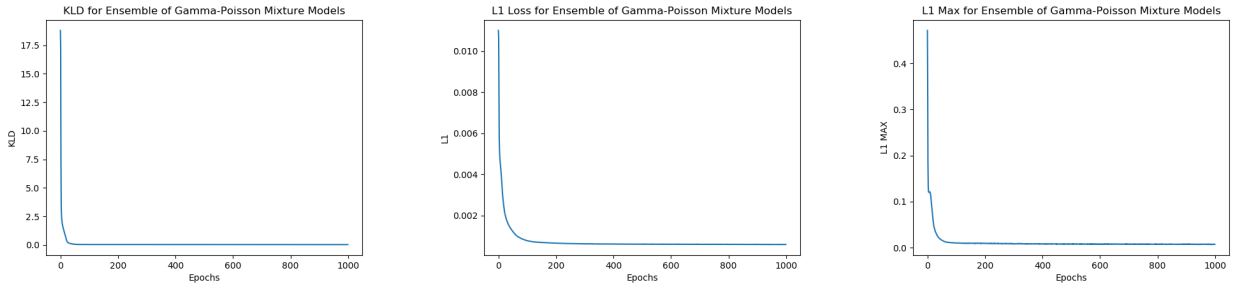

Figure 30: L1,L1 MAX and KLD on the validation datasets for an Ensemble of 5 Mixture Models consisting of 4 Gamma-Poisson distributions on the distribution of Proteins in a self-regulating gene model

VIII.C Comparison

The Mixture Model with the Gamma-Poisson distribution significantly outperforms the LSTM model in all metrics and trains significantly faster with less memory usuage. We do however with the mixture lose some generality as we need to find an appropriate distribution but the Gaussian mixture trained across 5 folds still had lower validation loss than the LSTM Model.

IX Experiments: mRNA and Protein Joint Distribution

To show the increased versatility of the Mixture Model we use it to predict the bivariate joint distribution of proteins and mRNA. The creation of the data is almost the same as the univariate case of the protein distribution except we also store the number of mRNA at each sampled point and fix v_0 such that the average number of mRNAs is 50. The formula for the average number of mRNA is shown below.

$$M = \frac{v_0 k_0}{d_0(k_0 + k_1)}$$

We only show this with the Mixture Model as to use the LSTM model we would need a model with a prohibitively large number of LSTM arrays for the bivariate case.

IX.A Mixture Model

We first evaluate the performance across 5 folds as we increase the number of Dirichlet-Multinomial Gamma-Poisson distributions in our mixture. Using hyperbolic tangent activation for the dense layers and the Exponential as the activation for the parameters. The results on the validation dataset averaged over the kfolds are shown in figure 31 and the minimums in figure 54. The training of this model takes much longer than before due to the extra dimension which is why we only checked using $1, 2, 4$ distributions. The KLD with 1 distribution is high and the L1 and L1 Max are also much higher than that of 2 or 4 distributions suggesting it fails to capture additional peaks. We see similar performance for 2 and 4 distributions with a slight increase when moving to 4 from 2. We opt to use 4 distributions from here on.

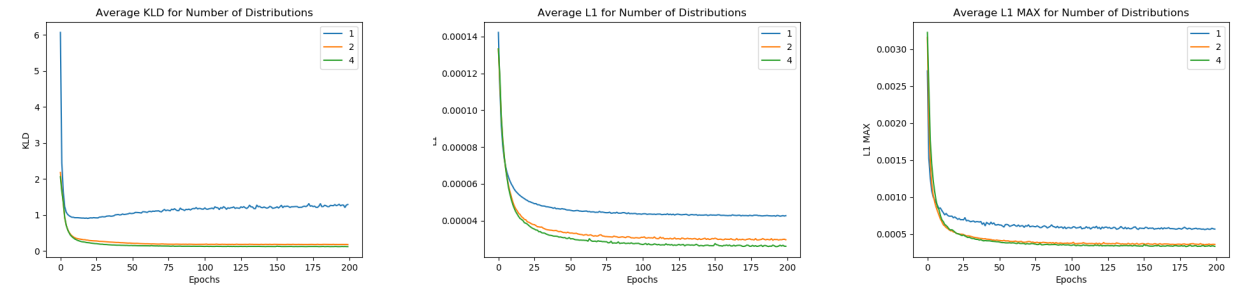

Figure 31: L1,L1 MAX and KLD on the validation dataset averaged across 5 folds for different numbers of Dirichlet-Multinomial Gamma-Poisson distributions

To examine the performance further we train the Mixture Model with 4 distributions for 800 epochs. The performance on the validation dataset is shown in figure 32 and the minimum loss achieved in figure 55. We see that the models performance doesn't improve much after the first 200 epochs but we see the loss is lower than it was averaged over the 5 folds.

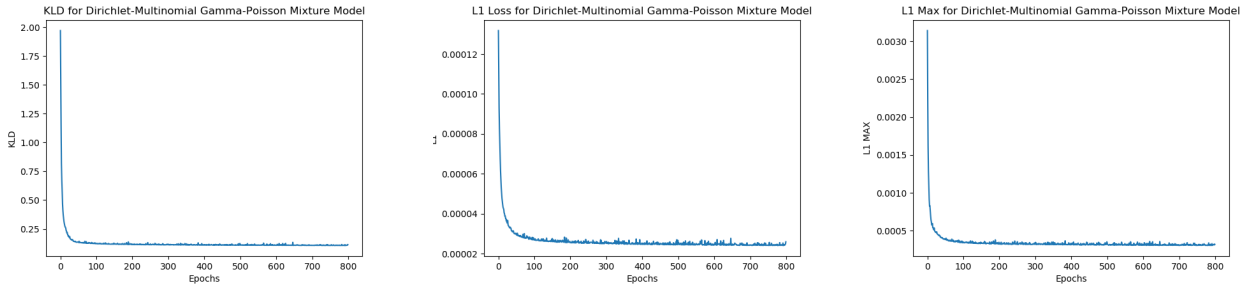

Figure 32: L1,L1 MAX and KLD on the validation dataset for a Mixture Model using 4 Dirichlet-Multinomial Gamma-Poisson distributions trained for 800 epochs

As previously we construct an ensemble using 5 of these models trained for 500 epochs. The loss on the validation dataset are in figure 33 and the minimum loss in figure 56. We see minimum loss very close to that of the individual model and thus we haven't gained a significant increase in emulation accuracy from using an ensemble.

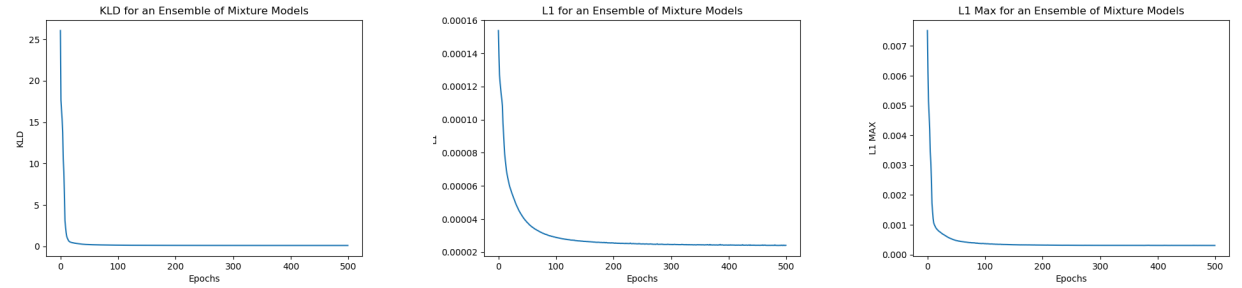

Figure 33: L1,L1 MAX and KLD on the validation dataset for a ensemble of 5 Mixture Models using 4 Dirichlet-Multinomial Gamma-Poisson distributions trained for 500 epochs

X Concluding Remarks

Using neural networks to produce emulating mixture distributions provides a fast and effective method that can be scaled easily to higher dimension problems. An exploration of its performance on smaller datasets and on more complex networks would provide an insight into whether it can replace Approximate Bayesian Computation (ABC) in larger stochastic reaction networks. We suggest in such problems using a deeper neural network and exploring other distributions which could further increase its emulating ability with much more complex problems such as the growth of cancerous tumours. In higher dimensions the Poisson Log-Normal distribution, provided an effective gradient preserving numerical approximation technique is used, could provide emulation of n-variate distributions. Stochastic reaction networks that exhibit a wider range of complex behaviours may increase the performance of ensembles of Mixture Models where they can emulate rare complex behaviours. Their lack of improvement in predictive ability is likely due to the individual models performance being good enough that they very rarely produce incorrect predictions.

The main advantage the LSTM based model has is in its increased generality as we aren't required to select a distribution to fit the problem. Although its performance here was significantly worse than that of the mixture model, both taking longer to train and having a significantly worse emulating ability. Improvements could be found increasing the depth of the network or in the application of clockwork RNN's *Koutník et al.* [20] that can learn more complex relationships in series but increase the number of hyperparameters that need to be tuned.

In the training of our models it was clear the the MSE offered poor performance and functions with larger gradients for values close to zero but the L1 Norm and the Squared Hellinger Distance offer an attractive alternative. The Squared Hellinger Distance also is twice differentiable which can be important when we want to achieve very accurate models. Our Exponential Divergence Distance Measure despite offering the lowest validation loss is too

unstable to train models except maybe in the case of further training of an already well-trained model. Exploring the performance of other f-divergence measures like those stated in *Eguchi* [13] modifided to become distance measures could provided other loss functions of interest.

References

[1] J. AITCHISON and C. H. HO. The multivariate Poisson-log normal distribution. *Biometrika*, 76(4):643–653, 12 1989. ISSN 0006-3444. doi: 10.1093/biomet/76.4.643. URL https://doi.org/10.1093/biomet/76.4.643.

[2] L. Amrhein, K. Harsha, and C. Fuchs. A mechanistic model for the negative binomial distribution of single-cell mrna counts. *bioRxiv*, 2019. doi: 10.1101/657619. URL https://www.biorxiv.org/content/early/2019/06/05/657619.

[3] G. E. Bates and J. Neyman. Contributions to the theory of accident proneness. 1. an optimistic model of the correlation between light and severe accidents. 1952.

[4] M. A. Beaumont. Estimation of population growth or decline in genetically monitored populations. *Genetics*, 164(3):1139–1160, 2003. ISSN 0016-6731. URL https://www.genetics.org/content/164/3/1139.

[5] M. A. Beaumont. Approximate bayesian computation in evolution and ecology. *Annual Review of Ecology, Evolution, and Systematics*, 41(1):379–406, 2010. doi: 10.1146/annurev-ecolsys-102209-144621. URL https://doi.org/10.1146/annurev-ecolsys-102209-144621.

[6] J. M. Bernardo, M. J. Bayarri, J. O. Berger, A. P. Dawid, D. Heckerman, A. F. M. Smith, M. W. (eds, J. M. Bernardo, and M. A. Juárez. Intrinsic estimation. In *Bayesian Statistics 7*, pages 456–476. Oxford University Press, 2003.

[7] C. Bishop. Mixture density networks. Technical Report NCRG/94/004, January 1994. URL https://www.microsoft.com/en-us/research/publication/mixture-density-networks/.

[8] L. Breiman. Bagging predictors. Technical report, Department of Statistics, University of California Berkeley, September 1994. URL https://www.stat.berkeley.edu/~breiman/bagging.pdf.

[9] M. G. Bulmer. On fitting the poisson lognormal distribution to species-abundance data. *Biometrics*, 30(1):101–110, 1974. ISSN 0006341X, 15410420. URL http://www.jstor.org/stable/2529621.

[10] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.07289.

[11] K. Cranmer, J. Brehmer, and G. Louppe. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences*, 2020. ISSN 0027-8424. doi: 10.1073/pnas.1912789117. URL https://www.pnas.org/content/early/2020/05/28/1912789117.

[12] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011. ISSN 1532-4435.

[13] S. Eguchi. A differential geometric approach to statistical inference on the basis of contrast functionals. *Hiroshima Mathematical Journal*, 15(2):341–391, 1985. doi: 10.32917/hmj/1206130775. URL http://projecteuclid.org/euclid.hmj/1206130775.

[14] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194 vol.3, 2000.

[15] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000. doi: 10.1162/089976600300015015. URL https://doi.org/10.1162/089976600300015015.

[16] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, Mar 9, 2000. doi: 10.1021/jp993732q. URL http://dx.doi.org/10.1021/jp993732q.

[17] U. Gonzales-Barron and F. Butler. A comparison between the discrete poisson-gamma and poisson-lognormal distributions to characterise microbial counts in foods. *Food Control*, 22(8):1279 – 1286, 2011. ISSN 0956-7135. doi: https://doi.org/10.1016/j.foodcont.2011.01.029. URL http://www.sciencedirect.com/science/article/pii/S0956713511000491.

[18] J. Guo, S. Lu, H. Cai, W. Zhang, Y. Yu, and J. Wang. Long text generation via adversarial training with leaked information, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16360/16061.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[20] J. Koutník, K. Greff, F. J. Gomez, and J. Schmidhuber. A clockwork RNN. *CoRR*, abs/1402.3511, 2014. URL http://arxiv.org/abs/1402.3511.

[21] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL https://doi.org/10.1007/3-540-49430-8_2.

[22] Q. Li, A. Cassese, M. Guindani, and M. Vannucci. Bayesian negative binomial mixture regression models for the analysis of sequence count and methylation data. *Biometrics*, 75(1):183–192, 2019. doi: 10.1111/biom.12962. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/biom.12962.

[23] F. Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5):183 – 197, 1991. ISSN 0925-2312. doi: https://doi.org/10.1016/0925-2312(91)90023-5. URL http://www.sciencedirect.com/science/article/pii/0925231291900235.

[24] J. F. Nelson. Multivariate gamma-poisson models. *Journal of the American Statistical Association*, 80(392):828–834, Dec 1, 1985. doi: 10.1080/01621459.1985.10478190. URL http://www.tandfonline.com/doi/abs/10.1080/01621459.1985.10478190.

[25] H. D. Nguyen and G. McLachlan. On approximations via convolution-defined mixture models. *Communications in Statistics - Theory and Methods*, 48(16):3945–3955, Aug 18, 2019. doi: 10.1080/03610926.2018.1487069. URL http://www.tandfonline.com/doi/abs/10.1080/03610926.2018.1487069.

[26] T. T. Nguyen, H. D. Nguyen, F. Chamroukhi, and G. J. McLachlan. Approximation by finite mixtures of continuous density functions that vanish at infinity. *Cogent Mathematics Statistics*, 7(1), Jan 1, 2020. doi: 10.1080/25742558.2020.1750861. URL http://www.tandfonline.com/doi/abs/10.1080/25742558.2020.1750861.

[27] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. *CoRR*, abs/1904.09237, 2019. URL http://arxiv.org/abs/1904.09237.

[28] C. P. Robert, J.-M. Cornuet, J.-M. Marin, and N. S. Pillai. Lack of confidence in approximate bayesian computation model choice. *Proceedings of the National Academy of Sciences*, 108(37):15112–15117, 2011. ISSN 0027-8424. doi: 10.1073/pnas.1102900108. URL https://www.pnas.org/content/108/37/15112.

[29] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[30] V. Shahrezaei and P. S. Swain. The stochastic nature of biochemical networks. *Current Opinion in Biotechnology*, 19(4):369–374, 2008. doi: 10.1016/j.copbio.2008.06.011. URL https://www.clinicalkey.es/playcontent/1-s2.0-S0958166908000815.

[31] S. Siami-Namini and A. S. Namin. Forecasting economics and financial time series: ARIMA vs. LSTM. *CoRR*, abs/1803.06386, 2018. URL http://arxiv.org/abs/1803.06386.

[32] S. A. Sisson, G. W. Peters, M. Briers, and Y. Fan. A note on target distribution ambiguity of likelihood-free samplers, 2010. URL http://arxiv.org/abs/1005.5201.

[33] T. Székely and K. Burrage. Stochastic simulation in systems biology. *Computational and Structural Biotechnology Journal*, 12(20):14 – 25, 2014. ISSN 2001-0370. doi: https://doi.org/10.1016/j.csbj.2014.10.003. URL http://www.sciencedirect.com/science/article/pii/S2001037014000403.

[34] R. T. Trevor Hastie and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, second edition, 2009. ISBN 9780387848570. doi: 10.1007/978-0-387-84858-7.

[35] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017. URL http://arxiv.org/abs/1708.04782.

[36] S. Wang, K. Fan, N. Luo, Y. Cao, F. Wu, C. Zhang, K. A. Heller, and L. You. Massive computational acceleration by using neural networks to emulate mechanism-based biological models. *Nature Communications*, 10(1):1–9, Dec 2019. doi: 10.1038/s41467-019-12342-y. URL https://search.proquest.com/docview/2297119101.

[37] R. D. Wilkinson. Accelerating abc methods using gaussian processes, 2014. URL http://arxiv.org/abs/1401.1436.

XI Data Tables

XI.A Gaussian Mixture

XI.A.1 LSTM Model

Batch Size	H2	MSE	DIV	L1 MAX	L1
128	0.227146	0.000109	-0.308390	0.033424	0.005180
256	0.255286	0.000091	-0.288083	0.031748	0.005326
512	0.264446	0.000080	-0.281693	0.030103	0.005364
1024	0.278667	0.000082	-0.271466	0.031440	0.005432
2048	0.293843	0.000083	-0.260971	0.030916	0.005507

Figure 34: Minimum Validation Loss for different batch sizes

Bidirectional	H2	MSE	DIV	L1 MAX	L1
False	0.324661	0.000085	-0.239323	0.031700	0.005676
True	0.367802	0.000087	-0.210061	0.033016	0.005895

Figure 35: Minimum Validation Loss for bidirectionality

Optimiser	Loss	H2	MSE	DIV	L1 MAX	L1
Adam	H2	0.351781	0.000082	-0.221339	0.031448	0.005801
Adam	L1	0.379517	0.000078	-0.207952	0.029633	0.005806
Adam	MDDEDIV	0.392923	0.000098	-0.190069	0.033060	0.006040
Adam	MDDKLD	0.371288	0.000092	-0.198279	0.032741	0.005962
Adam	MSE	0.612590	0.000074	-0.089754	0.026787	0.006337
SGD	H2	NaN	NaN	NaN	NaN	NaN
SGD	L1	NaN	NaN	NaN	NaN	NaN
SGD	MDDEDIV	NaN	NaN	NaN	NaN	NaN
SGD	MDDKLD	NaN	NaN	NaN	NaN	NaN
SGD	MSE	NaN	NaN	NaN	NaN	NaN

Figure 36: Minimum Validation Loss for different loss functions

Dense Activation	Output Activation	H2	MSE	DIV	L1 MAX	L1
ELU(alpha=1.0)	Sigmoid()	0.356720	0.000080	-0.217725	0.030024	0.005828
ELU(alpha=1.0)	Softplus(beta=1, threshold=20)	0.328169	0.000090	-0.236009	0.035070	0.005701
ReLU()	Sigmoid()	0.368639	0.000089	-0.209631	0.033706	0.005890
ReLU()	Softplus(beta=1, threshold=20)	0.369432	0.000085	-0.208994	0.031530	0.005902
Softplus(beta=1, threshold=20)	Sigmoid()	0.367826	0.000080	-0.210125	0.029928	0.005890
Softplus(beta=1, threshold=20)	Softplus(beta=1, threshold=20)	0.367821	0.000100	-0.210016	0.034692	0.005889
Tanh()	Sigmoid()	0.355825	0.000085	-0.218434	0.031420	0.005826
Tanh()	Softplus(beta=1, threshold=20)	0.333291	0.000090	-0.232506	0.033869	0.005730
Tanhshrink()	Sigmoid()	0.366828	0.000084	-0.211328	0.033465	0.005871
Tanhshrink()	Softplus(beta=1, threshold=20)	0.370049	0.000085	-0.209338	0.032591	0.005882

Figure 37: Minimum Validation Loss for different activation functions

LSTM Type	Encoder Size	H2	MSE	DIV	L1 MAX	L1
Standard	1024	NaN	NaN	NaN	NaN	NaN
Standard	256	0.366975	0.000083	-0.210589	0.030676	0.005889
Standard	512	0.369820	0.000083	-0.208983	0.032110	0.005899
Peephole	1024	0.342686	0.000094	-0.226813	0.036187	0.005760
Peephole	256	0.355613	0.000085	-0.218492	0.032919	0.005825
Peephole	512	0.346774	0.000091	-0.224123	0.034359	0.005781

Figure 38: Minimum Validation Loss for different LSTM types and sizes

H2	MSE	DIV	L1 MAX	L1
0.36021	0.000081	-0.215393	0.030346	0.005847

Figure 39: Minimum Validation Loss for the best performing LSTM model

H2	MSE	DIV	L1 MAX	L1
0.365106	0.000076	-0.211963	0.0275	0.005877

Figure 40: Minimum Validation Loss for an ensemble of 5 LSTM models

XI.A.2 Mixture Model

Number of Distributions	H2	MSE	DIV	L1 MAX	L1
1	0.242886	0.000108	-0.271140	0.029988	0.005349
2	0.185605	0.000112	-0.336140	0.033075	0.005039
3	0.178443	0.000112	-0.341469	0.033409	0.005007
4	0.074123	0.000048	-0.292148	0.019272	0.002257
5	0.039166	0.000025	-0.285674	0.011980	0.001300
6	0.004963	0.000002	-0.260716	0.005065	0.000372
8	0.004098	0.000002	-0.274334	0.004872	0.000341
16	0.004160	0.000002	-0.207477	0.005007	0.000343
32	0.003968	0.000002	-0.209130	0.004879	0.000332
64	0.003714	0.000002	-0.217570	0.004746	0.000318

Figure 41: Minimum Validation Loss for different numbers of Gaussians

Batch Size	H2	MSE	DIV	L1 MAX	L1
128	0.001002	5.431698e-07	-0.272449	0.002245	0.000163
256	0.001445	7.430181e-07	-0.257867	0.002724	0.000194
512	0.002397	1.255268e-06	-0.259909	0.003521	0.000254
1024	0.004255	2.106039e-06	-0.267012	0.004788	0.000351
2048	0.007447	3.747046e-06	-0.290907	0.006572	0.000475

Figure 42: Minimum Validation Loss for different batch sizes

Optimiser	Loss	H2	MSE	DIV	L1 MAX	L1
Adam	H2	0.003647	1.931740e-06	-0.289956	0.004535	0.000318
Adam	L1	0.005569	2.449547e-06	0.004191	0.004605	0.000330
Adam	MDDEDIV	0.000454	3.273433e-07	-0.345629	0.001878	0.000121
Adam	MDDKLD	0.010005	4.498752e-06	-0.264407	0.007208	0.000550
Adam	MSE	0.014379	3.517463e-06	-0.144875	0.005877	0.000600
SGD	H2	0.437010	2.242867e-04	-0.145695	0.054646	0.006649
SGD	L1	0.965049	6.209610e-04	2.212706	0.106964	0.009700
SGD	MDDEDIV	0.355855	1.507516e-04	-0.202062	0.037870	0.005945
SGD	MDDKLD	0.413791	2.062625e-04	-0.162953	0.050932	0.006470
SGD	MSE	0.931416	5.906887e-04	1.979915	0.103181	0.009548

Figure 43: Minimum Validation Loss for different loss functions

Dense Activation	Parameter Activation	H2	MSE	DIV	L1 MAX	L1
ELU(alpha=1.0)	Clamped	0.292729	0.000117	-0.258812	0.034004	0.005541
ELU(alpha=1.0)	Exponential	0.175812	0.000112	-0.343054	0.033598	0.004997
ELU(alpha=1.0)	Softplus(beta=1, threshold=20)	0.176226	0.000110	-0.342970	0.032654	0.005000
ReLU()	Clamped	0.038657	0.000020	-0.311674	0.016242	0.001219
ReLU()	Exponential	0.008960	0.000004	-0.235422	0.006167	0.000506
ReLU()	Softplus(beta=1, threshold=20)	0.178130	0.000113	-0.341622	0.033533	0.005000
Softplus(beta=1, threshold=20)	Clamped	0.368433	0.000121	-0.209523	0.034099	0.005894
Softplus(beta=1, threshold=20)	Exponential	0.030627	0.000019	-0.262770	0.015570	0.001045
Softplus(beta=1, threshold=20)	Softplus(beta=1, threshold=20)	0.033786	0.000020	-0.300614	0.016318	0.001114
Tanh()	Clamped	0.178223	0.000111	-0.341584	0.035044	0.005002
Tanh()	Exponential	0.004261	0.000002	-0.280361	0.004700	0.000343
Tanh()	Softplus(beta=1, threshold=20)	0.014528	0.000004	-0.341571	0.006930	0.000579
Tanhshrink()	Clamped	0.049479	0.000023	-0.110702	0.017988	0.001294
Tanhshrink()	Exponential	NaN	NaN	NaN	NaN	NaN
Tanhshrink()	Softplus(beta=1, threshold=20)	NaN	NaN	NaN	NaN	NaN

Figure 44: Minimum Validation Loss for different activation functions

H2	MSE	DIV	L1 MAX	L1
0.000538	2.961174e-07	-0.300871	0.001511	0.000108

Figure 45: Minimum Validation Loss for Mixture Model trained with the Squared Hellinger Distance

H2	MSE	DIV	L1 MAX	L1
0.000095	6.200201e-08	-0.346132	0.000766	0.000054

Figure 46: Minimum Validation Loss for Mixture Model trained with the Maximum Exponential Divergence Distance

H2	MSE	DIV	L1 MAX	L1
0.003457	0.000002	-0.337599	0.004106	0.000273

Figure 47: Minimum Validation Loss for ensemble of 5 Mixture Models

XI.B Protein Distribution

XI.B.1 LSTM Model

H2	MSE	DIV	L1 MAX	L1
0.469361	0.001254	0.370608	0.312515	0.00693

Figure 48: Minimum Validation Loss for LSTM Model

H2	MSE	DIV	L1 MAX	L1
0.314929	0.000853	0.268331	0.231965	0.005414

Figure 49: Minimum Validation Loss for an ensemble of 5 LSTM Models

XI.B.2 Mixture Model

Number of Distributions	H2	MSE	DIV	L1 MAX	L1
1	0.327024	0.000267	2.763582	0.092562	0.003736
2	0.077675	0.000036	0.247811	0.033208	0.002169
3	0.052161	0.000029	0.191391	0.032986	0.001885
4	0.036786	0.000021	0.154179	0.030207	0.001673
5	0.026860	0.000016	0.124969	0.026716	0.001505
6	0.023504	0.000015	0.111989	0.026096	0.001407
8	0.025782	0.000016	0.123313	0.025397	0.001475
16	0.014816	0.000008	0.078032	0.017465	0.001108
32	0.012609	0.000007	0.070254	0.016728	0.001007

Figure 50: Minimum Validation Loss for different numbers of Gaussians

Number of Distributions	H2	MSE	DIV	L1 MAX	L1
1	0.018553	0.000011	0.081763	0.022607	0.001179
2	0.008533	0.000004	0.038923	0.011393	0.000739
3	0.007913	0.000003	0.030430	0.009762	0.000678
4	0.007860	0.000003	0.029604	0.009391	0.000669
5	0.007865	0.000003	0.029268	0.009045	0.000666
6	0.007748	0.000003	0.027307	0.008933	0.000654
8	0.007708	0.000003	0.026384	0.009249	0.000652
16	0.007555	0.000003	0.024800	0.008063	0.000640
32	0.007338	0.000003	0.022266	0.008377	0.000641

Figure 51: Minimum Validation Loss for different numbers of Gamma-Poissons

H2	MSE	DIV	L1 MAX	L1
0.007295	0.000002	0.020879	0.005897	0.000579

Figure 52: Minimum Validation Loss for Gamma-Poisson Mixture Model

H2	MSE	DIV	L1 MAX	L1
0.007323	0.000002	0.023029	0.006814	0.000591

Figure 53: Minimum Validation Loss for ensemble of 5 Gamma-Poisson Mixture Models

XI.C mRNA and Protein Joint Distribution

Number of Distributions	H2	MSE	DIV	L1 MAX	L1
1	0.316246	3.691391e-07	0.901115	0.000561	0.000042
2	0.139137	5.578090e-08	0.177821	0.000355	0.000029
4	0.100825	4.854157e-08	0.118900	0.000331	0.000026

Figure 54: Minimum Validation Loss for different numbers of Dirichlet-Multinomial Gamma-Poisson distributions

H2	MSE	DIV	L1 MAX	L1
0.089727	3.815524e-08	0.106211	0.000306	0.000024

Figure 55: Minimum Validation Loss for Mixture Model of 4 Dirichlet-Multinomial Gamma-Poisson distributions

H2	MSE	DIV	L1 MAX	L1
0.091038	3.841848e-08	0.107542	0.000307	0.000024

Figure 56: Minimum Validation Loss for a ensemble of 5 Mixture Models of 4 Dirichlet-Multinomial Gamma-Poisson distributions

XII Source Code

Source code for the simulation and all the models is provided below.

XII.A Regulating Gene Model Simulation

XII.A.1 Gillespie simulation joint Protein distribution

```
import gillespy2 as gp
from gillespy2.solvers.cpp import SSACSolver
import numpy as np
import os
from sklearn.model_selection import ParameterGrid
import mpi4py.MPI as MPI
import h5py
import time


class geneModel(gp.Model):
    """
    Gene Model gillespie simulator
    """
    def __init__(self):
        gp.Model.__init__(self,name='geneModel')# inherit

        self.paramGrid = None
        self.n=0

    def setValues(self,n):
        # set model parameters
        parameters = self.paramGrid[n]
        self.delete_all_parameters()
        self.delete_all_reactions()
        self.delete_all_species()

        # set reaction rates
        k0 = gp.Parameter(name='k0', expression=parameters[0])
        k1 = gp.Parameter(name='k1', expression=parameters[1])
        v0 = gp.Parameter(name='v0', expression=parameters[2])
        v1 = gp.Parameter(name='v1', expression=parameters[-1])#v1Value
        d0 = gp.Parameter(name='d0', expression=1.0)#1.0
```

```
        d1 = gp.Parameter(name='d1', expression=parameters[3])
        self.add_parameter([k0,k1,v0,v1,d0,d1])

        # define species
        ip = gp.Species(name='ip', initial_value=0)
        ap = gp.Species(name='ap', initial_value=1)
        m = gp.Species(name='m', initial_value=int(parameters[-2]))
        p = gp.Species(name='p', initial_value=100)
        self.add_species([ip,ap,m,p])

        # define reactions
        r_k0 = gp.Reaction(name="on", rate=k0, reactants={ip:1}, products={ap:1})
        r_k1 = gp.Reaction(name="off", rate=k1, reactants={ap:1}, products={ip:1})
        r_v0 = gp.Reaction(name="m_creation", rate=v0, reactants={ap:1}, products={m:1,ap:1})
        r_d0 = gp.Reaction(name="m_destruction", rate=d0, reactants={m:1}, products={})
        r_v1 = gp.Reaction(name="p_creation", rate=v1, reactants={m:1}, products={p:1,m:1})
        r_d1 = gp.Reaction(name="p_destruction", rate=d1, reactants={p:1}, products={})
        self.add_reaction([r_k0,r_k1,r_v0,r_d0,r_v1,r_d1])

        # get independent sample dists
        self.sample = int(np.maximum(np.ceil((1/parameters[np.arange(len(parameters))!=4])).max(),1))
        # get end time
        self.t = self.sample*10**5
        self.timespan(np.linspace(0,self.t,10**5+1))# set times to return results for

    def generateParams(self,n):
        # set parameters
        self.n = n
        k01 = np.power(10,np.random.uniform(-2,1,(self.n,2)))
        d1v0 = np.random.uniform(0.01,1,(self.n,2))
        self.paramGrid = np.ones((self.n,6))
        self.paramGrid[:,0:2] = k01
        self.paramGrid[:,2:4] = d1v0
        self.paramGrid[:,-2] =  self.paramGrid[:,2]* self.paramGrid[:,0]/(( self.paramGrid[:,0]+ self.paramGrid[:,1]))
        self.paramGrid[:,-1]= 100*self.paramGrid[:,3]/self.paramGrid[:,-2]



def MPE_DECOMP1D(n,numprocs,myid):
    """
    Python function to get start end indicies for mpi proc

    """
    nlocal = n // numprocs
    s = myid * nlocal
    deficit = n % numprocs
    s = s + min(myid,deficit)
    if myid < deficit:
        nlocal = nlocal +1
    e = s + nlocal
    if e > n or myid==numprocs-1:
        e = n
    return s,e



if __name__=="__main__":
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    nprocs = comm.Get_size()
    model = geneModel()
    n = 10**5 # number of samples
    if rank==0:# generate parameters on root
        model.generateParams(n)
        temp =model.paramGrid
    else:
        temp=None



    start,end = MPE_DECOMP1D(n,nprocs,rank) # get indicies
    temp = comm.bcast(temp,root=0) # bcast parameters to all processes
    if rank != 0:
        model.paramGrid = temp[start:end] # set paramGrids of model
    del temp

    sample = np.empty((end-start,10**5),dtype="f") # create array to store results
    if rank ==0:# start timer
        tic = time.perf_counter()
    for i in range(end-start):

        model.setValues(i)# set rates
        sample[i]= model.run(number_of_trajectories=1,solver=SSACSolver())["p"][1:]# run gillespie and store protein series
    # gather indicies on root
    ends = np.array(comm.gather(int(end),root=0))
    starts = np.array(comm.gather(int(start),root=0))
    comm.Barrier()
    if rank == 0:
        toc = time.perf_counter()# stop timer
        temp = np.empty((n,10**5),dtype="f")
        temp[start:end] = sample
```

```
        with h5py.File("modelData.hdf5","w") as f:# create file to store results
            f.create_dataset("results",data=temp,dtype="f")
            f.create_dataset("parameters",data=model.paramGrid)
            f.create_dataset("avg_time",data=np.array([(toc-tic)*nprocs/n]))
        del temp
    for i in range(1,nprocs):# send samples from each process individually due to Gather limit on elements
        if rank == i:
            comm.Send([sample,MPI.FLOAT],dest=0)
        elif rank == 0:
            temp = np.empty((ends[i]-starts[i],10**5),dtype="f")
            comm.Recv(temp,source=i)
            with h5py.File("modelData.hdf5","a") as f:# save sample from proc
                f["results"][starts[i]:ends[i]] = temp
```

XII.A.2 Gillespie simulation joint mRNA and Protein distribution

```
from GeneModelSimulation import *
class biGeneModel(geneModel):
    """
    Joint distribution of mrna and proteins
    """
    def __init__(self):
        super().__init__()# inherit
    def generateParams(self,n):
        # generate parameters
        self.n = n
        k01 = np.power(10,np.random.uniform(-2,1,(self.n,2)))
        d1 = np.random.uniform(0.01,1,(self.n))
        self.paramGrid = np.ones((self.n,6))
        self.paramGrid[:,0:2] = k01
        self.paramGrid[:,2] = 50*(( self.paramGrid[:,0]+ self.paramGrid[:,1]))/self.paramGrid[:,0]
        self.paramGrid[:,3] = d1
        self.paramGrid[:,-2] = 50
        self.paramGrid[:,-1]= 100*self.paramGrid[:,3]/self.paramGrid[:,-2]


if __name__=="__main__":
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    nprocs = comm.Get_size()
    model = biGeneModel()
    n = 10**5 # number of samples
    if rank==0:
        model.generateParams(n)# generate params on root
        temp =model.paramGrid
    else:
        temp=None



    start,end = MPE_DECOMP1D(n,nprocs,rank) # get indicies
    temp = comm.bcast(temp,root=0) #bcast params
    if rank != 0:
        model.paramGrid = temp[start:end] # set param grid
    del temp
    # create array to store proteins and mrna
    sampleP = np.empty((end-start,10**5),dtype="f")
    sampleM = np.empty((end-start,10**5),dtype="f")
    if rank ==0:# time
        tic = time.perf_counter()
    for i in range(end-start):

        model.setValues(i)# set rates
        res =  model.run(number_of_trajectories=1,solver=SSACSolver())# run gillespie
        # store samples
        sampleP[i]=res["p"][1:]
        sampleM[i]=res["m"][1:]
    # get indicies on root
    ends = np.array(comm.gather(int(end),root=0))
    starts = np.array(comm.gather(int(start),root=0))
    comm.Barrier()
    if rank == 0:
        toc = time.perf_counter()#stop timer
        temp1 = np.empty((n,10**5),dtype="f")
        temp1[start:end] = sampleP
        temp2 = np.empty((n,10**5),dtype="f")
        temp2[start:end] = sampleM
        # save data
        with h5py.File("modelDataBi.hdf5","w") as f:
            f.create_dataset("resultsP",data=temp1,dtype="f")
            f.create_dataset("resultsM",data=temp2,dtype="f")
            f.create_dataset("parameters",data=model.paramGrid)
            f.create_dataset("avg_time",data=np.array([(toc-tic)*nprocs/n]))
```

```
68          del temp1
69          del temp2
70      for i in range(1,nprocs):# we send from each proc seperatly rather than using Gather as Gather has a limit on the number of
        ↪  elements
71          if rank == i:# send proteins from i to root
72              comm.Send([sampleP,MPI.FLOAT],dest=0)
73          elif rank == 0:
74              temp = np.empty((ends[i]-starts[i],10**5),dtype="f")
75              comm.Recv(temp,source=i)
76              with h5py.File("modelDataBi.hdf5","a") as f:# save into file
77                  f["resultsP"][starts[i]:ends[i]] = temp
78          if rank == i:# send mrna from i to root
79              comm.Send([sampleM,MPI.FLOAT],dest=0)
80          elif rank == 0:
81              temp = np.empty((ends[i]-starts[i],10**5),dtype="f")
82              comm.Recv(temp,source=i)
83              with h5py.File("modelDataBi.hdf5","a") as f:# save into file
84                  f["resultsM"][starts[i]:ends[i]] = temp
```

XII.A.3 Vectorised Binning

```
1   import numpy as np
2   import h5py
3   import time
4   from tqdm import tqdm
5   from sklearn.preprocessing import scale
6   import mpi4py.MPI as MPI
7   from GeneModelSimulation import MPE_DECOMP1D
8   from numba import jit
9
10  @jit(nopython=True)
11  def vbins(x,b,r):
12
13      bins = np.linspace(r[0],r[1],b+1)# get bins
14      dig = np.searchsorted(bins, x, side='right')# get bins
15      res = np.empty((b,x.shape[0]),dtype=np.int32)
16      for i in range(b):# count numbers in each bin
17          res[i] = np.sum(dig==i+1,axis=-1)
18      res = res.T
19      s = res.sum(-1)
20      # stop division by zero and normalise
21      s[s==0] = 1
22      return res/s.reshape(x.shape[0],1)
23
24
25
26  if __name__=="__main__":
27      comm = MPI.COMM_WORLD
28      rank = comm.Get_rank()
29      nprocs = comm.Get_size()
30      t = 100 # number of time points
31      comm.Barrier()
32      if rank ==0:# start timer
33          tic = time.perf_counter()
34      with h5py.File("modelData.hdf5","r") as f:
35          tx = f["results"]
36          s = tx.shape
37          n= s[0]
38          start,end = MPE_DECOMP1D(n,nprocs,rank)# get index
39          it = np.append(np.arange(0,end-start,1536,dtype=int),[int(end-start)])# only load 1536 rows at once due to mem constraints
40          x = np.empty((end-start,t),dtype="f")
41          for i in range(len(it)-1):
42
43              x[it[i]:it[i+1]] = vbins(tx[(it[i]+start):(it[i+1]+start)],t,(0,300))# bin
44
45          if rank ==0:# in root store reaction rates and time
46              y = f["parameters"][:,np.arange(6)!=4]
47              times = f["avg_time"][:]
48      # gather send counts
49      sendcounts = np.array(comm.gather((end-start)*t, root=0))
50      disp = np.array(comm.gather((start)*t, root=0))
51      if rank == 0:
52          r = np.empty((n,t),dtype="f")
53
54      else:
55          r = None
56      # use gatherv to put results in r
57      comm.Gatherv(x,[r,sendcounts,disp,MPI.FLOAT],root=0)
58      comm.Barrier()
59      if rank ==0:
60
61          toc =time.perf_counter() # stop timer
62          # get scaling parameters
63          scalesM = np.mean(y,axis=0)
64          scalesS = np.std(y,axis=0)
65
66          with h5py.File(f"modelDataBin_{t}.hdf5","w") as f: # save binned data
67              f.create_dataset("results",data=r)
```

```
            f.create_dataset("scalesM",data=scalesM)
            f.create_dataset("scalesS",data=scalesS)

            f.create_dataset("parameters",data=scale(y))
            f.create_dataset("avg_time",data=times+(toc-tic)*nprocs/(s[0]))
```

XII.A.4 Bivariate Vectorised Binning

```
from binning import *

@jit(nopython=True)
def vbinsbi(x1,x2,b1,b2,r1,r2):
    # bivariate version of binner
    # get bin edges
    bins1 = np.linspace(r1[0],r1[1],b1+1)
    bins2 = np.linspace(r2[0],r2[1],b2+1)
    # get bin of each
    dig1 = np.searchsorted(bins1, x1, side='right')
    dig2 = np.searchsorted(bins2, x2, side='right')
    res = np.empty((x1.shape[0],b1,b2),dtype=np.int32)
    for i in range(b1):# for all bins for x1
        for j in range(b2):# for all bins for x2
            res[:,i,j] = np.sum((dig1==i+1) & (dig2==j+1),axis=-1)
    s = res.sum(-1).sum(-1)
    # normalise
    s[s==0] = 1
    return res/s.reshape(x1.shape[0],1,1)


if __name__=="__main__":
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    nprocs = comm.Get_size()
    t = 100 # set number of bins
    comm.Barrier()
    if rank ==0:
        tic = time.perf_counter()
    with h5py.File("modelDataBi.hdf5","r") as f:
        xP = f["resultsP"]
        xM = f["resultsM"]
        s = xP.shape
        n= s[0]
        start,end = MPE_DECOMP1D(n,nprocs,rank) # get indicies
        it = np.append(np.arange(0,end-start,768,dtype=int),[int(end-start)]) # only load 768 rows at a time
        x = np.empty((end-start,t,t),dtype="f")
        for i in range(len(it)-1): # bin data

            x[it[i]:it[i+1]] = vbinsbi(xM[(it[i]+start):(it[i+1]+start)],xP[(it[i]+start):(it[i+1]+start)],t,t,(0,300),(0,300))

        if rank ==0:# in root store reaction rates and time
            y = f["parameters"][:,np.arange(6)!=4]
            times = f["avg_time"][:]
    # send send counts and disp to root
    sendcounts = np.array(comm.gather((end-start)*t*t, root=0))
    disp = np.array(comm.gather((start)*t*t, root=0))
    if rank == 0:
        r = np.empty((n,t,t),dtype="f")

    else:
        r = None
    # gather results in root
    comm.Gatherv(x,[r,sendcounts,disp,MPI.FLOAT],root=0)
    comm.Barrier()
    if rank ==0:
        toc =time.perf_counter() # stop timer
        # get scaling parameters
        scalesM = np.mean(y,axis=0)
        scalesS = np.std(y,axis=0)

        with h5py.File(f"modelDataBiBin_{t}.hdf5","w") as f:# save binned results
            f.create_dataset("results",data=r)
            f.create_dataset("scalesM",data=scalesM)
            f.create_dataset("scalesS",data=scalesS)

            f.create_dataset("parameters",data=scale(y))
            f.create_dataset("avg_time",data=times+(toc-tic)*nprocs/(s[0]))
```

XII.B LSTM Model

XII.B.1 LSTM Cells

```
import torch
import torch.nn as nn

```

```
4
5   class LSTMCell(nn.Module):
6       """
7       Standard LSTM cell
8       """
9       def __init__(self, input_size : int, hidden_size : int):
10          super().__init__()
11          # set initial parameters
12          self.input_size = input_size
13          self.hidden_size = hidden_size
14          self.weight_ih = nn.Parameter(torch.randn(input_size,4 * hidden_size))
15          self.weight_hh = nn.Parameter(torch.randn(hidden_size,4 * hidden_size))
16          self.bias = nn.Parameter(torch.randn(4 * hidden_size))
17      @staticmethod
18      def __str__():
19          return "LSTM"
20      def forward(self,x,hx,cx ):
21          # forward method of LSTM
22          # do gate matric operations
23          gates = (torch.mm(x, self.weight_ih) + self.bias +
24                   torch.mm(hx, self.weight_hh))
25          ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)
26
27          # apply activation functions
28          ingate = torch.sigmoid(ingate)
29          forgetgate = torch.sigmoid(forgetgate)
30          cellgate = torch.tanh(cellgate)
31          outgate = torch.sigmoid(outgate)
32
33          # update cell and hidden state
34          cy = (forgetgate * cx) + (ingate * cellgate)
35          hy = outgate * torch.tanh(cy)
36
37          return hy, cy
38
39  class PeepholeLSTMCell(nn.Module):
40      """
41      Peephole LSTM
42      """
43      def __init__(self, input_size, hidden_size):
44          super().__init__()
45          # initialise parameters
46          self.input_size = input_size
47          self.hidden_size = hidden_size
48          self.weight_ih = nn.Parameter(torch.randn(input_size,3 * hidden_size))
49          self.weight_hh = nn.Parameter(torch.randn(hidden_size,3 * hidden_size))
50          self.bias =nn.Parameter(torch.randn(3*hidden_size))
51          self.weight_c =nn.Parameter(torch.randn(input_size,hidden_size))
52          self.bias_c =nn.Parameter(torch.randn(hidden_size))
53      @staticmethod
54      def __str__():
55          return "PeepholeLSTM"
56      def forward(self, x,hx,cx ):
57          # forward of cell takes hidden state despite it not being used so it can be interchanged with standard LSTM
58          # do matrix operations
59          gates = (torch.mm(x, self.weight_ih) + self.bias +
60                   torch.mm(cx, self.weight_hh) )
61          ingate, forgetgate, outgate = gates.chunk(3, 1)
62
63          # activations
64          ingate = torch.sigmoid(ingate)
65          forgetgate = torch.sigmoid(forgetgate)
66
67          outgate = torch.sigmoid(outgate)
68
69          # get new cell and hidden state
70          cy = (forgetgate * cx) + (ingate * torch.sigmoid(torch.mm(x, self.weight_c)+self.bias_c))
71          hy = outgate*cy
72
73          return hy,cy
74  class GRUCell(nn.Module):
75      """
76      Grated Recurrent Unit Cell
77      """
78      def __init__(self, input_size, hidden_size):
79          super().__init__()
80          self.input_size = input_size
81          self.hidden_size = hidden_size
82          self.weight_ih = nn.Parameter(torch.randn(input_size,2 * hidden_size))
83          self.weight_hh = nn.Parameter(torch.randn(hidden_size,2 * hidden_size))
84          self.bias =nn.Parameter(torch.randn(2*hidden_size))
85          self.weight_ic =nn.Parameter(torch.randn(input_size,hidden_size))
86          self.weight_hc =nn.Parameter(torch.randn(hidden_size,hidden_size))
87          self.bias_c =nn.Parameter(torch.randn(hidden_size))
88      @staticmethod
89      def __str__():
90          return "GRU"
91      def forward(self, x,hx,cx):
92          gates = (torch.mm(x, self.weight_ih) + self.bias +
93                   torch.mm(hx, self.weight_hh) )
94          updategate, resetgate = gates.chunk(2, 1)
```

```
95
96          updategate = torch.sigmoid(updategate)
97          resetgate = torch.sigmoid(resetgate)
98
99          hy = (updategate * hx) +
            ↪  (1.0-updategate)*torch.tanh(torch.mm(x,self.weight_ic)+torch.mm(resetgate*hx,self.weight_hc)+self.bias_c)
100
101         return hy,cx
```

XII.B.2 LSTM Model

```
1   import numpy as np
2   import os
3   from tqdm import tqdm
4   import torch.nn as nn
5   import torch
6   from torch.utils.data import DataLoader
7   from torch.utils.tensorboard import SummaryWriter
8   import torch.nn.functional as F
9   from scipy.stats import norm
10  import pickle
11  from sklearn.model_selection import train_test_split
12  from sklearn.model_selection import ParameterGrid
13  from sklearn.preprocessing import scale
14  import matplotlib.pyplot as plt
15  from multiprocessing import Pool
16  from functools import partial
17  import h5py
18  import time
19  #from copy import deepcopy
20  import pickle
21  import gc
22  from torch.distributions.normal import Normal
23  from cells import LSTMCell,PeepholeLSTMCell,GRUCell
24
25
26
27  def gaussianMix(x,points,dt):
28      g1 = norm(loc=x[3],scale=x[0])
29      g2 = norm(loc=x[4],scale=x[1])
30      g3 = norm(loc=x[5],scale=x[2])
31      return np.diff(x[6]*(g1.cdf(points))+x[7]*(g2.cdf(points))+x[8]*(g3.cdf(points)))/2
32
33  def h2_dist(x,y,ep=1e-12):
34      return
        ↪  2*((y.clamp(ep).sqrt()-x.clamp(ep).sqrt()).pow(2).sum(axis=tuple(np.linspace(1,x.dim()-1,x.dim()-1,dtype=np.int)))).mean()
35
36  def e_div(x,y,ep=1e-12):
37      return (x*(((x.clamp(ep)/(y.clamp(ep)))).log()).pow(2)).sum(axis=tuple(np.linspace(1,x.dim()-1,x.dim()-1,dtype=np.int))).mean()
38
39  def e_div2(x,y,ep=1e-12):
40      return (y*(((y.clamp(ep)/(x.clamp(ep)))).log()).pow(2)).sum(axis=tuple(np.linspace(1,x.dim()-1,x.dim()-1,dtype=np.int))).mean()
41
42  def gaussianLoss(x,y,sigma=0.5):
43      dist = Normal(x,sigma)
44      l = dist.log_prob(y)
45      ind = l.ne(l)
46      l[ind] = 0.0
47      return -l.sum(axis=-1).mean()
48
49  def polynomial(x,points):
50      return  np.diff(np.polynomial.polynomial.polyval(points,x))
51  kld = nn.KLDivLoss(reduction="batchmean")
52  def MDDKLD(x,y):
53      d1 = kld((x.clamp(1e-12)).log(),y)
54      d2 = kld((y.clamp(1e-12)).log(),x)
55      return torch.max(d1,d2)
56  def MDDEDIV(x,y):
57      d1 = e_div(x,y)
58      d2 = e_div2(x,y)
59      return torch.max(d1,d2)
60  def LOGMSE(x,y,ep=1e-12):
61      return (y.clamp(ep)/x.clamp(ep)).log().pow(2).mean()
62
63
64
65  def dataLoader(T=100,data="Gene"):
66      """
67      loads data from files into python and if it doesnt exist in the case
68      of the gaussian data it will generate it.
69      """
70      if data=="Gene":
71          print(f"Using modelDataBin_{T}.hdf5")
72          #load gene data
73          with h5py.File(f"modelDataBin_{T}.hdf5","r") as f:
74              xall = f["parameters"][:]
75              yall = f["results"][:]
76              scalesM =f["scalesM"][:]
```

```
            scalesS =f["scalesS"][:]
    elif data=="BiGene":
        print(f"Using modelDataBiBin_{T}.hdf5")
        #load bigene data
        with h5py.File(f"modelDataBiBin_{T}.hdf5","r") as f:
            xall = f["parameters"][:]
            yall = f["results"][:]
            scalesM =f["scalesM"][:]
            scalesS =f["scalesS"][:]


    elif data == "Gaussian":

        path = f"gaussian_{T}.hdf5"
        if os.path.exists(path):
            #load gaussian data
            print(f"Loading gaussian mixture from {path}")
            with h5py.File(path,"r") as f:
                xall=f["parameters"][:]
                scalesM =f["scalesM"][:]
                scalesS =f["scalesS"][:]
                yall = f["results"][:]
        else:

            print("Generating mixture of gaussians...")
            n = 10**5 # samples
            mu = np.random.uniform(-3,3,size=(n,3))
            var = np.random.uniform(0.1,5,size=(n,3))
            k = np.zeros((n,4))
            k[:,3] = 1
            k[:,1:3] = np.sort(np.random.uniform(0,1,size=(n,2)),axis=-1)
            k = np.diff(k)
            x = np.hstack((var,mu,k))

            points = np.linspace(-15,15,T+1)

            dt = 30/T

            # openmp parallelised calls function to get the bin probability
            with Pool(os.cpu_count()-1) as p:
                y = p.map(partial(gaussianMix,points=points,dt=dt),x)
            y= np.array(y)
            # get scaling parameters
            scalesM = np.mean(x,axis=0)
            scalesS = np.std(x,axis=0)
            xall = scale(x) # scale input
            yall = y
            # save gaussian data
            with h5py.File(path,"w") as f:
                f.create_dataset("results",data=yall)
                f.create_dataset("scalesM",data=scalesM)
                f.create_dataset("scalesS",data=scalesS)
                f.create_dataset("parameters",data=xall)
    return xall,yall,scalesM,scalesS

class model(nn.Module):
    """
    LSTM Model class
    """
    def __init__(self,input_dim=9,
                 T=100,prev=32,lstm_size=512,device_index=0,
                 batch_size=1024,e_learning_rate=1e-5,bidir=True,
                 output_act=nn.Sigmoid(),output_act_dense=nn.Softmax(dim=1),dense_act=nn.ELU(),celltype=LSTMCell):

        super().__init__()
        # set model constants
        self.input_dim = input_dim
        self.T = T
        self.prev = prev
        self.batch_size = batch_size
        self.e_learning_rate = e_learning_rate
        self.enc_size = lstm_size
        self.output_act_type = output_act
        self.output_dense_act_type = output_act_dense
        self.dense_act_type = dense_act
        self.bidir = bidir
        self.celltype=celltype
        self.device_index = device_index

        self._build_net()
    def _build_net(self):
        # build layers
        self.use_cuda = torch.cuda.is_available()
        self.device = torch.device(f"cuda:{self.device_index}" if self.use_cuda else "cpu")
        self.index_tensor = torch.tensor([[[t]]*self.batch_size for t in
        ↪  range(self.T+1)]).to(self.device,non_blocking=self.use_cuda)
        self.input_1_1 = nn.Linear(self.input_dim,32).to(self.device,non_blocking=self.use_cuda)
        self.input_act = self.dense_act_type.to(self.device,non_blocking=self.use_cuda)
        self.input_1_2 = nn.Linear(32,64).to(self.device,non_blocking=self.use_cuda)
        self.input_1_3 = nn.Linear(64,128).to(self.device,non_blocking=self.use_cuda)
        self.input_1_4 = nn.Linear(128,256).to(self.device,non_blocking=self.use_cuda)
```

```
        self.input_1_5 = nn.Linear(256,128).to(self.device,non_blocking=self.use_cuda)
        self.input_1_x = nn.ModuleList([nn.Linear(128,64) for _ in range(self.T)]).to(self.device,non_blocking=self.use_cuda)


        self.output_layer=nn.Linear((self.T)*(self.bidir+1),self.T).to(self.device,non_blocking=self.use_cuda)
        self.output_act = self.output_act_type.to(self.device,non_blocking=self.use_cuda)
        self.loss = torch.zeros(1,dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)
        # build lstm array
        if self.bidir:


            self.lstmArray("array1",self.input_dim,self.T,self.prev,self.enc_size,self.batch_size,self.dense_act_type,self.output_act_type)

            self.lstmArray("array2",self.input_dim,self.T,self.prev,self.enc_size,self.batch_size,self.dense_act_type,self.output_act_type)
        else:

            self.lstmArray("array1",self.input_dim,self.T,self.prev,self.enc_size,self.batch_size,self.dense_act_type,self.output_act_type)

    def lstmArray(self,array,input_dim=4,
                  T=100,prev=32,lstm_size=512,
                  batch_size=128,dense_act=nn.ReLU(),output_act=nn.Sigmoid(),celltype=LSTMCell):

        ys = [nn.Linear(self.enc_size,1) for _ in range(self.T)]
        setattr(self,array+"_ys",nn.ModuleList(ys).to(self.device,non_blocking=self.use_cuda))
        ys_sigmoid = [output_act for _ in range(self.T)]
        setattr(self,array+"_ys_sigmoid",nn.ModuleList(ys_sigmoid).to(self.device,non_blocking=self.use_cuda))
        y_prev_dense = [nn.Linear(self.prev,128) for _ in range(self.T)]
        setattr(self,array+"_y_prev_dense",nn.ModuleList(y_prev_dense).to(self.device,non_blocking=self.use_cuda))
        y_prev_dense_act = [dense_act for _ in range(self.T)]
        setattr(self,array+"_y_prev_dense_act",nn.ModuleList(y_prev_dense_act).to(self.device,non_blocking=self.use_cuda))
        #setattr(self,array+"_dropouts",nn.ModuleList([nn.Dropout(0.1) for _ in range(self.T+1)]).to(self.device,non_blocking=self.use_cuda))

        setattr(self,array+"_lstm_enc",torch.jit.script(self.celltype(192,self.enc_size)).to(self.device,non_blocking=self.use_cuda))

    def _embedding_input(self,x):
        # forward function for initial dense layers
        x = self.input_1_1(x)
        x = self.input_act(x)
        x = self.input_1_2(x)
        x = self.input_act(x)
        x = self.input_1_3(x)
        x = self.input_act(x)
        x = self.input_1_4(x)
        x = self.input_act(x)
        x = self.input_1_5(x)
        x = self.input_act(x)
        return x
    def forward(self,x):

        # put through initial dense layers
        xe = self._embedding_input(x)
        # Feed through LSTM array
        ys = self._forwardLoop1(xe)
        if self.bidir: # if bidirectional feed through other lstm array and concatenate
            ys =  torch.cat((self._forwardLoop2(xe),ys),1)
        # feed through final dense layer and softmax
        x=ys
        x = self.output_layer(x)
        x = self.output_dense_act_type(x)
        return x

    def _forwardLoop1(self,xe):
        # forward LSTM array
        enc_state_0,enc_state_1 = self.initial_hidden() # initialise hidden layers
        # initialise list to store previous outputs
        y_prev = [torch.zeros([self.batch_size,1],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)] * self.prev
        # initialise result tensor
        ys = torch.zeros([self.batch_size,self.T],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)

        for t in range(0,self.T):# go through time
            # feed previous results into dense layer
            x = self.array1_y_prev_dense[t](torch.cat(tuple(y_prev),dim=1))
            x = self.array1_y_prev_dense_act[t](x)
            # feed output initial of dense layers into dense layer
            xt = self.input_1_x[t](xe)
            xt = self.input_act(xt)
            x = torch.cat((xt,x),1)# concatenate
            # feed concatenated results into lstm array
            enc_state_0,enc_state_1 = self.array1_lstm_enc(x,enc_state_0,enc_state_1)
            # put output into dense layer
            x = self.array1_ys[t](enc_state_0)
            x = self.array1_ys_sigmoid[t](x)
            ys.scatter_(1,self.index_tensor[t],x) # put results into results tensor
            # update previous tensor
            y_prev = y_prev[1:] +[x]


```

```
250             return ys
251
252         def _forwardLoop2(self,xe):
253             enc_state_0,enc_state_1 = self.initial_hidden() # initialise hidden states of lstm
254             # initialise list to store previous results
255             y_prev = [torch.zeros([self.batch_size,1],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)] * self.prev
256             # initialise tensor to store output
257             ys = torch.zeros([self.batch_size,self.T],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)
258
259             for t in range(0,self.T): # go through time index in for loop reverses order
260                 # feed previous states into dense layer
261                 x = self.array2_y_prev_dense[t](torch.cat(tuple(y_prev),dim=1))
262                 x = self.array2_y_prev_dense_act[t](x)
263                 # feed output of initial dense layers into dense layer
264                 xt = self.input_1_x[-t-1](xe)
265                 xt = self.input_act(xt)
266                 x = torch.cat((xt,x),1)#concetenate
267                 # feed into lstm array
268                 enc_state_0,enc_state_1 = self.array2_lstm_enc(x,enc_state_0,enc_state_1)
269                 # feed output into dense layer and apply softmax
270                 x = self.array2_ys[t](enc_state_0)
271                 x = self.array2_ys_sigmoid[t](x)
272                 # cast to results tensor
273                 ys.scatter_(1,self.index_tensor[t],x)
274                 # update previous results list
275                 y_prev = y_prev[1:] +[x]
276             return ys
277
278         @staticmethod
279         def get_yprev23(y_prev,t,ys):
280             return torch.cat((y_prev[:,1:],ys[:,t-1].unsqueeze(1)),1)
281
282         def initial_hidden(self):
283             # create initial states for lstm arrays
284             enc_state_0 = torch.zeros([self.batch_size,self.enc_size],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)
285             enc_state_1 = torch.zeros([self.batch_size,self.enc_size],dtype=torch.float32).to(self.device,non_blocking=self.use_cuda)
286             return (enc_state_0,enc_state_1)
287         @torch.jit.export
288         def train_model(self,epochs=100,data="Gene",
289                         optim="Adam",loss_fn="MSE",custom_data=None,
290                         model_name=None):
291             # create directory to store models
292             self._directory_maker()
293             # set constants
294             self.epochs = epochs
295             self.loss_fn = loss_fn
296             self.optim = optim
297
298             if custom_data is None:
299                 # load data
300                 xall,yall=self.load_data(data=data)
301                 # create train and test split
302                 x_train, x_test,y_train,y_test = train_test_split(xall,yall,train_size=0.8)
303             else:
304                 # if custom data is given use that
305                 x_train, x_test,y_train,y_test =
                    ↪  custom_data["x_train"],custom_data["x_test"],custom_data["y_train"],custom_data["y_test"]
306                 self.scalesM,self.scalesS = custom_data["scalesM"],custom_data["scalesS"]
307             print(f"Loading model onto {self.device}")
308             print(f"Loading tensorboard writer")
309             # create model name
310             if model_name is None:
311                 oact = str(self.output_act_type).split("(",1)[0]
312                 dact = str(self.dense_act_type).split("(",1)[0]
313                 odact = str(self.output_dense_act_type).split("(",1)[0]
314                 celltype = self.celltype.__str__()
315                 t = time.strftime("%d-%m-%H-%M-%S",time.localtime())
316                 self.model_name = f"cell={celltype}_b={self.batch_size}_loss={self.loss_fn}_oact={oact}_dact={dact}_odact={odact}"
317             else:
318                 self.model_name = model_name
319             # start tensorboard log
320             writer = SummaryWriter(comment=self.model_name)
321             # create optimiser
322             self._init_optim()
323             print("Loading dataset")
324             # create data loaders for train and test data
325             xtrainTensor=torch.from_numpy(x_train).type(torch.float32)
326             ytrainTensor=torch.from_numpy(y_train).type(torch.float32)
327             dataset = torch.utils.data.TensorDataset(xtrainTensor,ytrainTensor)
328             loader = DataLoader(dataset, batch_size=self.batch_size,drop_last=True,shuffle=True,
329                                 pin_memory=self.use_cuda,num_workers=2)
330             xtestTensor=torch.from_numpy(x_test).type(torch.float32)
331             ytestTensor=torch.from_numpy(y_test).type(torch.float32)
332             valDataset = torch.utils.data.TensorDataset(xtestTensor,ytestTensor)
333             valLoader = DataLoader(valDataset, batch_size=self.batch_size,drop_last=True,shuffle=False,
334                                 pin_memory=self.use_cuda,num_workers=2)
335             # create matrix to store loss
336             self._init_loss()
337             old_test_loss = np.inf
338             train_batches = x_train.shape[0] // self.batch_size
339             test_batches =x_test.shape[0] // self.batch_size
```

```
        del valDataset,dataset,xtestTensor,ytestTensor,x_train,x_test,y_train,y_test
        # start train loop
        print("Train Loop Starting...")
        for i in range(self.epochs):
            #nvtx.range_push("Epoch " + str(i+1))
            self.train() # put model in train mode
            print(f"STEP: {i+1}/{self.epochs}")
            #nvtx.range_push("Training")
            # train
            with tqdm(total=train_batches,ascii=True,desc="Training") as pbar:# loading bar
                for idx, (local_batch, local_labels) in enumerate(loader):
                    #nvtx.range_push("Batch "+str(idx))

                    #nvtx.range_push("Copying to device")
                    # get batch and send to device
                    y=local_labels.to(self.device,non_blocking=self.use_cuda)
                    x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                    #nvtx.range_pop()
                    #nvtx.range_push("Forward pass")
                    # forward pass
                    x=self(x_batch)
                    #nvtx.range_pop()

                    #nvtx.range_push("Backward pass")
                    # update loss and backpropigate
                    self._update_loss_train(x,y,i)


                    #nvtx.range_pop()
                    #nvtx.range_pop()



                    pbar.update(1)

            #nvtx.range_pop()
            # average
            self.train_loss[i] = self.train_loss[i]/train_batches
            if np.isnan(self.train_loss[i,0]):# if NaN stop training
                print("GOT NaN")
                break
            # update tensorbard and print
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/train'
                writer.add_scalar(name, float(self.train_loss[i,l]), i)
                print(f"{name}: {self.train_loss[i,l]}")



            #nvtx.range_push("Evaluation")
            # get val loss
            with torch.no_grad():
                self.eval() # set to eval
                with tqdm(total=test_batches,ascii=True,desc="Validating") as pbar:# loading bar
                    for local_batch, local_labels in valLoader:
                        # get batch
                        y = local_labels.to(self.device,non_blocking=self.use_cuda)
                        x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                        # forward pass
                        pred = self(x_batch)
                        # update val loss
                        self._update_loss_val(pred,y,i)
                        pbar.update(1)
            #nvtx.range_pop()
            # average
            self.val_loss[i] = self.val_loss[i]/test_batches
            # update tensorboard and print results
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/test'
                writer.add_scalar(name, float(self.val_loss[i,l]), i)
                print(f"{name}: {self.val_loss[i,l]}")
            #nvtx.range_pop()
            # save modelif val loss is lower than previous best
            if self.val_loss[i,0] < old_test_loss:
                #nvtx.mark("Saving best model")
                torch.save({"state_dict":self.state_dict(),
                            "scalesM":self.scalesM,
                            "scalesS":self.scalesS,
                            "epoch":i+1,
                            "optim":self.optimiser,
                            "val_loss":self.val_loss,
                            "dense_act":str(self.dense_act_type),
                            "output_act":str(self.output_act_type),
                            "batch_size": self.batch_size,
                            "enc_size":self.enc_size,
                            "bidir":self.bidir,
                            "T": self.T,
                            "output_dense_act":self.output_dense_act_type,
                            "prev": self.prev,
                            "input_dim": self.input_dim,
                            "loss_fn":self.loss_fn}, f"./best/bestModel_{self.model_name}.pth")
```

```
                old_test_loss = self.val_loss[i,0]
        # save final model
        torch.save({"state_dict":self.state_dict(),
                        "scalesM":self.scalesM,
                        "scalesS":self.scalesS,
                        "epoch":self.epochs,
                        "optim":self.optimiser,
                        "val_loss":self.val_loss,
                        "dense_act":str(self.dense_act_type),
                        "output_act":str(self.output_act_type),
                        "batch_size": self.batch_size,
                        "enc_size":self.enc_size,
                        "bidir":self.bidir,
                        "T": self.T,
                        "output_dense_act":self.output_dense_act_type,
                        "prev": self.prev,
                        "input_dim": self.input_dim,
                        "loss_fn":self.loss_fn}, f"./final/finalModel_{self.model_name}.pth")
        return self.train_loss,self.val_loss
    def _directory_maker(self):
        # create directories for models
        try:
            print("Making directory for best models")
            os.mkdir("./best")
        except FileExistsError:
            print("Directory for best models exists")
        try:
            print("Making directory for final models")
            os.mkdir("./final")
        except FileExistsError:
            print("Directory for final models exists")

    def _init_optim(self):
        # create optimiser
        if self.optim == "Adam":
            self.optimiser = torch.optim.Adam(self.parameters(),
                                        lr=self.e_learning_rate,
                                        betas=(0.5,0.999))#,weight_decay=0.1
        elif self.optim == "SGD":
            self.optimiser = torch.optim.SGD(self.parameters(),
                                        lr=self.e_learning_rate)
                                        #momentum=0.1),weight_decay=0.1

    def _init_loss(self):
        # initialise loss functions
        self.mse_fn = nn.MSELoss()
        self._kld = nn.KLDivLoss(reduction="batchmean")
        self.div_fn = lambda x,y: self._kld(x.clamp(1e-20).log(),y)
        self.l1LossFN_t = nn.L1Loss(reduction="none")
        self.l1LossFN = nn.L1Loss()
        self.h2 = h2_dist
        self.ediv = e_div
        self.ediv2 = e_div2
        self.MDDKLD = MDDKLD
        self.MDDEDIV = MDDEDIV
        self.LOGMSE = LOGMSE
        #self.ediv_c = e_div_c
        #self.a_mse =lambda x,y : adj_mse(x,y)+ dist_loss(x,y)

        self.l1_max_fn = lambda x,y : self.l1LossFN_t(x,y).max(dim=-1)[0].mean()
        allLoss = np.array(["MSE","DIV","L1 MAX","L1","H2","EDIV","EDIV2","MDDKLD","MDDEDIV","LOGMSE"])
        lossFns =
        ↪  np.array([self.mse_fn,self.div_fn,self.l1_max_fn,self.l1LossFN,self.h2,self.ediv,self.ediv2,self.MDDKLD,self.MDDEDIV,self.LOGMSE])
        ind1 = np.where(allLoss==self.loss_fn)
        ind2 = np.where(allLoss!=self.loss_fn)
        s = allLoss.size
        self.loss_names = np.concatenate((allLoss[ind1],allLoss[ind2]))
        self.loss_fns = np.concatenate((lossFns[ind1],lossFns[ind2]))
        # create array to store loss
        self.train_loss = np.zeros((self.epochs,s))
        self.val_loss = np.zeros((self.epochs,s))

    def _update_loss_train(self,x,y,i):
        # get loss
        loss = self.loss_fns[0](x,y)
        self.optimiser.zero_grad()
        # backpropigate
        loss.backward()
        self.optimiser.step()
        # update loss
        self.train_loss[i,0] += float(loss.detach().cpu())
        self.train_loss[i,1:] += [float(s(x.detach(),y).cpu()) for s in self.loss_fns[1:]]


    def _update_loss_val(self,x,y,i):
        # update validation loss
        self.val_loss[i] += [float(s(x.detach(),y).cpu()) for s in self.loss_fns]

    def load_data(self,data="Gene"):
        #load data using data loader function
        xall,yall,self.scalesM,self.scalesS = dataLoader(T=self.T,data=data)
```

```
        return xall,yall

    @torch.jit.export
    def load_model(self,PATH):
        # load model
        model = torch.load(PATH,map_location=self.device)
        self.scalesM = model["scalesM"]
        self.scalesS = model["scalesS"]
        self.output_act_type = eval("nn."+model["output_act"])
        self.dense_act_type = eval("nn."+model["dense_act"])
        self.batch_size = model["batch_size"]
        self.bidir = model["bidir"]
        self.enc_size = model["enc_size"]
        self.T = model["T"]
        self.prev = model["prev"]
        self.input_dim=model["input_dim"]
        #self._build_net()
        self.load_state_dict(model["state_dict"])




    @torch.jit.export
    def predict(self,x):
        # predict outputs
        self.eval()#eval mode
        #scale
        x = (np.atleast_2d(x) - self.scalesM)/self.scalesS
        p = int(np.ceil(x.shape[0] /self.batch_size))
        t = x.shape[0]
        r = self.batch_size - (t % self.batch_size)
        x = np.pad(x,((0,r),(0,0)))
        xs = np.array_split(x,p)
        res = [None] * p
        for i in range(p):# feed x in batch size chunks
            res[i] =
            ↪  self(torch.from_numpy(xs[i]).type(torch.float32).to(self.device,non_blocking=self.use_cuda)).detach().cpu().numpy()

        res[-1] = res[-1][:-r]
        return np.vstack(tuple(res))

if __name__ == "__main__":
    # learning_rate=1e-4
    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")
    cell_types = [LSTMCell,PeepholeLSTMCell,GRUCell]
    lstm_size=[512,1024]
    dense_act = [nn.ELU(),nn.ReLU(),nn.Tanhshrink(),nn.Tanh(),nn.Softplus()]
    output_act = [nn.Sigmoid(),nn.Softplus()]
    output_act_dense =[nn.Sigmoid(),nn.Softplus(),nn.Softmax()]
    loss = ["MSE","L1","H2","MDDKLD","MDDEDIV","LOGMSE"]
    optim = ["Adam","SGD"]
    prev = [1,4,8,16,32,64]
    bidir = [True,False]
    g = ParameterGrid({"optim":optim,"loss":loss})

    val_loss = [[None,i] for i in g]
    train_loss = [[None,i] for i in g]


    for v,i in enumerate(g):
        try:
            print(i)
            m = model(bidir=False,e_learning_rate=1e-4,batch_size=128).to(device)
            train_loss[v][0],val_loss[v][0] = m.train_model(data="Gaussian",optim=i["optim"],loss_fn=i["loss"],epochs=100)

        except Exception as inst:
            print("ERROR!!!!")
            print(type(inst))
            print(inst)
    with open('train_loss.pickle', 'wb') as f:
        pickle.dump(train_loss, f, protocol=pickle.HIGHEST_PROTOCOL)
    with open('val_loss.pickle', 'wb') as f:
        pickle.dump(val_loss, f, protocol=pickle.HIGHEST_PROTOCOL)
```

XII.B.3 Ensemble LSTM Model

```
from pytorchModel import *

class ensembleLSTM(model):
    """
    ensemble of lstm models
    """
    def __init__(self,T,n,batch_size=1024,output_activation=nn.Softmax(dim=1),learning_rate=1e-5):
```

```
8           super(model,self).__init__()# inherit
9           # set consts
10          self.use_cuda = torch.cuda.is_available()
11          self.device = torch.device("cuda" if self.use_cuda else "cpu")
12          self.n = n
13          self.T = T
14          self.batch_size=batch_size
15          self.e_learning_rate=learning_rate
16          # create layer
17          self.output_activation=output_activation
18          self.linear = nn.Linear(self.n*(self.T),self.T).to(self.device,non_blocking=self.use_cuda)
19
20      def forward(self,x):
21          # forward method
22          return self.output_activation(self.linear(x))
23      @torch.jit.export
24      def train_model(self,loss_fn,optim,epochs,data):
25          self._directory_maker() # create directory to store models
26          # set const
27          self.epochs = epochs
28          self.loss_fn = loss_fn
29          self.optim = optim
30          # create date
31          self.create_data(data)
32          # set model name
33          self.model_name = f"ensemble_n={self.n}"
34          # create tensorboard writer
35          writer = SummaryWriter(comment=self.model_name)
36          # create optimiser
37          self._init_optim()
38          print("Loading dataset")
39          # create data loader
40          xtrainTensor=torch.from_numpy(self.x_train).type(torch.float32)
41          ytrainTensor=torch.from_numpy(self.y_train).type(torch.float32)
42          dataset = torch.utils.data.TensorDataset(xtrainTensor,ytrainTensor)
43          loader = DataLoader(dataset, batch_size=self.batch_size,drop_last=True,shuffle=True,
44                      pin_memory=self.use_cuda,num_workers=2)
45          xtestTensor=torch.from_numpy(self.x_test).type(torch.float32)
46          ytestTensor=torch.from_numpy(self.y_test).type(torch.float32)
47          valDataset = torch.utils.data.TensorDataset(xtestTensor,ytestTensor)
48          valLoader = DataLoader(valDataset, batch_size=self.batch_size,drop_last=True,shuffle=False,
49                      pin_memory=self.use_cuda,num_workers=2)
50          # create loss array
51          self._init_loss()
52          old_test_loss = np.inf
53          train_batches = self.x_train.shape[0] // self.batch_size
54          test_batches =self.x_test.shape[0] // self.batch_size
55          print("Train Loop Starting...")
56          # start train loop
57          for i in range(self.epochs):
58              #nvtx.range_push("Epoch " + str(i+1))
59              self.train()
60              print(f"STEP: {i+1}/{self.epochs}")
61              #nvtx.range_push("Training")
62              with tqdm(total=train_batches,ascii=True,desc="Training") as pbar:#loading bar
63                  for idx, (local_batch, local_labels) in enumerate(loader):
64                      #nvtx.range_push("Batch "+str(idx))
65
66                      #nvtx.range_push("Copying to device")
67                      # get batches
68                      y=local_labels.to(self.device,non_blocking=self.use_cuda)
69                      x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
70                      #nvtx.range_pop()
71                      #nvtx.range_push("Forward pass")
72                      # forward pass
73                      x=self(x_batch)
74                      #nvtx.range_pop()
75
76                      #nvtx.range_push("Backward pass")
77                      # update loss and back prop
78                      self._update_loss_train(x,y,i)
79
80
81                      #nvtx.range_pop()
82                      #nvtx.range_pop()
83
84
85
86                      pbar.update(1)
87
88              #nvtx.range_pop()
89              # average
90              self.train_loss[i] = self.train_loss[i]/train_batches
91              if np.isnan(self.train_loss[i,0]):# break if nan
92                  print("GOT NaN")
93                  break
94              # write loss to tensorboard log and print loss
95              for l,s in enumerate(self.loss_names):
96                  name=f'Loss {s}/train'
97                  writer.add_scalar(name, float(self.train_loss[i,l]), i)
98                  print(f"{name}: {self.train_loss[i,l]}")
```

```
99
100
101
102             #nvtx.range_push("Evaluation")
103             # get val loss
104             with torch.no_grad():
105                 self.eval() # set to eval mode
106                 with tqdm(total=test_batches,ascii=True,desc="Validating") as pbar:# loading bar
107                     for local_batch, local_labels in valLoader:
108                         #get batch
109                         y = local_labels.to(self.device,non_blocking=self.use_cuda)
110                         x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
111                         # forward pass
112                         pred = self(x_batch)
113                         # update val loss
114                         self._update_loss_val(pred,y,i)
115                         pbar.update(1)
116             #nvtx.range_pop()
117             # average
118             self.val_loss[i] = self.val_loss[i]/test_batches
119             # write tensorboard log print loss
120             for l,s in enumerate(self.loss_names):
121                 name=f'Loss {s}/test'
122                 writer.add_scalar(name, float(self.val_loss[i,l]), i)
123                 print(f"{name}: {self.val_loss[i,l]}")
124             #nvtx.range_pop()
125             # if model has the lowest val loss save
126             if self.val_loss[i,0] < old_test_loss:
127                 torch.save({"state_dict":self.state_dict(),
128                             "n":self.n,
129                             "epoch":i+1,
130                             "optim":self.optimiser,
131                             "val_loss":self.val_loss,
132                             "batch_size": self.batch_size,
133                             "T": self.T,
134                             "loss_fn":self.loss_fn}, f"./best/bestModel_{self.model_name}.pth")
135         # save final model
136         torch.save({"state_dict":self.state_dict(),
137                             "n":self.n,
138                             "epoch":i+1,
139                             "optim":self.optimiser,
140                             "val_loss":self.val_loss,
141                             "batch_size": self.batch_size,
142                             "T": self.T,
143                             "loss_fn":self.loss_fn}, f"./final/finalModel_{self.model_name}.pth")
144         return self.train_loss,self.val_loss
145     def _load_model(self,i):
146         # load sub models and set to eval
147         m = model(bidir=True,batch_size=512, input_dim=self.inp, T=100, dense_act=nn.ELU(),output_act=nn.Softplus(),
            ↪  output_act_dense=nn.Softmax()).to(self.device)
148         m.load_model(f"bestModel_ensemble_model_lstm_{self.data}_{i+1}.pth")
149         m.eval()
150
151         return m
152
153     def _submodel_predict(self,x):
154         # load each model and get its prediction in series
155         # done in series to save memory
156         out = [None] * self.n
157         for i in range(self.n):
158             out[i] = self._load_model(i).predict(x)
159         return np.hstack(tuple(out))
160     def create_data(self,data):
161         # set const
162         self.data=data
163         if data == "gau":
164             self.inp = 9
165         else:
166             self.inp = 5
167         # load data
168         with h5py.File(f"ensemble_data_{data}.hdf5","r") as f:
169             x_train = f["x_train"][:]
170             x_test = f["x_test"][:]
171             y_train = f["y_train"][:]
172             y_test = f["y_test"][:]
173         self.x_train,self.x_test,self.y_train,self.y_test
            ↪  =self._submodel_predict(x_train),self._submodel_predict(x_test),y_train,y_test
174
175 if __name__=="__main__":
176     use_cuda = torch.cuda.is_available()
177     device = torch.device("cuda" if use_cuda else "cpu")
178
179
180     m = ensemble(n=10,T=100).to(device)
181     m(torch.randn((1024,10*101)).to(device))
182     train_loss,val_loss=m.train_model("EDIV2","adam",500)
183     with open('train_loss.pickle', 'wb') as f:
184         pickle.dump(train_loss, f, protocol=pickle.HIGHEST_PROTOCOL)
185     with open('val_loss.pickle', 'wb') as f:
186         pickle.dump(val_loss, f, protocol=pickle.HIGHEST_PROTOCOL)
187
```

XII.C Mixture Model

XII.C.1 Mixture Model

```
from torch.distributions.normal import Normal
from torch.distributions.log_normal import LogNormal
from torch.distributions.poisson import Poisson
from torch.distributions.exp_family import ExponentialFamily
from torch.distributions import constraints
from torch.distributions.utils import broadcast_all
from torch.distributions.categorical import Categorical
from numbers import Number
from pytorchModel import *
import itertools


# define these wrappers so functions have a readable name when converted to a string

class Exponential():
    def __init__(self):
        return None
    def __call__(self,x):
        return x.exp().clamp(1e-12)
    def __str__(self):
        return "Exponential"

class Clamped():
    def __init__(self,ep=1e-12):
        self.ep = ep
        return None
    def __call__(self,x):
        return x.clamp(self.ep)
    def __str__(self):
        return "Clamped"



class gammaPoisson(ExponentialFamily):
    """
    create gamma-poisson distribution class
    """


    arg_constraints = {'alpha': constraints.positive,
                       'beta': constraints.positive}
    support = constraints.nonnegative_integer
    def __init__(self,alpha,beta,validate_args=None):
        # set parameters
        self.alpha,self.beta = broadcast_all(alpha,beta)
        if isinstance(alpha, Number):
            batch_shape = torch.Size()
            self.device = torch.device("cpu")
            self.batch_size = torch.tensor([1])
        else:
            batch_shape = self.alpha.size()
            self.device = self.alpha.device
            self.batch_size = batch_shape[:-1]

        super(gammaPoisson, self).__init__(batch_shape, validate_args=validate_args)
    def cdf(self,x):
        if self._validate_args:
            self._validate_sample(x)
        x, = broadcast_all(x)
        n = x.floor().long() # cast to integers
        # create tensor of all numbers up to max
        r = torch.linspace(0,n.max(),n.max()+1).to(self.device)
        res = torch.zeros((*self.batch_size,n.max()+2)).to(self.device)
        t1 = (r+self.alpha).lgamma()
        t2 = -self.alpha.lgamma()-(r+1.0).lgamma()
        t3 = self.alpha*self.beta.log() - (r+self.alpha)*(1.0+self.beta).log()
        # evaluate cdf and take cumsum
        res[...,1:] = (t1+t2+t3).exp().cumsum(dim=-1).clamp(0.0,1.0)
        # return relevant values
        ind = n +1
        ind[n<0] = 0
        return res[...,ind]
    @property
    def mean(self):
        return self.alpha/self.beta

    @property
    def variance(self):
        return (self.alpha * (1.0 + self.beta)+1.0)/self.beta.pow(2)

class bivariateGammaPoisson(ExponentialFamily):
```

```
    """
    bivariate multinomial-dirichlet gamma-poisson distribution class
    """
    def __init__(self,alpha,beta,theta0,theta1,validate_args=None):
        # initialise parameters of distribution
        self.alpha,self.beta,self.theta0,self.theta1 = broadcast_all(alpha, beta, theta0, theta1)
        self.alpha,self.beta,self.theta0,self.theta1 = self.alpha.reshape(self.alpha.shape[0],1,1),
        ↪  self.beta.reshape(self.alpha.shape[0],1,1), self.theta0.reshape(self.alpha.shape[0],1,1),
        ↪  self.theta1.reshape(self.alpha.shape[0],1,1)
        self.thetaT = self.theta0 + self.theta1
        if isinstance(alpha, Number):
            batch_shape = torch.Size()
            self.device = torch.device("cpu")
            self.batch_size = torch.tensor([1])
        else:
            batch_shape = self.alpha.size()
            self.device = self.alpha.device
            self.batch_size = batch_shape[:2]
        super(bivariateGammaPoisson, self).__init__(batch_shape, validate_args=validate_args)
    def cdf(self,x,y):
        # returns cdf
        if self._validate_args:
            self._validate_sample(x)
            self._validate_sample(y)
        x,y = broadcast_all(x,y)
        # cast to integers
        nx = x.floor().long()
        ny = y.floor().long()

        # create grid of all points up to max
        rx = torch.linspace(0,nx.max(),nx.max()+1).to(self.device)
        ry = torch.linspace(0,ny.max(),ny.max()+1).to(self.device)
        rx,ry=torch.meshgrid([rx,ry])

        # calculate log pdf
        rx = rx.repeat(self.batch_size[0], 1).reshape(self.batch_size[0], *rx.shape)
        ry = ry.repeat(self.batch_size[0], 1).reshape(self.batch_size[0], *ry.shape)
        t1 = self.alpha*(self.beta.log()-(self.beta+1).log() ) + (self.alpha+rx+ry).lgamma()
        t2 = - self.alpha.lgamma() - (rx+ry+1).lgamma() - (rx+ry)*(self.beta+1).log()
        t3 = (rx+ry+1).lgamma() - (rx+1).lgamma()-(ry+1).lgamma() + self.thetaT.lgamma() +
        ↪  (rx+self.theta0).lgamma()+(ry+self.theta1).lgamma()
        t4 = -(self.thetaT+rx+ry).lgamma() - self.theta0.lgamma() - self.theta1.lgamma()
        res = torch.zeros(self.batch_size[0],nx.max()+2,ny.max()+2).to(self.device)
        # using cumsum get cdf
        res[:,1:,1:] = (t1+t2+t3+t4).exp().cumsum(-1).cumsum(-2)
        # get relevant indicies
        ind1 = nx +1
        ind1[nx<0] = 0
        ind2 = ny +1
        ind2[ny<0] = 0
        return res[:,ind1,ind2].clamp(0.0,1.0)
    def _cdf(self,x,y):
        # get cdf across bins
        if self._validate_args:
            self._validate_sample(x)
            self._validate_sample(y)
        # cast to integers
        x,y = broadcast_all(x,y)
        nx = x.floor().long()
        ny = y.floor().long()
        # get bin sizes
        sx = (x[1:] - x[:-1]).long()
        sy = (y[1:] - y[:-1]).long()

        # get indicies to sum over
        indx = torch.zeros(sx.sum()+1, dtype=torch.long).to(self.device)
        indx[torch.cumsum(sx, dim=0)] = torch.tensor(1).to(self.device)
        indx = torch.cumsum(indx, dim=0)[:-1]

        indy = torch.zeros(sy.sum()+1, dtype=torch.long).to(self.device)
        indy[torch.cumsum(sy, dim=0)] = torch.tensor(1).to(self.device)
        indy = torch.cumsum(indy, dim=0)[:-1]
        #create grid of points up to max
        rx = torch.linspace(0,nx.max(),nx.max()+1).to(self.device)
        ry = torch.linspace(0,ny.max(),ny.max()+1).to(self.device)
        rx,ry=torch.meshgrid([rx,ry])
        # calculate log pdf
        rx = rx.repeat(self.batch_size[0],1).reshape(self.batch_size[0],*rx.shape)
        ry = ry.repeat(self.batch_size[0],1).reshape(self.batch_size[0],*ry.shape)
        t1 = self.alpha*(self.beta.log()-(self.beta+1).log() ) + (self.alpha+rx+ry).lgamma()
        t2 = - self.alpha.lgamma() - (rx+ry+1).lgamma() - (rx+ry)*(self.beta+1).log()
        t3 = (rx+ry+1).lgamma() - (rx+1).lgamma()-(ry+1).lgamma() + self.thetaT.lgamma() +
        ↪  (rx+self.theta0).lgamma()+(ry+self.theta1).lgamma()
        t4 = -(self.thetaT+rx+ry).lgamma() - self.theta0.lgamma() - self.theta1.lgamma()
        temp = torch.zeros(self.batch_size[0],len(x)-1,ny.max()+1).to(self.device)
        # sum pdf over x for bins
        temp.index_add_(1,indx,(t1+t2+t3+t4).exp())
        res = torch.zeros(self.batch_size[0],len(x)-1,len(y)-1).to(self.device)
        # sum pdf over y for bins
        res.index_add_(2,indy,temp)#.cumsum(-1).cumsum(-2)

```

```
169
170             return res.clamp(0.0,1.0)
171     class poissonLogNormal(ExponentialFamily):
172         """
173         poisson log-normal distribution class
174         """
175         def __init__(self,mu,sigma,quad_points=8,validate_args=None):
176             # set parameters
177             self.mu,self.sigma = broadcast_all(mu,sigma)
178             self.quad_points = quad_points
179             if isinstance(mu, Number):
180                 batch_shape = torch.Size()
181                 self.device = torch.device("cpu")
182                 self.batch_size = torch.tensor([1])
183             else:
184                 batch_shape = self.mu.size()
185                 self.device = self.mu.device
186                 self.batch_size = batch_shape[0]
187             # get quadrature points
188             grid = self.quadrature()
189             # create poissons
190             self.poisson = Poisson(grid.exp())
191             # calculate logits
192             self.logits =-np.log(self.quad_points)
193
194
195             super(poissonLogNormal, self).__init__(batch_shape, validate_args=validate_args)
196         def quadrature(self):
197             # calculate quad points
198             edges = torch.linspace(0.,1., steps=self.quad_points+3).to(self.device)[1:-1] # dont use 0,1 as this may give inf
199             logNormal = LogNormal(self.mu,self.sigma)
200             # use icdf to get quad points
201             quantiles =logNormal.icdf(edges)
202             # take midpoints
203             grid = (quantiles[:,1:]+quantiles[:,:-1])/2
204             return grid
205         def log_prob(self,x):
206             # return log prob
207             if self._validate_args:
208                 self._validate_sample(x)
209             x, = broadcast_all(x)
210             return torch.logsumexp(self.logits + self.poisson.log_prob(x[:,None,None]),dim=-1)
211         def cdf(self,x):
212             if self._validate_args:
213                 self._validate_sample(x)
214             x, = broadcast_all(x)
215             # cast to integers
216             n = x.floor().long()
217             # create vector of all points up to maximum
218             r = torch.linspace(0,n.max(),n.max()+1).to(self.device)
219             # calculate cdf from log prob
220             res = self.log_prob(r).exp().t().cumsum(-1)
221             res[res.ne(res)]=0.0 # replace nans with 0
222             return res[:,n].clamp(0.0,1.0)
223
224     class bivariatePoissonLogNormal(ExponentialFamily):
225         """
226         bivariate poisson log-normal distribution class
227         (not working)
228         """
229         def __init__(self,mu,sigma,rho,quad_points=10,validate_args=None):
230             self.mu,self.sigma = broadcast_all(mu,sigma)
231             self.rho,= broadcast_all(rho)
232             self.quad_points= quad_points
233             if isinstance(mu, Number):
234                 batch_shape = torch.Size()
235                 self.device = torch.device("cpu")
236                 self.batch_size = torch.tensor([1])
237             else:
238                 batch_shape = self.mu.size()
239                 self.device = self.mu.device
240                 self.batch_size = batch_shape[0]
241             c = self.sigma.sqrt().prod(dim=-1)[:,None]*self.rho
242
243             self.covmatrix = torch.cat((self.sigma[:,0,None], c, c, self.sigma[:,1,None]),dim=-1).view(self.batch_size,2,2)
244             self.sigmaM = self.covmatrix.cholesky()
245             self.quadrature()
246
247             self.poissonx = Poisson(self.lam1)
248             self.poissony = Poisson(self.lam2)
249
250
251             super(bivariatePoissonLogNormal, self).__init__(batch_shape, validate_args=validate_args)
252         def quadrature(self):
253             p,w = np.polynomial.hermite.hermgauss(self.quad_points)
254             #p,w = torch.from_numpy(p).type(torch.float32).to(self.device),torch.from_numpy(w).type(torch.float32).to(self.device)
255             xn = torch.tensor([i for i in itertools.product(p, repeat=2)]).type(torch.float32).to(self.device)
256             self.wn =torch.tensor([i for i in itertools.product(w, repeat=2)]).type(torch.float32).to(self.device).prod(-1).log()
257             r = (torch.matmul(self.sigmaM,xn.t()).transpose(1,2)*np.sqrt(2) + self.mu.unsqueeze(1).repeat(1,100,1)).exp()
258             lr = torch.matmul(self.sigmaM,(1./r).transpose(1,2)).transpose(1,2)
259             self.lam1,self.lam2 = r[...,0], r[...,1]
```

```
260             self.llam1,self.llam2 = lr[...,0], lr[...,1]
261             self.const = self.sigmaM.det()/(np.pi)
262 
263             return
264         def prob(self,x,y):
265             if self._validate_args:
266                 self._validate_sample(x)
267                 self._validate_sample(y)
268             x,y = broadcast_all(x,y)
269 
270             return (self.const*(self.wn+(self.llam1*self.llam2).log() +
                ↪  self.poissonx.log_prob(x.unsqueeze(-1).unsqueeze(-1))+self.poissony.log_prob(y.unsqueeze(-1).unsqueeze(-1)) -
                ↪  (self.lam1*self.lam2).log()).exp().sum(-1)).reshape(self.batch_size,*x.shape)
271         def cdf(self,x,y):
272             if self._validate_args:
273                 self._validate_sample(x)
274             x,y = broadcast_all(x,y)
275             nx = x.floor().long()
276             ny = y.floor().long()
277             rx = torch.linspace(0,nx.max(),nx.max()+1).to(self.device)
278             ry = torch.linspace(0,ny.max(),ny.max()+1).to(self.device)
279             rx,ry=torch.meshgrid([rx,ry])
280             rx,ry=rx.flatten(),ry.flatten()
281 
282             res = torch.zeros((self.batch_size,nx.max()+2,ny.max()+2)).to(self.device)
283             res[:,1:,1:] = self.prob(rx,ry).view(self.batch_size,nx.max()+1,ny.max()+1).cumsum(-1).cumsum(-2)
284             return res
285     class MixD(model):
286         """
287         Mixture Model
288         """
289 
290         def __init__(self,T,num_mix,input_dim,bins,
291         dense_act=nn.ELU(),mix_dist="Gaussian",
292         p_act=(Exponential(),Exponential(),Exponential(),Exponential()),
293         learning_rate=1e-4,batch_size=1024):
294             super(model,self).__init__() # inherit
295             # set constants
296             self.use_cuda = torch.cuda.is_available()
297             self.device = torch.device(f"cuda" if self.use_cuda else "cpu")
298             self.batch_size = batch_size
299             self.T = T
300             self.input_dim = input_dim
301             self.p0_act = p_act[0]
302             self.p1_act = p_act[1]
303             self.mix_dist = mix_dist
304             self.bivariate = False
305             self.num_mix = num_mix
306             # create layers
307             self.p0 = nn.Linear(64,self.num_mix).to(self.device,non_blocking=self.use_cuda)
308             self.p1 = nn.Linear(64,self.num_mix).to(self.device,non_blocking=self.use_cuda)
309 
310             self.e_learning_rate = learning_rate
311             self.embedding1 = nn.Linear(self.input_dim,64).to(self.device,non_blocking=self.use_cuda)
312             self.embedding1Act = dense_act
313             self.embedding2 = nn.Linear(64,128).to(self.device,non_blocking=self.use_cuda)
314             self.embedding2Act = dense_act
315             self.embedding3 = nn.Linear(128,256).to(self.device,non_blocking=self.use_cuda)
316             self.embedding3Act = dense_act
317             self.embedding4 = nn.Linear(256,128).to(self.device,non_blocking=self.use_cuda)
318             self.embedding4Act = dense_act
319             self.embedding5 = nn.Linear(128,64).to(self.device,non_blocking=self.use_cuda)
320             self.embedding5Act = dense_act
321 
322 
323 
324 
325             self.ai = nn.Linear(64,self.num_mix)
326             self.aiAct = nn.Softmax(dim=1)
327             self.mix = torch.arange(0,self.num_mix,dtype=torch.int).to(self.device,non_blocking=self.use_cuda)
328             self.mix_dist = mix_dist
329             # set which function to use for getting predictions and set bins
330             if mix_dist == "Gaussian":
331                 self.distPredict = self.normalPredict
332                 self.bins = bins.to(self.device,non_blocking=self.use_cuda)
333             elif mix_dist == "GammaPoisson":
334                 self.distPredict = self.gammaPoissonPredict
335                 self.bins = (bins-1).to(self.device,non_blocking=self.use_cuda)
336             elif mix_dist == "PoissonLogNormal":
337                 self.distPredict = self.poissonLogNormalPredict
338                 self.bins = (bins-1).to(self.device,non_blocking=self.use_cuda)
339             elif mix_dist =="bivariateGammaPoisson":
340                 self.distPredict = self.biGammaPoissonPredict
341 
342                 self.binsx = (bins[0]-1).to(self.device,non_blocking=self.use_cuda)
343                 self.binsy = (bins[1]-1).to(self.device,non_blocking=self.use_cuda)
344                 # create additional layers required for this distribution
345                 self.p2 = nn.Linear(64,self.num_mix).to(self.device,non_blocking=self.use_cuda)
346                 self.p3 = nn.Linear(64,self.num_mix).to(self.device,non_blocking=self.use_cuda)
347                 self.p2_act = p_act[2]
348                 self.p3_act = p_act[3]
```

```
            self.bivariate = True
        else:
            raise ValueError("Invalid mixture distribution")

    def forward(self, x):
        # forward function
        # initial dense layers
        x = self.embedding1Act(self.embedding1(x))
        x = self.embedding2Act(self.embedding2(x))
        x = self.embedding3Act(self.embedding3(x))
        x = self.embedding4Act(self.embedding4(x))
        x = self.embedding5Act(self.embedding5(x))
        # get predictions of parameters
        if self.bivariate:
            p0 = self.p0_act(self.p0(x))
            p1 = self.p1_act(self.p1(x))
            p2 = self.p2_act(self.p2(x))
            p3 = self.p3_act(self.p3(x))
            p = (p0,p1,p2,p3)
        else:
            p0 = self.p0_act(self.p0(x))
            p1 = self.p1_act(self.p1(x))
            p = (p0,p1)
        # get weights
        pr = self.aiAct(self.ai(x))

        return p,pr
    def normalPredict(self,x):

        (m,s),p = x
        y = 0.0
        for i in self.mix: # for each dist in mixture evaluate cdf at bins
            dist = Normal(m[:,i,None],s[:,i,None])
            t = dist.cdf(self.bins)
            y += (t[:,1:]-t[:,:-1]).clamp(0)*p[:,i,None]
        return y
    def poissonLogNormalPredict(self,x):

        (m,s),p = x
        y = 0.0
        for i in self.mix:# for each dist in mixture evaluate cdf at bins
            dist = poissonLogNormal(m[:,i,None],s[:,i,None])
            t = dist.cdf(self.bins)
            y += (t[:,1:]-t[:,:-1]).clamp(0)*p[:,i,None]
        return y
    def gammaPoissonPredict(self,x):

        (alpha,beta),p = x
        y = 0.0
        for i in self.mix:# for each dist in mixture evaluate cdf at bins
            dist = gammaPoisson(alpha[:,i,None],beta[:,i,None])
            t = dist.cdf(self.bins)
            y += (t[:,1:]-t[:,:-1]).clamp(0)*p[:,i,None]
        return y
    def biGammaPoissonPredict(self,x):

        (alpha,beta,theta0,theta1),p = x
        y = 0.0
        for i in self.mix:# for each dist in mixture evaluate cdf at bins
            dist = bivariateGammaPoisson(alpha[:,i,None],beta[:,i,None],theta0[:,i,None],theta1[:,i,None])
            y += dist._cdf(self.binsx,self.binsy).clamp(0)*p[:,i,None,None]
        return y

    @torch.jit.export
    def train_model(self,epochs=100,optim="Adam",data="Gene",loss_fn="H2",custom_data=None,model_name=None):
        self._directory_maker() # create directory
        # set consts
        self.loss_fn = loss_fn
        self.epochs = epochs
        self.optim = optim

        if custom_data is None:
            # load data and split into train and test data
            xall,yall=self.load_data(data=data)
            x_train, x_test,y_train,y_test = train_test_split(xall,yall,train_size=0.8)
        else:
            # if custom data is given use that
            x_train, x_test,y_train,y_test =
            ↪  custom_data["x_train"],custom_data["x_test"],custom_data["y_train"],custom_data["y_test"]
            self.scalesM,self.scalesS = custom_data["scalesM"],custom_data["scalesS"]
        # set name of model
        if model_name == None:
            self.model_name = f"mix_{self.mix_dist}_{loss_fn}_{optim}_{self.num_mix}"
        else:
            self.model_name=model_name
        # create tensorboard writer
        writer = SummaryWriter(comment=self.model_name)
        # create optimiser
        self._init_optim()
        print("Loading dataset")
        # cast from numpy to tensor
```

```
        xtrainTensor=torch.from_numpy(x_train).type(torch.float32)
        ytrainTensor=torch.from_numpy(y_train).type(torch.float32)

        if self.mix_dist=="bivariateGammaPoisson":
            # use no additional workers for loading in the bivariate case
            # to stop the pytorch data loader loading too many batches and causing
            # a memory overflow
            num_workers = 0
        else:
            # for the univariate distribution use extra loaders as we dont
            # get a memory overflow
            num_workers = 2


        # create datat loaders
        dataset = torch.utils.data.TensorDataset(xtrainTensor,ytrainTensor)
        loader = DataLoader(dataset, batch_size=self.batch_size,drop_last=True,shuffle=True,
                    pin_memory=self.use_cuda,num_workers=num_workers)#self.use_cuda
        xtestTensor=torch.from_numpy(x_test).type(torch.float32)
        ytestTensor=torch.from_numpy(y_test).type(torch.float32)
        valDataset = torch.utils.data.TensorDataset(xtestTensor,ytestTensor)
        valLoader = DataLoader(valDataset, batch_size=self.batch_size,drop_last=True,shuffle=False,
                    pin_memory=self.use_cuda,num_workers=num_workers)#self.use_cuda
        old_test_loss = np.inf
        train_batches = x_train.shape[0] // self.batch_size
        test_batches =x_test.shape[0] // self.batch_size
        # intialise loss arrays
        self._init_loss()
        del valDataset,dataset,xtestTensor,ytestTensor,x_train,x_test,y_train,y_test
        # start train loop
        print("Train Loop Starting...")
        for i in range(self.epochs):
            #nvtx.range_push("Epoch " + str(i+1))
            self.train() # put in train mode
            print(f"STEP: {i+1}/{self.epochs}")
            with tqdm(total=train_batches,ascii=True,desc="Training") as pbar:#loading bar
                for idx, (local_batch, local_labels) in enumerate(loader):
                    #nvtx.range_push("Batch "+str(idx))

                    #nvtx.range_push("Copying to device")
                    # get batchees
                    y=local_labels.to(self.device,non_blocking=self.use_cuda)
                    x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                    #nvtx.range_pop()
                    #nvtx.range_push("Forward pass")
                    # forward pass
                    x=self(x_batch)
                    #nvtx.range_pop()

                    #nvtx.range_push("Backward pass")
                    # get output of mixture for bins and update loss and back prop
                    self._update_loss_train(self.distPredict(x),y,i)


                    #nvtx.range_pop()
                    #nvtx.range_pop()



                    pbar.update(1)

            #nvtx.range_pop()
            # average
            self.train_loss[i] = self.train_loss[i]/train_batches
            if np.isnan(self.train_loss[i,0]):# break if we get nan
                print("GOT NaN")
                break
            # update tensorboard log and print loss
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/train'
                writer.add_scalar(name, float(self.train_loss[i,l]), i)
                print(f"{name}: {self.train_loss[i,l]}")
            # get val loss
            with torch.no_grad(): # dont keep gradients
                self.eval()# set to eval
                with tqdm(total=test_batches,ascii=True,desc="Validating") as pbar:
                    for local_batch, local_labels in valLoader:
                        # get batch
                        y = local_labels.to(self.device,non_blocking=self.use_cuda)
                        x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                        # forward pass
                        pred = self(x_batch)
                        # update val loss
                        self._update_loss_val(self.distPredict(pred),y,i)
                        pbar.update(1)
            #nvtx.range_pop()
            # average
            self.val_loss[i] = self.val_loss[i]/test_batches
            # update tensorboard print val loss
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/test'
```

```
                writer.add_scalar(name, float(self.val_loss[i,l]), i)
                print(f"{name}: {self.val_loss[i,l]}")
            #nvtx.range_pop()
            # if achieves lowest val loss save
            if self.val_loss[i,0] < old_test_loss:
                #nvtx.mark("Saving best model")
                torch.save({"state_dict":self.state_dict(),
                            "scalesM":self.scalesM,
                            "scalesS":self.scalesS,
                            "epoch":i+1,
                            "optim":self.optimiser,
                            "val_loss":self.val_loss,
                            "T": self.T,
                            "input_dim": self.input_dim}, f"./best/mdBestModel_{self.model_name}.pth")
                old_test_loss = self.val_loss[i,0]
        # save final model
        torch.save({"state_dict":self.state_dict(),
                    "scalesM":self.scalesM,
                    "scalesS":self.scalesS,
                    "epoch":self.epochs,
                    "optim":self.optimiser,
                    "val_loss":self.val_loss,
                    "T": self.T,
                    "input_dim": self.input_dim}, f"./final/mdFinalModel_{self.model_name}.pth")
        return self.train_loss,self.val_loss
    @torch.jit.export
    def predict(self,x,raw=True):
        # get predictions
        self.eval() # set to eval
        # scale x
        x = (np.atleast_2d(x) - self.scalesM)/self.scalesS
        p = int(np.ceil(x.shape[0] /self.batch_size))
        t = x.shape[0]
        r = self.batch_size - (t % self.batch_size)
        x = np.pad(x,((0,r),(0,0)))
        xs = np.vsplit(x,p)
        res = [None] * p
        if raw: # return parameters
            for i in range(p):# feed x in batch size chunks
                # forward
                a,p = self(torch.from_numpy(xs[i]).type(torch.float32).to(self.device,non_blocking=self.use_cuda))
                # detach and send to numpy
                p = p.detach().cpu().numpy()
                a = torch.cat(a,dim=-1).detach().cpu().numpy()
                res[i] = np.hstack((p,a))

        else:# return bin probs
            for i in range(p):# feed x in batch size chunks
                # forward
                x = self(torch.from_numpy(xs[i]).type(torch.float32).to(self.device,non_blocking=self.use_cuda))
                # get dist predictions
                x = self.distPredict(x)

                res[i] = x.detach().cpu().numpy()
        res[-1] = res[-1][:-r]

        return np.vstack(tuple(res))
    @torch.jit.export
    def load_model(self,PATH):
        # load model
        model = torch.load(PATH,map_location=self.device)
        self.scalesM = model["scalesM"]
        self.scalesS = model["scalesS"]

        self.load_state_dict(model["state_dict"])




if __name__=="__main__":
    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")
    num_mix = [2**i for i in range(7)]
    loss = ["MSE","L1","H2","MDDKLD","MDDEDIV"]
    optim = ["Adam","SGD"]
    gau = torch.linspace(-15,15,100+1)
    real = torch.linspace(0,300,100+1)
    g = ParameterGrid({"loss":loss,"optim":optim})

    val_loss = [[None,i] for i in g]
    train_loss = [[None,i] for i in g]
    for v,i in enumerate(g):
        try:
            print(i)
            m = MixD(T=100,num_mix=6,input_dim=9,bins=gau,mix_dist="Gaussian").to(device)

            train_loss[v][0],val_loss[v][0] = m.train_model(epochs=200,data="Gaussian",loss_fn=i["loss"],optim=i["optim"])

        except Exception as inst:
```

```
            print("ERROR!!!!")
            print(type(inst))
            print(inst)
    with open('train_loss.pickle', 'wb') as f:
        pickle.dump(train_loss, f, protocol=pickle.HIGHEST_PROTOCOL)
    with open('val_loss.pickle', 'wb') as f:
        pickle.dump(val_loss, f, protocol=pickle.HIGHEST_PROTOCOL)


```

XII.C.2 Ensemble Mixture Model

```
from MIXD import *

class ensembleMIX(MixD):
    """
    model to make ensemble of n mixture models
    """
    def __init__(self,n,T,bins,num_mix,
    mix_dist="Gaussian",batch_size=1024,
    p_act=(Exponential(),Exponential(),Exponential(),Exponential()),
    dense_act=nn.ELU(),learning_rate=1e-5):
        super(model,self).__init__()#inherit
        # set consts
        self.use_cuda = torch.cuda.is_available()
        self.device = torch.device("cuda" if self.use_cuda else "cpu")
        self.n = n
        self.T = T
        self.bini = bins
        self.batch_size=batch_size
        self.e_learning_rate=learning_rate
        self.dense_act=dense_act
        self.num_mix = num_mix
        const = 3 if mix_dist != "bivariateGammaPoisson" else 5
        self.input_dim = const*self.n*self.num_mix
        # create layers
        self.embedding1 = nn.Linear(self.input_dim, self.input_dim*2).to(self.device,non_blocking=self.use_cuda)
        self.embedding1Act = dense_act
        self.embedding2 = nn.Linear(self.input_dim*2, self.input_dim*4).to(self.device,non_blocking=self.use_cuda)
        self.embedding2Act = dense_act
        self.embedding3 = nn.Linear(self.input_dim*4, self.input_dim*8).to(self.device,non_blocking=self.use_cuda)
        self.embedding3Act = dense_act
        self.embedding4 = nn.Linear(self.input_dim*8, self.input_dim*4).to(self.device,non_blocking=self.use_cuda)
        self.embedding4Act = dense_act
        self.embedding5 = nn.Linear(self.input_dim*4, self.input_dim).to(self.device,non_blocking=self.use_cuda)
        self.embedding5Act = dense_act
        self.p0_act = p_act[0]
        self.p1_act = p_act[1]
        self.bivariate = False


        self.p0 = nn.Linear(self.input_dim,self.num_mix).to(self.device,non_blocking=self.use_cuda)
        self.p1 = nn.Linear(self.input_dim,self.num_mix).to(self.device,non_blocking=self.use_cuda)

        self.ai = nn.Linear(self.input_dim,self.num_mix)
        self.aiAct = nn.Softmax(dim=1)
        self.mix = torch.arange(0,self.num_mix,dtype=torch.int).to(self.device,non_blocking=self.use_cuda)
        self.mix_dist = mix_dist
        # set mixture distribution and bins
        if mix_dist == "Gaussian":
            self.distPredict = self.normalPredict
            self.bins = bins.to(self.device,non_blocking=self.use_cuda)
        elif mix_dist == "GammaPoisson":
            self.distPredict = self.gammaPoissonPredict
            self.bins = (bins-1).to(self.device,non_blocking=self.use_cuda)
        elif mix_dist == "PoissonLogNormal":
            self.distPredict = self.poissonLogNormalPredict
            self.bins = (bins-1).to(self.device,non_blocking=self.use_cuda)
        elif mix_dist =="bivariateGammaPoisson":
            self.distPredict = self.biGammaPoissonPredict
            self.binsx = (bins[0]-1).to(self.device,non_blocking=self.use_cuda)
            self.binsy = (bins[1]-1).to(self.device,non_blocking=self.use_cuda)
            # create additional layers required by this distribution
            self.p2 = nn.Linear(self.input_dim,self.num_mix).to(self.device,non_blocking=self.use_cuda)
            self.p3 = nn.Linear(self.input_dim,self.num_mix).to(self.device,non_blocking=self.use_cuda)
            self.p2_act = p_act[2]
            self.p3_act = p_act[3]
            self.bivariate = True
        else:
            raise ValueError("Invalid mixture distribution")


    def forward(self,x):
        # forward method
        # initial dense layers
        x = self.embedding1Act(self.embedding1(x))
        x = self.embedding2Act(self.embedding2(x))
        x = self.embedding3Act(self.embedding3(x))
```

```
        x = self.embedding4Act(self.embedding4(x))
        x = self.embedding5Act(self.embedding5(x))
        # predict parameters
        if self.bivariate:
            p0 = self.p0_act(self.p0(x))
            p1 = self.p1_act(self.p1(x))
            p2 = self.p2_act(self.p2(x))
            p3 = self.p3_act(self.p3(x))
            p = (p0,p1,p2,p3)
        else:
            p0 = self.p0_act(self.p0(x))
            p1 = self.p1_act(self.p1(x))
            p = (p0,p1)
        # predict weights
        pr = self.aiAct(self.ai(x))

        return p,pr
    @torch.jit.export
    def train_model(self,epochs=100,optim="Adam",data="gau",loss_fn="H2"):
        # create directory to store models
        self._directory_maker()
        # set consts
        self.loss_fn = loss_fn
        self.epochs = epochs
        self.optim = optim
        # get data
        self.create_data(data)
        self.model_name = f"mix_{self.mix_dist}_{loss_fn}_{optim}_{self.num_mix}"
        # set tensorboard writer
        writer = SummaryWriter(comment=self.model_name)
        # initialise optimiser
        self._init_optim()
        print("Loading dataset")

        # set number of workers for data loaders
        if self.mix_dist=="bivariateGammaPoisson":
            num_workers = 0
        else:
            num_workers = 2


        # create data loaders
        xtrainTensor=torch.from_numpy(self.x_train).type(torch.float32)
        ytrainTensor=torch.from_numpy(self.y_train).type(torch.float32)
        dataset = torch.utils.data.TensorDataset(xtrainTensor,ytrainTensor)
        loader = DataLoader(dataset, batch_size=self.batch_size,drop_last=True,shuffle=True,
                    pin_memory=self.use_cuda,num_workers=num_workers)
        xtestTensor=torch.from_numpy(self.x_test).type(torch.float32)
        ytestTensor=torch.from_numpy(self.y_test).type(torch.float32)
        valDataset = torch.utils.data.TensorDataset(xtestTensor,ytestTensor)
        valLoader = DataLoader(valDataset, batch_size=self.batch_size,drop_last=True,shuffle=False,
                    pin_memory=self.use_cuda,num_workers=num_workers)
        # create array to store loss
        self._init_loss()
        old_test_loss = np.inf
        train_batches = self.x_train.shape[0] // self.batch_size
        test_batches =self.x_test.shape[0] // self.batch_size
        del valDataset,dataset,xtestTensor,ytestTensor,xtrainTensor,ytrainTensor
        # start train loop
        print("Train Loop Starting...")
        for i in range(self.epochs):
            #nvtx.range_push("Epoch " + str(i+1))
            self.train()# set train mode
            print(f"STEP: {i+1}/{self.epochs}")
            with tqdm(total=train_batches,ascii=True,desc="Training") as pbar:# loading bar
                for idx, (local_batch, local_labels) in enumerate(loader):
                    #nvtx.range_push("Batch "+str(idx))

                    #nvtx.range_push("Copying to device")
                    # load batch
                    y=local_labels.to(self.device,non_blocking=self.use_cuda)
                    x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                    #nvtx.range_pop()
                    #nvtx.range_push("Forward pass")
                    # forward pass
                    x=self(x_batch)
                    #nvtx.range_pop()

                    #nvtx.range_push("Backward pass")
                    #evalate mixture and update loss and back prop
                    self._update_loss_train(self.distPredict(x),y,i)


                    #nvtx.range_pop()
                    #nvtx.range_pop()



                    pbar.update(1)

            #nvtx.range_pop()
```

```
            # average
            self.train_loss[i] = self.train_loss[i]/train_batches
            if np.isnan(self.train_loss[i,0]):# if nan break
                print("GOT NaN")
                break
            # write tensorboard log and print loss
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/train'
                writer.add_scalar(name, float(self.train_loss[i,l]), i)
                print(f"{name}: {self.train_loss[i,l]}")
            # get val loss
            with torch.no_grad():
                self.eval()# set eval mode
                with tqdm(total=test_batches,ascii=True,desc="Validating") as pbar:# loading bar
                    for local_batch, local_labels in valLoader:
                        # load batch
                        y = local_labels.to(self.device,non_blocking=self.use_cuda)
                        x_batch=local_batch.to(self.device,non_blocking=self.use_cuda)
                        # forward pass
                        pred = self(x_batch)
                        # evaluate mixture and get loss
                        self._update_loss_val(self.distPredict(pred),y,i)
                        pbar.update(1)
            #nvtx.range_pop()
            # average
            self.val_loss[i] = self.val_loss[i]/test_batches
            # update tensorbard and print loo
            for l,s in enumerate(self.loss_names):
                name=f'Loss {s}/test'
                writer.add_scalar(name, float(self.val_loss[i,l]), i)
                print(f"{name}: {self.val_loss[i,l]}")
            #nvtx.range_pop()
            # if it has the lowest val loss print
            if self.val_loss[i,0] < old_test_loss:
                #nvtx.mark("Saving best model")
                torch.save({"state_dict":self.state_dict(),
                            "scalesM":self.scalesM,
                            "scalesS":self.scalesS,
                            "epoch":i+1,
                            "optim":self.optimiser,
                            "val_loss":self.val_loss,
                            "T": self.T,
                            "input_dim": self.input_dim}, f"./best/mdEnsBestModel_{self.model_name}.pth")
                old_test_loss = self.val_loss[i,0]
        # save final model
        torch.save({"state_dict":self.state_dict(),
                    "scalesM":self.scalesM,
                    "scalesS":self.scalesS,
                    "epoch":self.epochs,
                    "optim":self.optimiser,
                    "val_loss":self.val_loss,
                    "T": self.T,
                    "input_dim": self.input_dim}, f"./final/mdRnsFinalModel_{self.model_name}.pth")
        return self.train_loss,self.val_loss
    def _load_model(self,i):
        # load sub model and put in eval mode
        m =
        ↪  MixD(T=self.T,input_dim=self.inp,num_mix=self.num_mix,bins=self.bini,mix_dist=self.mix_dist,dense_act=nn.Tanh()).to(self.device)
        m.load_model(f"mdBestModel_ensemble_model_{self.data}_{i+1}.pth")
        m.eval()

        return m

    def _submodel_predict(self,x,n):
        # get prediction of n submodels in series
        out = [None] * n
        for i in range(n):
            out[i] = self._load_model(i).predict(x)
        return np.hstack(tuple(out))
    def create_data(self,data):
        # set const
        self.data = data
        if data == "gau":
            self.inp = 9
        else:
            self.inp = 5
        # load data
        with h5py.File(f"ensemble_data_{data}.hdf5","r") as f:
            x_train = f["x_train"][:]
            x_test = f["x_test"][:]
            y_train = f["y_train"][:]
            y_test = f["y_test"][:]

        self.x_train,self.x_test,self.y_train,self.y_test
        ↪  =self._submodel_predict(x_train,self.n),self._submodel_predict(x_test,self.n),y_train,y_test
        xall = np.vstack((self.x_train,self.x_test))
        self.scalesM = np.mean(xall,0)
        self.scalesS = np.std(xall,0)
        self.x_train,self.x_test = (self.x_train-self.scalesM)/self.scalesS,(self.x_test-self.scalesM)/self.scalesS


```

This is my own work except where otherwise stated