A GENERALIZED STACKING METHOD USING MATRIX ENSEMBLE KALMAN FILTER-BASED MULTI-ARM NEURAL NETWORK

by

Ved Piyush

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Statistics

Under the Supervision of Professor Souparno Ghosh

Lincoln, Nebraska

Dec, 2023

A GENERALIZED STACKING METHOD USING MATRIX ENSEMBLE KALMAN FILTER-BASED MULTI-ARM NEURAL NETWORK

Ved Piyush, Ph.D.

University of Nebraska, 2023

Adviser: Souparno Ghosh

Deep learners (DLs) have turned out to be the state-of-the-art method for predictive inference. Since we do not have widely applicable generalization error bounds for DLs, we can prevent over-confident inferences and predictions from a single "best performing" DL by creating an ensemble of such models and then performing model averaging. Such model averaging has been shown to increase robustness in the realm of deep learning techniques [74]. This increase in robustness could be partially attributed to the fact that with model averaging, we no longer ignore the uncertainty due to model choice. Stacking is one of the most popular model-averaging protocols. In its standard form, stacking uses the output of base learners as non-stochastic inputs to a meta-learner. However, that ignores the uncertainty in the predictions generated by these base DLs. This practice is problematic because DLs are often trained with dropout layers, which induce uncertainty in their predictions. Consequently, a meta-learner should process that uncertainty hardwired into the base models.

In this dissertation, we derive a novel methodology that can perform model averaging and propagate the uncertainty associated with the base models more coherently. We utilize Matrix Ensemble Kalman Filters to design a multi-arm artificial neural network that drives stochastic weights and performs model averaging in every filter update and batch update step. By default, our method produces realizations from one-step ahead predictive distribution, enabling the construction of prediction

intervals from averaged models. We demonstrate that our methodology can be utilized for transfer learning and potentially identify a specific form of mean non-stationarity in the underlying data-generating model. We apply our model to cancer drug response predictions and classification of gut microbiota. All codes used in this dissertation can be obtained from: https://github.com/Ved-Piyush/UNL_Thesis_Codes_VP/tree/main.

DEDICATION

To my parents Manju Bala Sahoo, Alekh Kumar Sahu, and my sister Ved Pragyan.

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Souparno Ghosh, wholeheartedly. His unwavering support, patience, and expert guidance were a cornerstone throughout this research journey. His insightful feedback and advice played a pivotal role in shaping the direction of this dissertation.

I also want to express my gratitude to my committee members: Dr. Bertrand Clarke, Dr. Xueheng Shi, and Dr. Mohammad Rashedul Hasan. Their feedback, time, and valuable suggestions were immensely helpful.

A big shoutout goes to Dr. Yanbin Yin for giving me the opportunity to work as a Graduate Research Assistant. Additionally, I'm thankful for the support and mentorship of Dr. Yuzhen Zhou.

I can't go without mentioning the incredible professors at Sri Venkateswara College - Delhi University, the University of Minnesota - Twin Cities, and the University of Nebraska - Lincoln. They've left an indelible mark on my academic journey.

Lastly, a special thanks to my friends and peers. The memories we've created are an integral part of this Ph.D. adventure. Your companionship made this journey not just possible but enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consider a set of M base learners trained on the same dataset $\boldsymbol{Z} = (\boldsymbol{X}, Y)$ where $\boldsymbol{X} = (x_1, x_2, ..., x_p)$ is the set of predictors and the target variable is Y. Let $\hat{f}_1(\boldsymbol{x}), \hat{f}_2(\boldsymbol{x}), ...,$-$\hat{f}_M(\boldsymbol{x})$ be the set of M predictions obtained from these models. If the population distribution of the dataset is assumed to be known, i.e., $\boldsymbol{Z} \sim \mathcal{P}$, then the population level stacking is obtained as a weighted average of the predictions generated by M models, with the weights being

$$\hat{w} = \underset{w}{argmin}\ E_{\mathcal{P}} \left[Y - \sum_{m=1}^{M} w_m \hat{f}_m(x) \right]^2 \tag{1.1}$$

Since, $\hat{w}$ is the minimizer of the expression in the RHS of (1.1), it is easy to see

$$E_{\mathcal{P}} \left[Y - \sum_{m=1}^{M} \hat{w}_m \hat{f}_m(\boldsymbol{x}) \right]^2 \leq E_{\mathcal{P}} \left[Y - \hat{f}_m(\boldsymbol{x}) \right]^2 \forall m = 1, 2, .., M \tag{1.2}$$

indicating that if $\mathcal{P}$ is known, stacking never performs worse as compared to any single model.

Since $\mathcal{P}$ is typically unknown, the stacking estimates are extracted either via

cross-validation or over a validation dataset, i.e.,

$$\hat{w}^{(st)} = \underset{w}{argmin} \sum_{i=1}^{N} \left[y_i - \sum_{m=1}^{M} w_m \hat{f}_m^{-i}(\boldsymbol{x}_i) \right]^2$$

where $\hat{f}_m^{-i}(\boldsymbol{x}_i)$ is the prediction obtained from the m^{th} model with the i^{th} training instance removed.

Customarily, the stacking operation is performed using the point prediction yielded by each base learner as input. That is, the *cross learner weights* w_m's are estimated given the predictions from the base learners. Consequently, standard stacking produces optimal (under square error loss) *cross-learner weights* but does not address the joint optimality of *within-learner* and *cross-learner* weights. This question becomes important when we stack multiple deep learners to generate predictions. How do we attach uncertainty to the predictions obtained from conventionally stacked deep learners? We can use Monte Carlo dropout to attach uncertainty to the predictions generated by each base learner. But to obtain the stacked prediction, we need to *fix* the output from each base learner, and hence the uncertainty does not propagate coherently.

Our main methodological contribution in this dissertation is to develop a statistical framework that attaches uncertainty to the predictions obtained from a convex combination of base learners in a coherent way. We view this problem as a simultaneous estimation of *within-learner* and *cross-learner* weights. Therefore, we aim to optimize a suitably crafted square error loss function that yields optimal estimates of both *within-learner* and *cross-learner* weights simultaneously. Therefore, the entire set of weights can be viewed as *jointly optimal*. Observe that, while conventional stacking takes a sequential approach to estimate the whole set of model weights -

first, the *within-learner* weights are estimated followed by *cross-learner* weights, our approach estimates the complete set of weights simultaneously under a global loss criterion. We therefore view the proposed method as a *generalized stacking* approach. Although, as we shall see in subsequent chapters, our approach does not strictly adhere to the mathematical definition of stacking as specified in (1.1), it is similar in spirit to the concept of stacking in the sense that both approaches strive to arrive at a linear combination of base learners that improves the prediction performance as compared to any single constituent learner.

We propose an augmented state-space model for training artificial neural networks (ANNs) to achieve the foregoing goal. At the outset, we specify that our focal predictive vehicle will be a convex combination of these ANNs. Therefore, instead of training each constituent ANN separately, we train all of them simultaneously under a global square error loss function. The key novelty of this methodology is that with the augmented state-space formulation and simultaneous estimation strategy, the uncertainty associated with the predictions generated by the base models can be propagated more coherently to the model-averaged prediction. The main contributions of our proposed methodology are the following:

1. It offers a general procedure to attach uncertainty to model-averaged predictions, particularly when the constituent models are essentially algorithmic (neural networks, more precisely).

2. It offers two additional metrics, coverage probability and width of prediction intervals, to assess the statistical adequacy of the posited model - which we argue is essential because the base learners are essentially input/output algorithms.

3. It offers a way to perform transfer learning in a situation where a complex deep learner is trained on a large dataset in the source domain, but the dataset in the

target domain is small and only has a subset of features as compared to the set of features available in the source domain. Essentially, our methodology offers a way to transfer information from the source domain when the target domain requires training a *reduced* model.

4. It can detect a switch in the data-generating model under the assumption that the true data-generating models appear in the set of models that are being averaged and the switch in the data-generating model manifests through a specific type of mean non-stationarity.

We will demonstrate that by reconceptualizing the conventional stacking approach, we can achieve all the foregoing aspects under a single generic framework. Our model is deployed on observational cross-sectional data and requires numerical response variables and numerically encoded input features. We also show that if at least one of the base learners appearing in the *stack of the models* is the true data generating model, then the points estimates of the *within-learner* and *cross-learner* weights produced by our framework are jointly optimal under mean square error loss.

We explicate the foregoing facets of the proposed methodology in subsequent chapters of our dissertation. In Chapter 2, we offer a background on the techniques and concepts heavily used to develop and illustrate the proposed *generalized stacking* approach. We briefly discuss relevant literature and offer toy examples of the standard state-space model. In Chapter 3, we lay out the methodology of our *generalized stacking* procedure, discuss its theoretical aspects, and offer two illustrative applications of our methodology. In the first illustration, we demonstrate how our method could be utilized to transfer knowledge gleaned from a multi-arm deep learner (DeepCDR - developed to predict cancer drug responses using drug features and multi-omics data[44]) to perform predictions on query points arriving with a reduced set of features.

In the second illustration, we demonstrate how our approach can be utilized to attach uncertainty to the outputs generated by an averaging of two LSTM classifiers. We perform simulations to demonstrate that our approach can recover true model weights given that the true model appears in the *stack of base models*. We also show how coverage probability and average width of prediction intervals could be used to identify situations where true data generating model(s) do not appear in the *stack of base models*.

In Chapter 4, we examine the scalability of our approach. We show that using a modified training algorithm, our model can be trained on large datasets to perform model averaging of multiple multi-arm deep learners. We also demonstrate how our method offers a standardized approach to compare the prediction performance of two competing deep learners.

In Chapter 5, we extend our approach to accommodate multivariate responses. We offer simulation studies to demonstrate how our conceptualization of model averaging can identify changes in the data-generating model. We illustrate the predictive performance of the multivariate version of our approach, we deploy this technique to predict two important chemical properties of drugs from the chemical structures of the drug molecules.

The final chapter offers concluding remarks and potential future research directions.

Chapter 2

Literature Review

In this section, we offer a brief discussion of the concepts used in developing our proposed generalized stacking methodology. Our main conceptual vehicle is a matrix variate Ensemble Kalman Filter (EnKF). We, therefore, begin by describing the standard Kalman Filter (KF) framework and offer a synthetic example to illustrate how it works. We then discuss the EnKF formulation that approximates standard KF and provides a computationally inexpensive way to estimate the Kalman Gain. We then discuss the matrix state space model as a generalization of standard KF. Next, we pivot to some deep learning concepts used in this dissertation. We offer discussion on embeddings and Monte Carlo dropout, which are concepts that are routinely used in deep learning models. We end this chapter with a description of the Long Short Term Memory (LSTM) models and Graph Convolutional Network models, that are two deep learning models we heavily use to illustrate the application of our model averaging technique. In each section, we direct the audience to relevant literature that offers a more in-depth description of the concepts.

2.1 Linear State Space Model

A linear Gaussian state-space model can be described as

$$y_t = H_t x_t + v_t, \quad v_t \sim \mathcal{N}_{m_t}(0, R_t) \tag{2.1}$$

$$x_t = M_t x_{t-1} + w_t, \quad w_t \sim \mathcal{N}_n(0, Q_t) \tag{2.2}$$

Here, the subscript t is for the discrete-time point, and $y_t \in \mathbb{R}^{m_t}$ is the m_t dimensional observation vector at time step t. $x_t \in \mathbb{R}^n$ is the n dimensional (unknown) state vector at time t. H_t and M_t denote the observation and the state transition matrixes. H_t describes how the state variable x_t relates to the observation and M_t denotes how the state vector at time $t-1$ is related to the state vector at time t. $v_t \in \mathbb{R}^{m_t}$ is the measurement noise and $w_t \in \mathbb{R}^n$ is the process noise.

Statistical assumptions:

1. $x_0, w_0, w_1, ..., v_0, v_1, ...$ are jointly Gaussian and independent.
2. $E(w_t) = E(v_t) = 0, E(w_t w_t') = Q_t, E(v_t v_t') = R_t$.
3. For the moment we assume M_t, H_t, R_t, and Q_t are known.
4. v_t's are independent of x_t's.
5. w_t is independent of $x_0, x_1, ..., x_t$ and $y_0, y_1, ..., y_t$.
6. The state is first order Markovian, i.e., $[x_t|x_0, x_1, ..., x_{t-1}] = [x_t|x_{t-1}]$, where $[x_t|.]$ denotes the conditional distribution of x_t given the remaining arguments.

Given the above formulation, the goal is to estimate $\mu_{t|s} = E(x_t|y_0, y_1, ..., y_s)$, and $\Sigma_{t|s} = E[(x_t - E(x_t|y_0, y_1, ..., y_s))(x_t - E(x_t|y_0, y_1, ..., y_s))']$, for any pair of t and s, with $s \leq t$. That is, we would like to obtain an estimate of the expected state of the

system at time t based on $y_0, y_1, ..., y_s$ and estimate the covariance matrix associated with the error in the state estimation. The Kalman Filter offers a recursive way to solve the above-stated estimation problem. See [33] for a more detailed review.

2.2 Kalman Filter

Let $y_{1:t} := \{y_1, y_2, ..., y_t\}$ denote the set of observation vectors until time t. Similarly, $x_{1:t} := \{x_1, x_2, ..., x_t\}$ denote the collection of state vectors until t and x_0 is the initial state of the system.

KF mainly focuses on (i) estimating the current state given the current observation and past observed outputs, i.e., estimate $\mu_{t|t}$, and (ii) predicting the next state based on the current and past observed values of $y_{1:t}$, i.e., estimate $\mu_{t+1|t}$.

We begin with the assumption that the conditional distribution for the state variable at time step $t-1$ conditional on $y_{1:t-1}$ is given by

$$x_{t-1}|y_{1:t-1} \sim \mathcal{N}(\mu_{t-1}, \Sigma_{t-1}) \tag{2.3}$$

The above distribution is conventionally called the filtering distribution of x_t. Now, we can use the state transitional equation (2.2) to obtain the conditional distribution $x_t|y_{1:t-1}$. This conditional distribution is conventionally referred to as the forecast distribution at time t and is given by

$$x_t|y_{1:t-1} \sim \mathcal{N}(\tilde{\mu}_t, \tilde{\Sigma}_t) \tag{2.4}$$

$$\tilde{\mu}_t = \mu_{t|t-1} := M_t \mu_{t-1} \tag{2.5}$$

$$\tilde{\Sigma}_t = \Sigma_{t|t-1} := M_t \Sigma_{t-1} M_t^{'} + Q_t \tag{2.6}$$

Once the measurement data arrives at time $t(y_t)$, the Kalman Filter updates the

filtering distribution by first computing the joint distribution of x_t and y_t conditional on measurement data till time step $t-1$, i.e., $y_{1:t-1}$. That is KF finds $\mu_{t|t}$ and $\Sigma_{t|t}$ in terms of $\mu_{t|t-1}$ and $\Sigma_{t|t-1}$. Observe, $y_t|y_{1:t-1} = H_t x_t|y_{1:t-1} + v_t$ and $x_t|y_{1:t-1}$ is given in (2.4). Then the joint distribution of $x_t|y_{1:t-1}, y_t|y_{1:t-1}$ becomes

$$\begin{pmatrix} x_t \\ y_t \end{pmatrix} \Bigg| y_{1:t-1} \sim N\left(\begin{pmatrix} \tilde{\mu}_t \\ H_t\tilde{\mu}_t \end{pmatrix}, \begin{pmatrix} \tilde{\Sigma}_t & \tilde{\Sigma}_t H_t' \\ H_t\tilde{\Sigma}_t & H_t\tilde{\Sigma}_t H_t' + R_t \end{pmatrix} \right) \tag{2.7}$$

Then using (2.7) the updated filtering distribution is computed as $x_t|y_{1:t} \sim \mathcal{N}(\hat{\mu}_t, \hat{\Sigma}_t)$ where $\hat{\mu}_t$ and $\hat{\Sigma}_t$ are given below

$$\hat{\mu}_t = \mu_{t|t} := \tilde{\mu}_t + K_t(y_t - H_t\tilde{\mu}_t) \tag{2.8}$$

$$\hat{\Sigma}_t = \Sigma_{t|t} := (I_n - K_t H_t)\tilde{\Sigma}_t \tag{2.9}$$

$$K_t := \tilde{\Sigma}_t H_t'(H_t\tilde{\Sigma}_t H_t' + R_t)^{-1} \tag{2.10}$$

K_t is the Kalman Gain Matrix.

Now, we would like to perform a time update to predict the next state based on $y_{1:t}$, i.e., obtain $\mu_{t+1|t}$. Since $x_{t+1} = M_{t+1}x_t + w_t$, we can condition on $y_{1:t}$ and express $x_{t+1}|y_{1:t}$ as $M_{t+1}x_t|y_{1:t} + w_t$ (since w_t is independent of $y_{1:t}$ by assumption). Therefore $\mu_{t+1|t} = E(x_{t+1}|y_{1:t}) = M_{t+1}\mu_{t|t} = M_{t+1}\hat{\mu}_t$, and $\Sigma_{t+1|t} = E(x_{t+1} - \mu_{t+1|t})(x_{t+1} - \mu_{t+1|t})' = M_{t+1}\Sigma_{t|t}M_{t+1}' + Q_t$. We can interpret $\mu_{t+1|t}$ in the following fashion. Substituting the expression for $\hat{\mu}_t$ in the expression for $\mu_{t+1|t}$ yields

$$\mu_{t+1|t} = M_{t+1}\tilde{\mu}_t + M_{t+1}K_t(y_t - H_t\tilde{\mu}_t)$$

Notice $H_t\tilde{\mu}_t = E(y_t|y_{1:t-1})$, therefore $(y_t - H_t\tilde{\mu}_t)$ is the output prediction error.

Consequently, the predicted state of the system at $t+1$ is the prediction based on $y_{1:t-1}$ (as captured in $\tilde{\mu}_t$) added to a linear function of the prediction error in the observation (or measurement) model.

Now, since we have observations $y_{1:t}$, we explore how KF yields an optimal estimate of $E(x_t|y_{1:t})$. That is we would like to find an estimator of $E(x_t|y_{1:t})$ that minimizes $E(\|x_t - E(x_t|y_{1:t})\|^2)$ with respect to $E(x_t|y_{1:t})$. The normality of the updated filtering distribution (2.8) - (2.10) yields $\hat{\mu}_t$ to be the minimizer of $E(\|x_t - E(x_t|y_{1:t})\|^2)$. To interpret the Kalman Gain, we note that, instead of directly minimizing $E(\|x_t - E(x_t|y_{1:t})\|^2)$, an equivalent problem is to obtain the Kalman Gain matrix that minimizes the trace norm of the updated covariance matrix in the filtering distribution $\hat{\Sigma}_t$, i.e., obtain the Kalman Gain as the minimizer, $\text{argmin}_{K_t} \text{Tr}(\hat{\Sigma}_t)$. To minimize this trace, we first write express $\hat{\Sigma}_t$ using the Joseph formula $\hat{\Sigma}_t := (I_n - K_t H_t)\tilde{\Sigma}_t(I_n - K_t H_t)^T + K_t R_t K_t^T$ [78, 36]. Then, solving the normal equations directly, i.e $\frac{d\,\text{Tr}(\hat{\Sigma}_t)}{dK_t} = 0$ yields $K_t = \tilde{\Sigma}_t H_t'(H_t\tilde{\Sigma}_t H_t' + R_t)^{-1}$ which is of the same form as in (2.10).

The key implication of the above discussion is that the mean of the updated filtering distribution (given in (2.8)) is the minimum mean square estimator (MMSE) for the unknown state variable x_t. Optimality of $\mu_{t+1|t}$ as an estimator of predicted state at $t+1$ given $y_{1:t-1}$ and that of $H_t\mu_{t|t-1}$ as an estimator of y_t given $y_{1:t-1}$ immediately follows from the Gaussian specifications of (2.1)-(2.2). Additionally, the derivation of K_t by the above minimization implies that the Kalman Gain is optimal in the minimum variance sense.

2.2.1 Example: Training Kalman Filters for Time Series Forecasting

In this section, we will use simulated data to show how we can use KF to generate forecasts. First, we fix $M_t = M = I_6$, $H_t = H = [1,1,1,1,1,1]$, $Q_t = 2.25I_6$,

$R_t = I_1$ and assume that the initial distribution of the state variable is given by $x_0 \sim \mathcal{N}_6(0, 25I_6)$. Then invoking (2.1) and (2.2) we simulate the following sequence of observations $y_t, t = 1, 2, ..., 200$ shown in figure 2.1.

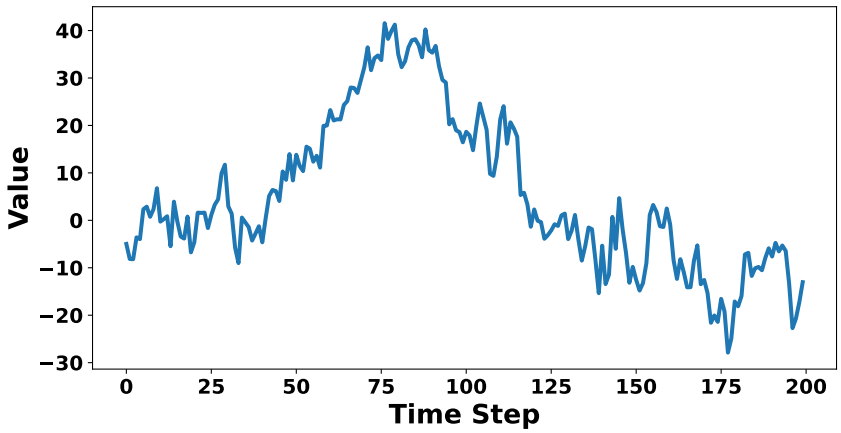

Figure 2.1: Time series

In order to train the KF, we start with specifying an initial mean, $\mu_{1|0} = \mu_0$, and covariance, $\Sigma_{1|0} = \Sigma_0$, of the state vector. Now, we apply updated filtering distribution (2.8) and (2.9) to obtain

$$\hat{\mu}_{1|1} = \mu_0 + \Sigma_0 K(y_1 - H\mu_0)$$

$$\hat{\Sigma}_{1|1} = (I - KH)\Sigma_0$$

We then apply the time updating equation with $\hat{\mu}_{2|1} = M\hat{\mu}_{1|1}$ and $\hat{\Sigma}_{2|1} = M\hat{\Sigma}_{1|1}M' + Q$.

The training of KF, therefore, consists of repeating the filtering and time updates sequentially. Thus, operationally, the KF recursive process consists of computing, at every time step t, $\hat{\mu}_{t|t}$ and $\hat{\Sigma}_{t|t}$ using $\hat{\mu}_{t|t-1}$ and $\hat{\Sigma}_{t|t-1}$ using the observation y_t. Followed by computing the time update $\hat{\mu}_{t+1|t}$ and $\hat{\Sigma}_{t+1|t}$. Figure 2.2 shows the sequence of one-step ahead forecasts obtained from the Kalman Filter superimposed with the original time series.

For KF to work, we need to know M_t, H_t, Q_t, R_t and initialize the filter with μ_0 and Σ_0. All subsequent estimates of the state vector are produced analytically without requiring us to perform any sampling from $[x_t|y_{1:t}]$ or $[x_{t+1}|y_{1:t}]$.

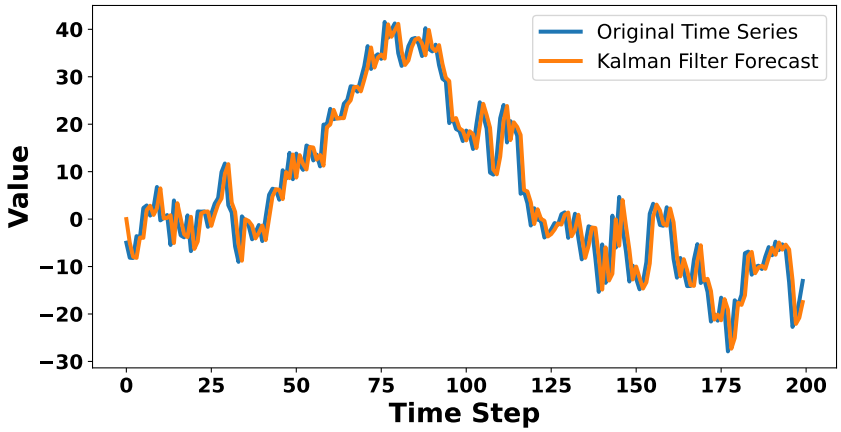

Figure 2.2: Forecasts from Kalman Filter superimposed with the original time series

2.3 Ensemble Kalman Filter

The Ensemble Kalman Filter (EnKF) is essentially a Monte Carlo approximation of the original KF described above. Operationally, EnKF requires drawing a sample of N particles of the state variable from the filtering distribution at time $t-1$. This ensemble of particles is denoted as $\hat{x}_{t-1}^{(1)}, \quad \hat{x}_{t-1}^{(2)}, ..., \hat{x}_{t-1}^{(N)}$ and is randomly drawn from $\mathcal{N}\left(\mu_{t-1|t-1}, \Sigma_{t-1|t-1}\right)$. Once this draw is made, the time updating and filter updating, similar to the KF steps, are applied to each particle $x^{(i)}$. First, we apply the time update step (2.2) to each ensemble member to obtain its evolution from time $t-1$ to time t. That is

$$\tilde{x}_t^{(i)} = M_t \hat{x}_{t-1}^{(i)} + w_t^{(i)}, \quad w_t^{(i)} \sim \mathcal{N}(0, Q_t), \qquad i = 1, \ldots, N \tag{2.11}$$

It can be shown that $(\tilde{x}_t^{(1)}, \tilde{x}_t^{(2)}, ..., \tilde{x}_t^{(N)})$ are mutually independent draws from

$\mathcal{N}(\mu_{t|t-1}, \Sigma_{t|t-1})$. Now when the observed measurements, y_t, at time step t, becomes available, all the particles in the ensemble are updated in the following way: First, N random samples of the measurement errors $v_t^{(1)}, v_t^{(2)}, ..., v_t^{(N)}$ are drawn from $\mathcal{N}(0, R_t)$. Then using these simulated measurement errors, N perturbed observations $\tilde{y_t}^{(1)}, \tilde{y_t}^{(2)}, \ldots, \tilde{y_t}^{(N)}$ are obtained using $\tilde{y_t}^{(i)} = H_t \tilde{x}_t^{(i)} + v_t^{(i)}$. It can be shown that the pair $(\tilde{x}_t^{(i)}\ , \tilde{y_t}^{(i)})|y_{1:t-1}$ are jointly normally distributed according to (2.7). The updated value of the particles representing the state variables (given that we have observed y_t) are given by:

$$\hat{x}_t^{(i)} = \tilde{x}_t^{(i)} + K_t(y_t - \tilde{y}_t^{(i)}), \qquad i = 1, \ldots, N \tag{2.12}$$

It can be easily shown that $\hat{x}_t^{(i)}|y_{1:t} \sim \mathcal{N}_n(\mu_{t|t}, \Sigma_{t|t})$. Recall from the previous section that the Kalman Gain matrix, which is required in (2.12) to compute the filtering distribution, contains $\Sigma_{t|t-1}$. Now, if Q_t is unstructured and is relatively high dimensional, the recursive exact computation of $\Sigma_{t|t-1}$ becomes computationally expensive. So instead of recursively computing this large matrix at each updating step, the EnKF uses the sample covariance matrix of the forecasted ensemble from (2.11) to estimate $\Sigma_{t|t-1}$. In other words, EnKF replaces the recursive computation associated with (2.6) by computing the sample covariance matrix at each time updating step. Thus, if the sample covariance matrix of the realizations from the conditional distribution $[x_t|y_{1:t-1}]$ is denoted by $\tilde{S}_t$, then EnKF approximates the Kalman Gain matrix by $\hat{K}_t := \tilde{S}_t H_t'(H_t \tilde{S}_t H_t' + R_t)^{-1}$. This approximation reduces the computational complexity in EnKF. Consider the situation where $n > N > m_t$, i.e., the dimension of the observation vector (y_t) is less than the number of particles in EnKF. It can be shown that when n is large, the cost of exactly computing the Kalman Gain (K_t) is given by $\mathcal{O}(n^2 m_t)$. However, the cost computing the EnKF analog of Kalman Gain

($\hat{K}_t$) is $\mathcal{O}(nNm_t)$ [62]. Clearly, for large n, EnKF offers significant computational savings.

Observe that, to train KF, we need to initialize with values for μ_0 and Σ_0 only. The updating steps recursively update the mean and variances of the conditional normal distribution from formulae supplied in (2.5)-(2.10). The EnKF, on the other hand, requires generating realizations from the initial distribution of the state variables and updates the state variables in each step. The analytical expression of K_t and hence, those of $\mu_{t|t}, \Sigma_{t|t}, \mu_{t+1|t}, \Sigma_{t+1|t}$ are replaced by their sample analog obtained from the updated realizations $\tilde{x}_t^{(i)}$ and $\hat{x}_t^{(i)}$, $i = 1, 2, ..., N$. We emphasize that conventional KF and EnKF are both feed-forward operations and do not perform any backward smoothing/sampling, i.e., they do not re-compute $\mu_{t|t}$ ($\Sigma_{t|t}$) from $\mu_{t+1|t}$ ($\Sigma_{t+1|t}$). Such backward smoothing could be performed in a formal Bayesian framework using the standard FFBS algorithm [72].

2.4 Matrix State Space Model

The matrix state space model introduces matrix variate state variables (as against vector-valued state variables in conventional linear state space model). The state evolution equation of the matrix state space model is therefore given by

$$X_t = \Theta_{t-1} X_{t-1} \psi_{t-1} + W_t, \tag{2.13}$$

with the measurement equation being:

$$Y_t = H_t X_t G_t + V_t \tag{2.14}$$

Here Y_t is a $p \times q$ dimensional observation matrix, X_t is a $m \times n$ dimensional state matrix, Θ_{t-1}, and ψ_{t-1} are $m \times m$ and $n \times n$ dimensional state transition matrices, H_t, and G_t are $p \times m$ and $n \times q$ dimensional observation matrices, W_t and V_t are $m \times n$ and $p \times q$ dimensional matrix-variate noises, respectively.

To solve this problem of matrix state estimation, [14] uses the linear property of the *vec* operator and identity-related to the Kronecker product to convert the matrix equations to their respective vector forms. Once the vector equivalent for (2.13) and (2.14) are found, the solution can be computed using the Ensemble Kalman Filter approach for the state vector in a linear state-space model. The *vec* operator is a mapping from $R^{m \times n}$ to R^{mn} in which the columns of the $R^{m \times n}$ matrix are stacked one below the other. For matrices of compatible dimensions, the following identity involving Kronecker products is used

$$vec(AXB) = (B^T \otimes A)vec(X) \tag{2.15}$$

Applying the linear property of the *vec* operator and (2.15) to (2.13), we get

$$vec(X_t) = (\psi_{t-1}^T \otimes \Theta_{t-1})vec(X_{t-1}) + vec(W_t) \tag{2.16}$$

Taking $\phi_{t-1} = \psi_{t-1}^T \otimes \Theta_{t-1}$ we get from (2.15)

$$x_t = \phi_{t-1}x_{t-1} + w_t \tag{2.17}$$

Here the lowercase letters denote the *vec* forms of their matrix analogs, for example, $x_t := vec(X_t)$. Proceeding similarly as above for (2.14) by applying the linear property of the *vec* operator and (2.15), we get

$$vec(Y_t) = (G_t^T \otimes H_t)vec(X_t) + vec(V_t) \tag{2.18}$$

Taking $\mathcal{H}_t = G_t^T \otimes H_t$ we get from (18)

$$y_t = \mathcal{H}_t x_t + v_t \tag{2.19}$$

We can see that once the original matrix state space equations of (2.13) and (2.14) have been converted to their vector analogs in (2.17) and (2.19), they are precisely similar to the state evolution and measurement equations in a linear state space model as seen in (2.1) and (2.2). Therefore, one can utilize standard EnKF to estimate the state variables appearing in a matrix state space model by first converting the matrix-variate state variable to a vectorized form.

In sections 2.1-2.4, we reviewed the concepts we will heavily use in the construction of our generalized stacking methodology. In sections 2.5-2.6, we discuss the concepts of embeddings and Monte Carlo dropouts that routinely appear during the training of deep learners. In sections 2.7 and 2.8, we will discuss two exemplar deep learners - LSTM and GCN. We will use these constructs during the deployment of our stacking methodology on real-life data.

2.5 Embeddings in Deep Learning

Embeddings in deep learning are used to convert high-dimensional sparse vectors into lower dimensions that are more amenable to machine learning and deep learning. Consider the case of a simple problem from natural language processing of predicting the sentiment of movie reviews. The training data comprises the texts of the reviews from various users and the associated sentiment, which is either positive or negative.

Since there are only two categories of reviews, the sentiment prediction problem can be considered a binary classification problem. The features or the predictors are the text reviews, and the targets are the associated sentiments of these reviews. Since the movie reviews are expressed in natural language, we must first convert them to a vector representation. The most common way to convert the text reviews to a vector representation is one hot encoding in which, first, the number of unique words in the text corpus (vocabulary) is calculated. Each text is expressed as a binary vector having the same dimension as the vocabulary size. The downside of one hot encoding is that the resulting binary vectors are highly sparse and very high dimensional, as often the vocabulary size can easily be in the millions.

Consider a fully connected artificial neural network as the modeling architecture of choice for the sentiment classification problem. The architecture consists of three layers. One is the input layer, which is the one hot encoded vector. The second layer is a hidden layer with n_{hidden} number of neurons, and the third layer is the prediction layer with a single neuron. The loss function is the binary categorical cross-entropy, which is optimized using the Adam optimizer. Since all the layers in the model are fully connected, all neurons in one layer are connected to all the neurons in the next layer. The embeddings of the individual words in the vocabulary can be obtained from these fully connected weights between the input and the hidden layer. Since each index of the one hot encoded vector corresponds to a word, all the weight connections emanating from that neuron and connecting it with the neurons of the next layer are the learned embeddings of the word corresponding to that index.

Once the model has been trained, these learned embeddings of the words can be extracted for use in many other machine learning applications, such as word embedding visualization and transfer learning. The learned embeddings of the various words can be reduced in dimensionality using any dimension reduction technique and can then be

visualized. Ideally, the words having similar semantic meaning should cluster together around each other. In the case of transfer learning, the learned embeddings of the words from an already trained model can be used with a different training corpus. Imagine a scenario when the new training corpus is small, and one hot encoding or learning the embeddings from scratch is not an option. In such a scenario, for each movie review, the embeddings of the constituent words can be extracted from a pretrained model, and such vectors can be element-wise averaged to form a vector for the entire text. For a survey on the use of embeddings in deep learning, refer to [71].

2.6 Monte Carlo Dropout

Monte Carlo Dropout is a regularization technique used to prevent overfitting in training deep neural networks. During each training iteration, all the neurons of the network have a probability p of being dropped. Consider that we train a network using an optimization algorithm such as stochastic gradient descent (SGD) in which the model parameters are learned iteratively. The optimization algorithm starts with a random initialization of the parameters and then proceeds in an iterative manner until some convergence criteria is met. Without Monte Carlo dropouts at each iteration of the SGD algorithm, all the parameters are updated. However, using Monte Carlo dropouts at each iteration, only the retained neurons are updated by the SGD algorithm. The dropped neurons have their values carried over to be used as starting values for the next iteration of SGD. In this way, the Monte Carlo Dropout technique trains many lightweight versions of the full deep neural network. At the culmination of the SGD algorithm, weights will be assigned to each neuron of the network, which is then used for prediction.

During testing time, the learned weight values from SGD are multiplied by the

dropout percentage p to account for the fact that during training at any iteration, only p percent of the total neurons were active. The same idea of dropouts used during training can also be used during testing to generate a range of predictions for each test sample, thereby allowing the computation of prediction intervals. In the case of prediction, the scaling step of multiplication by the dropout percentage is not required. Therefore, dropout presents a convenient way of computation of prediction intervals without training many different deep neural networks using either bootstrapping the training samples or the predictors. The challenge in this case is the repeated training of the full networks, which can often be overparameterized with many parameters, making the training slow and the constraint of storing the learned models, which is memory intensive. The Monte Carlo Dropout technique circumvents these issues by training a single model architecture but with many different variations in each training iteration. Therefore, at the end of each SGD iteration, we only need to store one set of weights, which is less memory-intensive. A more comprehensive discussion of the Monte Carlo dropout technique can be found in [64, 24].

2.7 Long Short Term Memory Models

Long Short Term Memory Models (LSTMs) are neural network architectures based on Recurrent Neural Networks, which are used for data that has a temporal dimension to it. Often, time series data is the most natural data type with a time dimension, but natural language data such as movie reviews and gene sequences are also treated as having a time dimension because the words or the genes appear in a particular sequence. LSTMs then process the sequence temporally, and in addition to using the features at each time step, they also have a hidden and a carry state that encodes information from the previous time steps.

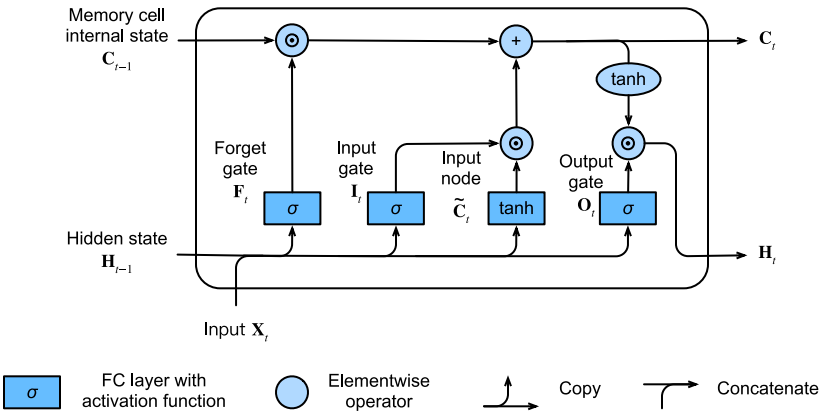

Figure 2.3: An LSTM cell showing the various operations at a time step involving the hidden state, carry state, and features[1]

Figure 2.3 from [79] shows the various operations involving the features, hidden, and carry state at the time step t. At any given time step t three types of information are available, including the features X_t, hidden state H_{t-1}, and the carry state C_{t-1}. The subscript on the hidden and the carry state is $t-1$, indicating that these values are computed from the previous time step $t-1$. F_t, I_t, $\tilde{C}_t$, and O_t are functions of X_t and H_{t-1} and are parameterized in the form of learnable weight matrices. For a complete discussion of LSTMs and how backpropagation through time is used to update the learnable parameters, refer to [79].

2.8 Graph Convolutional Networks

Graph Convolutional Networks (GCNs) are neural network architectures that are used for graphs. A graph is a collection of nodes and expresses how these nodes are interconnected with each other. Formally, a graph can be described as a collection of features for all the nodes and an adjacency matrix that defines the connection between the graph nodes. If there are N nodes and each node has D features, then the feature

[1]Figure is from [79]

matrix of the graph X is a $N \times D$ dimension matrix, and the adjacency matrix A is a $N \times N$ dimension matrix. The GCN then learns a $D \times F$ dimensional matrix to produce a processed feature matrix Z of dimensions $N \times F$. Notice that the number of rows in X and Z are the same as the number of nodes, N. Therefore, the GCN can be thought of as learning a matrix W, which transforms the original graph feature matrix, X, into the processed feature matrix, Z.

The transformed feature matrix, Z, is computed as $Z = AXW$. If the adjacency matrix, A, is binary, then the matrix product AXW takes a sum of the processed features for each node where the sum is over all of that node's neighbors. Since the number of neighbors can vary for each node, simply computing the processed feature matrix, $Z = AXW$, makes the resulting matrix biased to the number of neighbors of the nodes. To account for the difference in the number of nodes, Z is more commonly calculated as $Z = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW$. Here, D is a diagonal matrix of dimensions $N \times N$, where the diagonal elements of this matrix comprise the number of neighbors for each node. Pre and post-multiplication of the adjacency matrix, A, by $D^{-\frac{1}{2}}$, normalizes the adjacency matrix making the transformed feature matrix, $Z = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}XW$ an average of the transformed features of the neighbors. It is more common to use $\hat{A} = A + I$ instead of just A as the adjacency matrix so that the average also includes the transformed features of the node and not just the transformed features of the neighbors. For a complete discussion of GCNs, refer to [35].

Chapter 3

The Matrix Ensemble Kalman Filter-based multi-arm Neural Network

3.1 Introduction

Ensemble Kalman Filters (EnKF) have been repurposed for gradient-free training of artificial neural networks (ANNs) and deep learners (DL) [12, 11]. So far, EnKF-based training of ANN and DL only focused on single-type predictors that only require the construction of single-arm networks. However, multi-arm networks have become popular with increasing interest and availability of multi-type data. In a multi-arm network, each arm ingests a particular type of predictor, and the embeddings of these predictors get integrated before the dense prediction layer.

We consider multi-arm networks from a *model averaging* perspective. In our conceptualization, the sub-networks in each arm are constituent base learners and the prediction layer creates a convex combination of the outputs generated by these base learners (sub-networks). As mentioned in Chapter 1, our goal is to optimally estimate the weights within each sub-network, and the *cross-sub-network* model averaging weights simultaneously and attaching uncertainty to the predicted output. To that end, in this chapter, we introduce our novel matrix-variate Ensemble Kalman Filter methodology to train a multi-arm ANN (MEnKF-ANN) that performs model averaging

while training and hence could be envisioned as a generalization of conventional stacking procedure.

We offer two useful applications of our methodology in real-life data. In the first application, we deploy the MEnKF-ANN to mimic a recently developed multi-arm hybrid graph convolutional network - DeepCDR [44] and to transfer the knowledge extracted by this DL in a small sample. DeepCDR ingests drug features and different types of omics profiles in different arms to predict cancer drug responses. It integrates drug response and multi-omics data from Genomics of Drug Sensitivity in Cancer (GDSC) [76], Cancer Cell Line Encyclopedia (CCLE) [7], and The Cancer Genome Atlas (TCGA) [10] databases. Consequently, we may encounter situations where query samples came from one of the databases with a subset of features - for example, the cell lines that are unique to each database. The principal application question we want to answer is how to use DeepCDR on a small sample that comes with a reduced set of features. The existing solution is to fine-tune the original DL on a reduced set of features. We demonstrate that MEnKF-ANN can be trained on submodels of different orders with minor modifications in the state matrix. Consequently, we can store multiple MEnKF-ANN submodels and choose the appropriate submodel to make predictions when the query samples arrive with a subset of features. Our explicit *in-situ* model averaging is leveraged to generate predictions without fine-tuning the original DL when query points arrive with a reduced set of features.

The second application consists of averaging two LSTM classifiers trained to classify what carbohydrate substrates are digested and utilized by a microbiome sample characterized by genomic sequences consisting of polysaccharide utilization loci (PULs). Exploratory analyses show that the uncertainty intervals generated by Monte Carlo (MC) dropout heavily rely on the architecture of the LSTM. So, even though a standard stacking-based meta-learner could produce model-averaged estimates of

success probabilities, it fails to propagate the uncertainties inherent in the probabilities estimated by the constituent LSTMs. We apply MEnKF-ANN to resolve this issue. Our results show that the average width of the prediction intervals obtained from our approach is more stable than the variability observed in MC dropout-induced prediction intervals obtained from constituent base LSTMS.

The remainder of the chapter is organized as follows: Section 3.2 reviews how EnKF has been used in the context of training neural networks. Section 3.3 details the construction of our MEnKF-ANN stacker. In section 3.4, we deploy MEnKF-ANN to predict cancer drug responses on a small dataset by mimicking DeepCDR. We also demonstrate the ability of our approach to handle missing features in the query samples. Section 3.5 illustrates how we can use MEnKF-ANN to attach uncertainty to a convex combination of two LSTM classifiers. We offer concluding remarks in section 3.6.

3.2 Background

In this section, we offer a brief overview of how EnKF has been used to train NNs and DLs.

Using Kalman Filters for Deep Neural Networks: The usage of KF and EnKF techniques has been surprisingly sparse in deep learning literature. It is probably attributable to the fact that conventional KF and EnKF are suitable for estimating parameters in linear state-space models. However, several extensions have been proposed to generalize KF in nonlinear settings. [69] introduced the unscented KF that better approximates nonlinear systems while making it amenable to the KF framework. [2] developed the state augmentation method that offered a generic technique to handle nonlinearity in state-space models via the KF framework. [30]

utilized this state augmentation technique to develop a generic method to train ANNs using state-augmented KF. They derived the state-augmented KF's forecast and update equations in ANNs, thereby providing the algebraic framework to train DLs using the Ensemble Kalman Filters approach.

Imagine a scenario when the measurement equation of the linear Gaussian state-space model in (2.1) is not linear anymore. This situation can arise commonly in supervised learning models such as neural networks where the measurement function is non-linear. Consider now a slightly modified version of (2.1) as follows.

$$y_t = M(x_t) + v_t, \quad v_t \in \mathcal{N}_{m_t}(0, R_t) \tag{3.1}$$

Here, M is the non-linear function induced by the neural network architecture. The augmented state variable is an artifact introduced to cast the measurement equation in (3.1) into a linear form such as (2.1). Consider a new variable z, which is defined as

$$z_t = \begin{pmatrix} M(x_t) \\ x_t \end{pmatrix} \tag{3.2}$$

Now consider a modified version of H_t from (2.1) defined as follows.

$$H_t = \begin{pmatrix} I_{m_t}, 0_{m_t \times n} \end{pmatrix} \tag{3.3}$$

Now using the new definition of H_t and z_t from (3.3) and (3.2), respectively, the linear analog of (2.1) is given by

$$y_t = H_t z_t + v_t, \quad v_t \in \mathcal{N}_{m_t}(0, R_t) \tag{3.4}$$

The measurement equation is now given by (3.4), a linear measurement equation similar to (2.1). Using this new definition of H_t and the augmented state variable z_t, we can use the Ensemble Kalman Filtering forecast and update equations from (2.11) and (2.12). Equations (3.3) and (3.4) form the foundation of introducing Kalman Filters in deep learning.

Only a few studies used the foregoing ideas to train NNs and extend that to DLs. For instance, [61] used extended KF to train feed-forward NN. [11] used the updating equations in [30] to train a single hidden layer ANN and demonstrated how using state augmentation, one can estimate the measurement error variance. State-augmented EnKF formulation was also used to estimate parameters in LSTMs [12]. [77] trained a Convolutional Neural Network using EnKF with the strategies outlined above.

However, no study, to the best of our knowledge, used EnKF to perform model averaging on multiple base ANNs and demonstrated how this model averaging could be connected with multi-arm DLs. This dissertation addresses that gap in the literature.

3.3 Methodology

First, we offer a generic construction of the proposed MEnKF-ANN procedure and then describe how this method could be deployed to transfer information from a multi-arm hybrid graph convolutional network. We will use the following notations. $Y \in \mathcal{R}$ is our target response. We have a total of $m = \sum_{t=1}^{T} m_t$ training instances, with m_t being the number of training data points in the t^{th} batch. $v_t^f \in \mathcal{R}^p$ and $v_t^g \in \mathcal{R}^q$ denote two different representations of the features (possibly of different dimensions) for the t^{th} batch of data. Consider two ANN architectures, denoted by f and g. The architecture f takes v_t^f as input features and combines that with its own *within-learner* weights w_t^f to generate the prediction for the target response Y.

Similarly, the architecture g takes v_t^g as input features and combines that with its own *within-learner* weights w_t^g to generate the prediction for the target response Y. Clearly, if $m_t > 1$, we can envision f and g to be ANN architectures that get trained on a dataset $(Y, v_t^f)^{m_t \times dim(v_t^f)+1}$ and $(Y, v_t^g)^{m_t \times dim(v_t^g)+1}$, respectively. Assume that, in each batch, f and g have n_f, n_g number of learnable parameters with $n_f = n_g$ (if $n_f \neq n_g$, we can use suitable padding when updating the weights) and our final prediction for Y is going be a convex combination of the predictions produced by f and g. For expositional simplicity, we will simply refer to f and g as ANNs and they form the base learners for us. Our goal is to simultaneously update w_t^f, w_t^g and the *cross-learner* weight that generates the convex combination of the outputs from f and g and guarantee that the estimates are jointly optimal under expected square error loss.

3.3.1 Matrix Kalman Filter based Multi-arm ANN

Consider the state matrix, X_t, associated with the t^{th} batch of data given by

$$X_t^{(m_t+n_g+1)\times 2} = \begin{bmatrix} f(v_t^f, w_t^f), & g(v_t^g, w_t^g) \\ w_t^f, & w_t^g \\ 0, & a_t \end{bmatrix} \tag{3.5}$$

where a_t is a real-valued scalar parameter. Define $H_t^{m_t \times (m_t+n_g+1)} = [I_{m_t}, 0_{m_t \times (n_g+1)}]$ and $G_t^{2\times 1} = [1 - \sigma(a_t), \quad \sigma(a_t)]^T$ where $\sigma(.) : \mathcal{R} \to [0, 1]$, with the sigmoid function being a popular choice of $\sigma(.)$. Additionally, define $\Theta_{t-1} = I_{m_t+n_g+1}$ and $\psi_{t-1} = I_2$. We are now in a position to define the Matrix Kalman Filter.

The measurement equation is given by:

$$Y_t = H_t X_t G_t + \epsilon_t \tag{3.6}$$

with the state evolution equation being

$$X_t = \Theta_{t-1} X_{t-1} \psi_{t-1} + \eta_t \tag{3.7}$$

Writing in *vec* format, (3.7) becomes

$$x_t = vec(X_t) = (\psi_{t-1}^T \otimes \Theta_{t-1}) vec(X_{t-1}) + vec(\eta_t) \tag{3.8}$$

Now letting $\phi_{t-1} = \psi_{t-1}^T \otimes \Theta_{t-1}$ and $\tilde{\eta}_t = vec(\eta_t)$ we get from (3.8)

$$x_t = \phi_{t-1} x_{t-1} + \tilde{\eta}_t \tag{3.9}$$

(3.6) can similarly be compactified as

$$y_t = \mathcal{H}_t x_t + \epsilon_t \tag{3.10}$$

where $\mathcal{H}_t = G_t^T \otimes H_t$. Observe that(3.10) and (3.9) have the same form as the standard representation of linear state space model described in (2.1) and (2.2). Therefore, we can get the matrix state space model's solution by converting it to the vector state space model and then using EnKF to approximate the updating equations. We direct the audience to [14] for more details on Matrix Kalman Filters.

3.3.2 Reparametrizing MEnKF-ANN for Computational Efficiency

The above construction of X_t, H_t, and G_t performs automatic model averaging while training. First, consider the matrix multiplication of $H_t X_t$ from (3.6). This would be a $m_t \times 2$ dimensional matrix in which the first column is the prediction for the t^{th} batch, from the neural network f and the second column is the prediction from the

neural network g. Post multiplication by G_t would take the weighted average of each row in H_tX_t where the weights are defined inside the G_t matrix. Now consider the matrix multiplication of $H_tX_tG_t$ from (3.6)

$$\begin{aligned} H_tX_tG_t &= \begin{bmatrix} f(v_t^f, w_t^f), & g(v_t^g, w_t^g) \end{bmatrix} \begin{bmatrix} 1-\sigma(a_t) \\ \sigma(a_t) \end{bmatrix} \\ &= \begin{bmatrix} (1-\sigma(a_t))f(v_t^f, w_t^f) + \sigma(a_t)g(v_t^g, w_t^g) \end{bmatrix} \end{aligned} \tag{3.11}$$

(3.11) demonstrates how our construction explicitly performs model averaging across the batches with $1-\sigma(a_t)$ and $\sigma(a_t)$ being the convex weights allocated to the ANNs f and g, respectively.

Although the foregoing construction connects Matrix KF formulation with multi-arm ANN and performs explicit model averaging, it suffers from a computational bottleneck. Using (3.9) and (3.10) the estimated Kalman Gain Matrix would be $K_t = \tilde{S}_t\mathcal{H}_t^T(\mathcal{H}_t\tilde{S}_t\mathcal{H}_t^T + \sigma_y^2 I_{m_t})^{-1}$. However, in the above parameterization we have $G_t = [1-\sigma(a_t), \quad \sigma(a_t)]^T$ and $\mathcal{H}_t = G_t^T \otimes H_t$. This would require computation of the estimated Kalman Gain matrix for each member in EnKF since, at any given iteration of our MEnKF-ANN, we have an a_t for each ensemble member. Thus, computation complexity associated with Kalman Gain computation increases linearly with the ensemble size in the above parametrization of the MEnKF-ANN.

To alleviate this computational bottleneck, consider the following parametrization:

$$X_t = \begin{bmatrix} (1-\sigma(a_t))f(v_t^f, w_t^f), & \sigma(a_t)g(v_t^g, w_t^g) \\ w_t^f, & w_t^g \\ 0, & a_t \end{bmatrix} \tag{3.12}$$

and $G_t = [1, \ \ 1]^T$. We still have explicit model averaging in the measurement equation, i.e.,

$$H_t X_t G_t = \Big[(1 - \sigma(a_t)) f(v_t^f, w_t^f) + \sigma(a_t) g(v_t^g, w_t^g)\Big] \tag{3.13}$$

but $\mathcal{H}_t$ does not depend on a_t, therefore the matrix products for the Kalman Gain computation can now be computed once for each batch.

Turning to the variance parameter in the measurement equation (3.10). Assume $\epsilon_t \sim \mathcal{N}_{m_t}(0, \nu_y^2 I_{m_t})$. To estimate ν_y^2, we augment the state vector as follows:

$$X_t^{(m_t+n_g+2)\times 2} = \begin{bmatrix} (1 - \sigma(a_t)) f(v_t^f, w_t^f), & \sigma(a_t) g(v_t^g, w_t^g) \\ w_t^f, & w_t^g \\ 0, & a_t \\ 0, & b_t \end{bmatrix} \tag{3.14}$$

where $\nu_y^2 = \log(1 + e^{b_t})$ and H_t in (3.6) now becomes $[I_{m_t}, 0_{m_t \times (n_g+2)}]$. We used a softplus transformation for ν_y^2 instead of the usual log transformation for computational stability.

3.3.3 Explicating MEnKF-ANN

Recall, our goal, as laid out in Chapter 1, was to optimally estimate *cross-learner* and *within-learner* weights simultaneously and attach uncertainty to the predictions obtained from the averaged model. In the foregoing construction of the proposed MEnKF-ANN, we have the base learners $f(.), g(.)$, learner-specific weights w_t^f, w_t^g, and the model averaging weight (or *cross-learner* weight) a_t, all incorporated in the state

matrix X_t as shown in (3.5). The measurement model (3.6) is the model-averaged output coming out from the base learners $f(.)$ and $g(.)$. We have framed the problem in an augmented state-space fashion and invoked EnKF to estimate the state variables.

First notice that the notion of time in conventional EnKF setup is replaced by batch. This could be justified by the fact that in practice ANNs are often trained over batches with batch size smaller than the size of the training data predominantly because of memory requirements in training ANNs. Additionally, weights in the ANNs are updated after each batch is processed- analogous to the EnKF setup where the state variables are updated when new observations (observation vector, more precisely) come along. Thus the specification of $f(.)$ and $g(.)$ as ANNs correlate well with the conceptual framework of EnKF with augmented state space.

The EnKF machinery allows each element in the state matrix to be updated at the same time in each updating step of EnKF. Thus the *within-learner* weights and *cross-learner* weights are updated *in-situ*. Next, as we outlined in Chapter 2, the KF framework yields optimal estimates of the state variables under squared error loss. That is the $\hat{\mu_t}$ is indeed the minimum mean square error estimator for $E(X_t|y_{1:t})$. Although EnKF is an approximation of KF, we can show that under Gaussian specification, EnKF estimates converge in probability to their KF analog as the ensemble size increases (see section 3.7 for proof). Putting the pieces together, we surmise that the estimates of state variables produced by our MEnKF-ANN are asymptotically (with respect to ensemble size) optimal under mean square error loss. Recall that, conventional stacking estimates the *cross-learner* weights under square error loss as well. But, in the proposed MEnKF-ANN, we are estimating all the weights jointly by minimizing a global expected square error loss. Hence, we envision this approach to be a generalization of stacking wherein the *within-learner* and *cross-learner* weights are jointly optimal.

Turning to the uncertainty quantification issue, we first contrast the nature of the *within-learner* weights obtained from MEnKF-ANN with those obtained from conventional ANNs (or for that matter any DLs) trained via backpropagation. Conventional ANN training treats the set of *within-learner* weights as unknown internal parameters that are estimated via minimization of a suitably chosen loss function. Therefore, fundamentally the training process is geared to learn about the internal model parameters so as to reduce our epistemic uncertainty about them. As such [53] states unequivocally, *"uncertainty about parameters in statistical models is almost invariably epistemic"*. However, since conventional ANN treats parameters as constants, no probability statement can be attached to them directly. In our conceptualization, the weights are included in the state matrix implying that we are starting off with the assumption that the internal model parameters (w's) and *cross-model* parameters ($\sigma(a)$) are jointly normally distributed random variables. Consequently, we can directly attach probability statements to these parameters. To understand how we are attaching the probabilities, we recall (from Chapter 2) that each EnKF update step essentially consists of drawing independent realizations of state variables from the updated joint distribution of all the state variables. The point estimates of the model parameters are obtained from this updated distribution. The entire distribution profile of all model parameters attempts to capture the epistemic uncertainty about the parameters themselves, under the assumption that the model(s) is well-specified. Since the set of models (both $f(.)$ and $g(.)$ and probability models for measurement and state variables) are fixed a-priori uncertainty due to model misspecification cannot be captured.

Turning to uncertainty in predicting the target response, once again we observe that the point estimate of predictions in conventional ANNs (or DLs) is a deterministic function of the internal model parameters and non-stochastic feature set. Consequently,

the target of prediction is, again, an unknown constant to which probability statements can not be attached. The MEnKF-ANN construction, on the other hand, updates the entire conditional distribution of $y_t|y_{1:t-1}$ (see (2.7)), thereby generating the full distributional profile of the prediction target. This updated distribution, again, attempts to capture the epistemic uncertainty in the prediction under the assumption that the predictive models and probability models are not misspecified. Thus, the prediction uncertainty is quantified in the following sense:

Suppose the conditional distribution of the target response variable is Gaussian with unknown mean and variance. Suppose the mean is a convex combination of two ANNs (ingesting two types of non-stochastic features) with known architecture but unknown stochastic weights. Further, suppose the joint distribution of all unknown weights is Gaussian as well. Under this set of assumptions, MEnKF-ANN can coherently generate the predictive distribution of the response variable given the training dataset and the feature vector for the query point by propagating the uncertainty associated with all the model weights.

3.3.4 Connecting MEnKF-ANN with DL

To connect our conceptualization of MEnKF-ANN with a deep learner, we will use an illustrative example. A recently developed deep learner (DeepCDR) uses a multi-arm graph convolutional network (GCN) technique to predict in-vitro cancer drug responses on cell lines. DeepCDR uses drug features and multi-omics data as different sub-networks to predict the target response variable logarithm of half-maximal inhibitory concentration ($\log IC_{50}$), which is an indicator of drug response for cancer cell lines [44]. We consider the drug information and the omics information as two classes of predictors.

The MEnKF-ANN uses the observed $\log IC_{50}$ as the target response. The

DeepCDR embeddings of omics data and drug data are treated as numerical features v_t^f and v_t^g, respectively, and supplied to the ANNs f and g, respectively. The convex weight $\sigma(a)$ combines the prediction of Y generated by f and g using v_t^f as v_t^g as features, respectively. The estimate of $\sigma(a)$ assesses the relative predictive capacity of drug information compared to omics information. We emphasize that, in our current conceptualization, multiple DLs are not directly absorbed by MEnKF-ANN as base learners. Rather, we use these DLs simply to extract numerical values of features associated with complex predictors. In our context, DeepCDR is used to extract embeddings of drugs and omics features only.

To initialize the ensemble, we draw the members in the state vector (3.14) from $\mathcal{N}_{2n_g+2}(\mathbf{0}, \nu_x^2 I_{2n_g+2})$. We update each element of the augmented state vector $w_t^{f,(i)}$, $w_t^{g,(i)}$, $a_t^{(i)}$, $b_t^{(i)}$ using the t^{th} batch of data. N and m_t are treated as tuning parameters. These tuning parameters are chosen by tracking the empirical convergence of training error and the trajectory of dominant eigenvalue of $(I - K_t\mathcal{H}_t)$. We offer formal proof of this assertion in Proposition 1 (see section 3.7). Operationally, we train MEnKF-ANN for multiple N and batch sizes and track the training error. In each training run, the exit criterion is set such that MEnKF-ANN *sees* the entire training set and the dominant eigenvalue of $(I - K_t\mathcal{H}_t)$ does not exceed 1 for the last five updates. Due to the properties of Kalman Filters (see Proposition 1), we expect the training error trajectories obtained from different choices of N and m_t to converge.

3.4 Application I: Transfer learning using MEnKF-ANN

This section offers a high-level overview of the focal deep learner of DeepCDR and describes the dataset used to train this DL. We describe two scenarios to demonstrate how MEnKF-ANN could transfer information gleaned from training DeepCDR on a

large dataset to a small batch of additional data that DeepCDR did not *see*. We then investigate how the MEnKF-ANN could also transfer information when the additional data comes with a subset of predictors that were originally used to train DeepCDR.

3.4.1 Application I: Data description

DeepCDR is a hybrid convolutional neural network for cancer drug response prediction that consists of a graph convolutional network for integrating drug-specific features based on the chemical representations of drugs and multiple subnetworks for integrating multi-omics profiles [44]. This model encodes the multi-omic features using three subnetworks corresponding to each omics type. Each subnetwork takes as input individual omics features and then learns its embedding. These omics embeddings are then concatenated with the embeddings of the drugs, and then used as inputs to the final prediction layer.

In the data curation step, this method utilized the Cancer Cell Line Encyclopedia (CCLE) database to extract genomic mutation, gene expression, and DNA methylation profiles for cancer cell lines. It then extracted drug response data, in the form of $\log IC_{50}$, for these cell lines from the GDSC database. Finally, drugs were represented by a matrix consisting of a 75-dimensional feature vector representing each atom of the drug. PubChem library was used to obtain the structural files of the drugs. The final curated dataset consisted of 86530 instances across 238 drugs and 561 cell line combinations. [44] provided the datasets originally used to train DeepCDR at https://github.com/kimmo1019/DeepCDR/tree/master-/data. We first trained the DeepCDR model using 69214 samples and used 17316 samples as validation samples to determine the stopping time. We used the original network architecture in [44] with the author-recommended hyper-parameter combinations.

3.4.2 Application I: Results

We use the following design to demonstrate how to transfer information via MEnKF-ANN to a small sample. First, we chose 20 drugs that appear infrequently in the training dataset we used to train DeepCDR. These drugs were AKT inhibitor VIII, AZD6482, Afatinib, Avagacestat, BMS-536924, Bicalutamide, Bleomycin, CHIR-99021, GSK269962A, IOX2, JQ1, Olaparib, PFI-1, PLX-4720, Pictilisib, Refametinib, SB505124, SN-38, Selumetinib, and UNC0638. These 20 drugs accounted for approximately 10% of the total training samples for the DeepCDR training, i.e., 90% of cancer cell lines corresponding to these drugs have not been seen by DeepCDR. For 5000 of such samples that have not been used to train DeepCDR, we extracted the IC_{50} and the raw drug and omic features. These instances form the focal dataset ($\boldsymbol{Z}$) for training and testing MEnKF-ANN.

We performed 5-fold cross-validation with MEnKF-ANN, so in each fold, MEnKF-ANN was trained on 4000 instances, and 1000 data points were used for testing. We supplied all the drug and omics features available in $\boldsymbol{Z}$ to the original DeepCDR and extracted their embeddings. In the training set, we supplied the drug embeddings to one arm of the MEnKF-ANN, the second arm ingested all the omics embeddings, and then we trained MEnKF-ANN with $\log IC_{50}$ as the target variable. We then supplied the embeddings associated with the test set and predicted $\log IC_{50}$ for the test set. Figure 3.2 shows the training RMSE for different ensemble sizes (N) across the update iterations. Observe that the training RMSEs show signs of convergence as the training progresses, as predicted by the convergence theorem. The training RMSE obtained with $N = 200$ is nearly indistinguishable from that obtained with $N = 400$. This offers empirical support to our choice of $N = 196$. Figure 3.1 shows the cross-validation scatter plot of observed and predicted $\log IC_{50}$.

To benchmark the prediction performance of MEnKF-ANN, we notice that our method essentially approximates the dense layers of DeepCDR. Hence we can *freeze* the convolution layers of the DL and fine-tune the dense layers using $\boldsymbol{Z}$. Since the DL is trained with dropouts, we activate the dropout layers during the prediction phase and obtain a set of predicted values for each test instance. We compute the empirical 95% dropout-induced prediction interval for each test sample and report the average coverage probabilities and width of the 95% dropout prediction intervals. Table 3.1 reports the average cross-validated RMSPE, average Pearson correlation between predicted and observed values, average coverage probabilities, and width of the uncertainty intervals. We also report the average weightage associated with the drug arm of MEnKF-ANN.

Table 3.1: Averaged cross-validation metrics for MEnKF-ANN and fine-tuned DeepCDR trained with all available features.

Model	ν_x^2	N	RMSPE	Coverage	Average width	ρ	Drug weight
MEnKF-ANN	1	196	1.38	98%	7.18	0.70	0.77
Fine tuned DeepCDR	NA	NA	1.44	46%	1.71	0.67	NA

Observe that, in terms of conventional performance metrics (RMSPE and ρ), MEnKF-ANN and fine-tuned DeepCDR produce comparable results. However, our approach is vastly superior to the fine-tuned DeepCDR when we look at the prediction uncertainty. As such, coverage associated with dropout-induced uncertainty intervals raises questions about the adequacy of fine-tuned deep-learning models.

Next, we consider the situation where one of the omics features is completely missing in $\boldsymbol{Z}$. The DeepCDR network encodes the multi-omic features using a sub-network corresponding to each omics. When an omics feature, gene expression (say), is missing, the DeepCDR model cannot obtain its corresponding embedding, and either needs to be retrained or the positions in the concatenated vector allocated to the gene expression embeddings need to be padded before it can predict $\log(IC_{50})$.

However, since the three subnetworks in DeepCDR operate on the omics features individually, the learned embeddings for the omics features that are not missing can still be extracted from the DL. These available embeddings of the non-missing omics features can be used to retrain MEnKF-ANN with a state matrix that does not contain the weights corresponding to missing features. This reduced state matrix can be viewed as a marginalized version of the full state matrix used to train MEnKF-ANN with all features. Since matrix normal is closed under marginalization [26], the entire theoretical construct of MEnKF-ANN holds, and all the updating equations have the same form with appropriate adjustment in the dimensions of the matrices.

Hence, in this scenario, we pretend that the foregoing 5000 samples in $\boldsymbol{Z}$ come with missing gene expression. We supply the available omics features (gene mutation and DNA methylation) and drug features to appropriate subnetworks of the pre-trained DeepCDR to extract the respective embeddings. MEnKF-ANN is then trained on the reduced set of features. Once again, we report the 5-fold cross-validation results in Table 3.2.

Table 3.2: Averaged cross-validation metrics for MEnKF-ANN trained with a reduced set of features.

Model	ν_x^2	N	RMSPE	Coverage	Average width	ρ	Drug weight
MEnKF-ANN	1	132	1.39	91%	4.90	0.69	0.80

3.5 Application II: Attaching uncertainty to stacked LSTM classifier using MEnKF-ANN

In this section, we apply MEnKF-ANN to attach uncertainty to the predicted probabilities produced by a convex combination of two different architectures of LSTM. The training objective is to classify what carbohydrate substrates are digested and

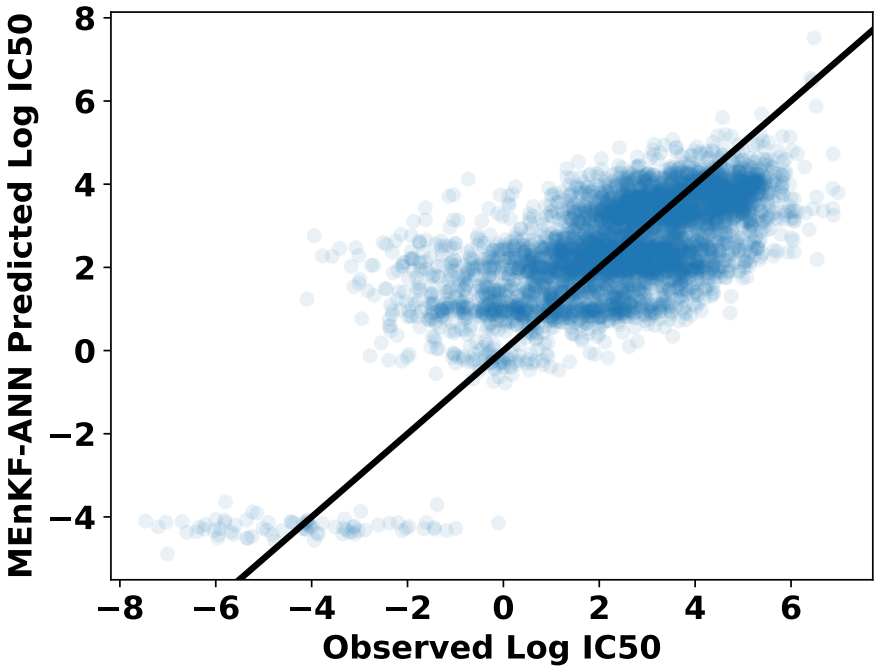

Figure 3.1: Scatterplot showing the observed $\log IC_{50}$ values with the predicted $\log IC_{50}$ values over all cross-validation test folds for MEnKF-ANN trained without gene expression features.

utilized by a microbiome sample characterized by genomic sequences consisting of polysaccharide utilization loci (PULs) [9] and their encoded genes.

3.5.1 Application II: Motivating Problem

The human gut, especially the colon, is a carbohydrate-rich environment [31]. However, most of the non-starch polysaccharides (for example, xylan, pectin, resistant glycans) reach the colon undegraded [58] because human digestive system does not produce the enzymes required to degrade these polysaccharides [22]. Instead, humans have developed a complex symbiotic relationship with gut microbiota, with the latter providing a large set of enzymes for degrading the aforementioned non-digestible dietary components [68]. Consequently, an essential task in studying the human gut microbiome is to predict what carbohydrate substrates a microbiome sample can

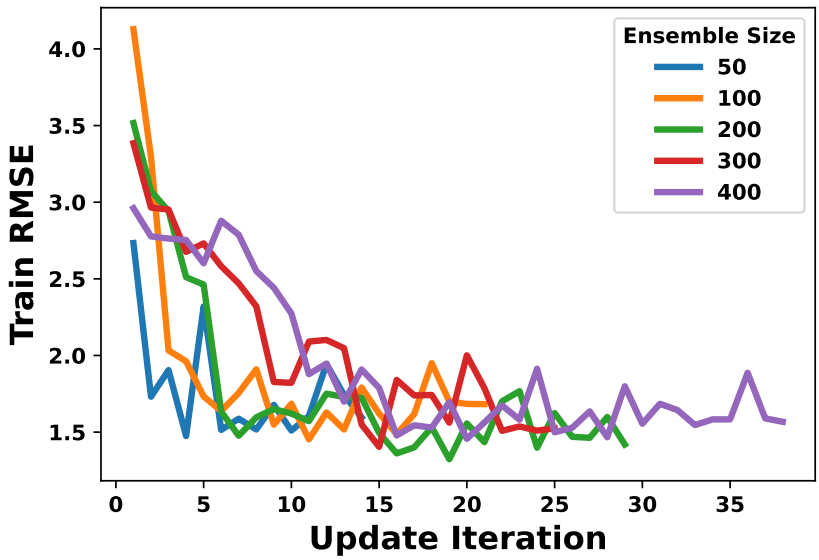

Figure 3.2: Trajectory of MEnKF-ANN training RMSE for different ensemble sizes (N)

digest from the genetic characterization of the said microbiome [37].

To generate a focused genetic characterization of the microbes that relates to their carbohydrate utilization property, one often investigates the genes encoding the Carbohydrate Active Enzymes (CAZymes) and other proteins that target glycosidic linkages and act to degrade, synthesize, or modify carbohydrates [45, 80]. This set of genes tends to form physically linked gene clusters in the genome known as polysaccharide utilization loci (PULs) [9]. Consequently, the gene sequences associated with PULs of microbes could be used as a predictor to ascertain the carbohydrate substrate the microbe can efficiently degrade. However, these gene sequences are string-valued quantities [29, 65] and hence their naive quantitative representations (for instance, one-hot-encoding or count vectorization) often do not produce classifiers with acceptable accuracy [4]. Instead, we can use LSTM to process the entire sequence of string-valued features and then implement a classifier with a categorical loss function. Since the experimental characterization of new PULs for carbohydrate utilization is

an expensive process [3], we need large enough labeled samples. Consequently, as we demonstrate below, the output of the LSTMs is sensitive to its architecture.

We extract the dataset from the dbCAN-PUL database [3] that contains experimentally verified PULs and the corresponding GenBank sequences of these PULs along with known target carbohydrate substrates. Figure 3.3 shows an example of a gene sequence associated with a PUL for the substrate Pectin. We have a total of approximately 411 data points. Figure 3.4 shows the dataset's frequency distribution of various target substrates. We do not have sufficient samples to train a complex DL to classify all the available substrates. Hence we propose to classify the two most frequently occurring target substrates - Xylan and Pectin. Seventy-four samples belong to these two classes of substrates in a reasonably balanced way.

We train two LSTM binary classifiers on 66 samples and retain eight hold-out samples for test purposes. One LSTM was trained with two dropout layers - one inside the LSTM and one just before the final prediction layer. The second LSTM was trained with dropout in the LSTM layer only. We activated the dropout layers during the prediction phase, which generated multiple copies of the prediction for each test sample. Figure 3.5 shows each test sample's predicted probabilities and the boxplot constructed using the foregoing set of predictions. The left panel (top and bottom) shows these metrics for eight held-out test samples for the first LSTM, while the right panel shows the same for the second LSTM. Observe that the point prediction remains reasonably stable under both LSTM configurations, but the width of these intervals are sensitive to the number and placement of the dropout layers. If we wish to perform simple averaging to predict the probabilities of the test sample, how should we attach uncertainty to the equally weighted model-averaged predictions? We deploy MEnKF-ANN to answer this question.

Suppose p is the probability of observing a sample of a particular category.

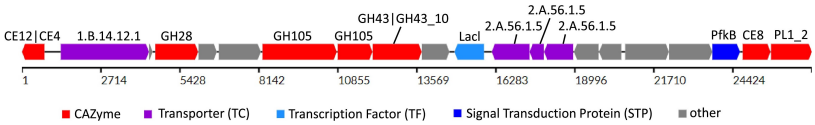

Figure 3.3: Pectin PUL

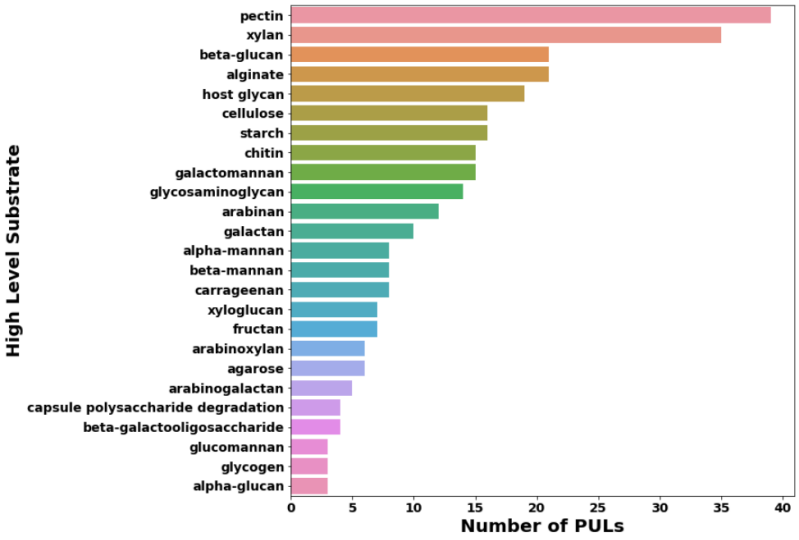

Figure 3.4: Frequency distribution for the various substrates

The trained LSTMs produce $\hat{p_1}$ and $\hat{p_2}$ for each training instance, along with the embedding of the associated gene sequences. To attach uncertainty to the average of the above probabilities, MEnKF-ANN uses $logit(0.5 * \hat{p_1} + 0.5 * \hat{p_2})$ as the target response. The embedding of the gene sequences obtained from one LSTM is fed into one arm of MEnKF-ANN while the other arm ingests the embeddings generated by another LSTM. The convex weights $\sigma(a)$ enter into the state matrix so that we can assess whether our MEnKF-ANN can accurately recover the *cross-learner* weights. We emphasize that MEnKF-ANN is not directly combining two LSTMs. The base learners in MEnKF-ANN are still two ANNs (f, g) where f absorbs the embeddings produced by one of the LSTMs and g absorbs the embeddings produced by the other LSTM. The measurement model specifies that the convex combination (with unknown weights) of the outputs from f and g is the mean of the target response variable which is the logit

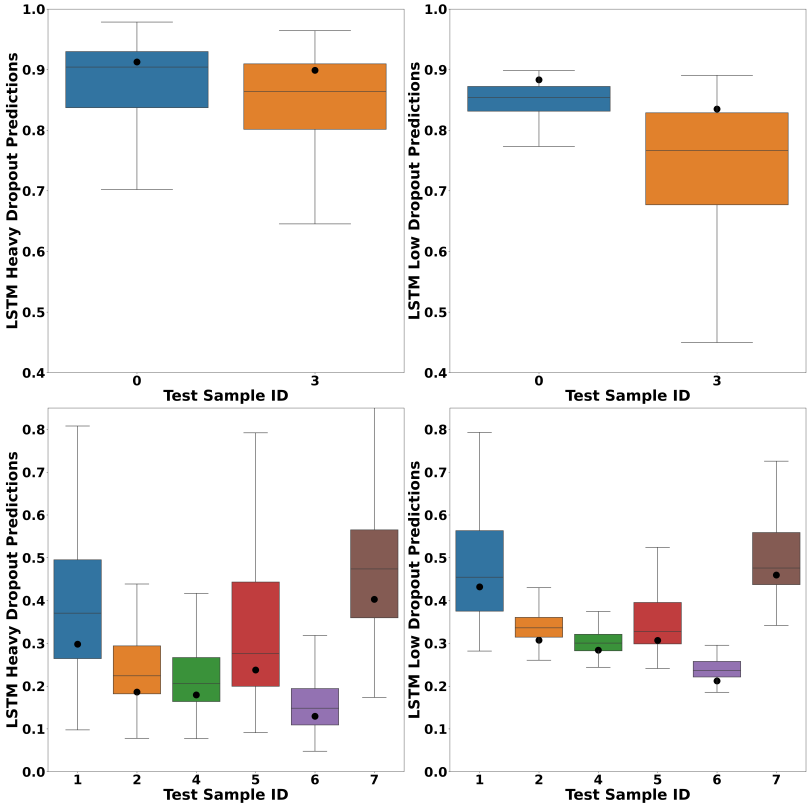

Figure 3.5: Boxplots showing the predictions superimposed with the ground truth value from the two LSTM architectures

transformed average of the two LSTM outputs. The uncertainty interval produced by MEnKF-ANN, quite obviously, targets to capture $logit(0.5 * \tilde{p_1} + 0.5 * \tilde{p_2})$ where $\tilde{p_i}$ is the predicted probabilities coming from the two LSTMs under consideration.

But how good are these intervals in capturing true probabilities? This is a relevant question because unlike the previous application (Section 3.4) the MEnKF-ANN here does not *see* the binary response. Instead, it targets relevant statistics generated by the LSTMs that get trained on the original labeled data. To answer this question, we perform extensive simulations.

3.5.2 Application II: Simulations

We conducted extensive simulations to assess how well our MEnKF-ANN can approximate an LSTM binary classifier. This simulation exercise aims to demonstrate that our MEnKF-ANN is not only "adequate" in approximating the probabilities produced by LSTM but can also capture the "true" probabilities that generate binary labels. We compute the coverage and width of the prediction intervals of the target probabilities in the test set to assess the "adequacy" of the approximator. Then, we compare this coverage and width with those computed directly via an LSTM trained with MC dropout. Admittedly, the prediction intervals obtained from the latter are different from those computed from MEnKF-ANN. However, if the ground truth probabilities are known, an adequate approximator should be able to achieve near-nominal coverage when the approximand is not misspecified.

Our simulation strategy mimics the focal dataset and uses the gene sequences associated with the original PULs to generate labels. As mentioned above, we extracted $\hat{p}$ from the LSTM trained on the original dbCAN-PUL data. We call this LSTM the *true LSTM*. We consider $\hat{p}$ the true probabilities for synthetic data generation. We then use noisy copies of $\hat{p}$ to generate a synthetic label in the following way: generate $logit(\tilde{p}_i^{(j)}) = logit(\hat{p}_i) + \epsilon_i^{*(j)},\ i = 1, 2, ..., m, j = 1, 2, ..., J$, where J is the number of the simulated dataset and m is the number of data points in each simulated set, the perturbation $\epsilon_i^{*(j)}$ are iid $\mathcal{N}(0, 0.01^2)$. We generate synthetic labels $\tilde{Y}$ by thresholding $\tilde{p}_i^{(j)}$ at 0.5, i.e $\tilde{Y}_i^{(j)} = I(\tilde{p}_i^{(j)} > 0.5)$. Then the simulated dataset consists of $D^{(j)} = \{\boldsymbol{F}, \tilde{Y}^{(j)},\ j = 1, 2, ..., J\}$, where $\boldsymbol{F}$ is the set of original gene sequences from dbCAN-PUL.

Simulation 1: Now in each $D^{(j)}$, we train a second LSTM (with two dropout layers) and extract $\tilde{\tilde{p}}_i^{(j)}, i = 1, 2, ..., m$ along with the embedding of the gene sequences.

We call these LSTMs, trained on $D^{(j)}$, the *fitted LSTMs*. Note that the embeddings from *fitted LSTMs* could potentially be different from those obtained from the *true LSTM*. We denote the embedding from *fitted LSTMs* by $v_i^{(j),f},\ j = 1, 2, ...J$. Our MEnKF-ANN is constructed to approximate the *fitted LSTMs*. To that end, the approximator uses $logit(\tilde{\tilde{p}}_i^{(j)})$ as the target response. $v_i^{(j),f}$ are supplied as features to one arm of the ANN, the other arm ingests $v_i^{(j),g}$ - the Doc2Vec [41] embedding of $\boldsymbol{F}$. Once the MEnKF-ANN is trained, we use a hold-out set in each simulated dataset to generate predictive probabilities from the forecast distribution for each member in the KF ensemble and compute the empirical 95% predictive interval at $logit^{-1}$ scale. To measure the adequacy of MEnKF-ANN, we compute the proportion of times the foregoing predictive interval contains $\hat{p}$ in held-out test data. We expect this coverage to be close to the nominal 95%, and the average width of these intervals should not be greater than 0.5. Additionally, observe that the data-generating model uses LSTM embedding of F; hence, using Doc2Vec embedding as input is a misspecification. Consequently, we expect the average model weight associated with v^f to be larger than v^g. Table 3.3 shows the performance of MEnKF-ANN in terms of coverage, the average width of prediction intervals, and average LSTM weight under two specifications of ensemble size (N) and initial ensemble variance (ν_x^2). To compare these results, we offer the coverage and average width of the prediction intervals when both the dropout layers are activated in the *fitted LSTM* during the prediction phase in Table 3.4. Observe how MEnKF-ANN recovered the *true probabilities* even better than the correctly specified LSTM with dropout. The average interval widths obtained from MEnKF-ANN are also lower than those from the *fitted LSTM*. These demonstrate the adequacy of MEnKF-ANN in approximating the target DL. Additionally, we observe that the average LSTM model weight is ≈ 1 indicating the ability of our approximator to identify the correctly specified data-generating model. Figure 3.6

shows the histogram of the predictive samples obtained from the ensemble members for eight test samples in a randomly chosen replicate. The red vertical line denotes the true logits, and the green vertical lines show the fences of the 95% prediction interval.

Simulation 2: Now, to demonstrate a situation where MEnKF-ANN is "inadequate," we supply the approximator with a completely different feature set representation. Instead of using the LSTM embedding v^f, we use Word2Vec [51] embedding of each gene in the predictor string and take the arithmetic average of these Word2Vec embeddings to represent the entire sequence. We denote this feature set by $\tilde{v}^f$ and then train the MEnKF-ANN using $\tilde{v}^f$ and v^g as the features and $logit(\tilde{\tilde{p}}^{(j)})$ as the target response. MEnKF-ANN is highly misspecified. Table 3.5 reports the coverage and average width of the prediction interval obtained from this model. Observing the huge width of the intervals essentially invalidates the point prediction. Such a large width indicates that MEnKF-ANN may not approximate the target DL. Therefore, we caution against using the coverage and width metrics to assess the "adequacy" of the *fitted LSTM* itself.

Simulation 3: We demonstrate how MEnKF-ANN can naturally handle model-averaged predictions. We train an ANN (with backpropagation) that takes Doc2Vec representation of gene sequences as predictors to estimate the probabilities $\hat{p}_{ANN}$. The true probabilities ($\hat{\bar{p}}$) are obtained by equally weighted average of $\hat{p}_{ANN}$ and the probabilities estimated by the LSTM ($\hat{p}_{LSTM}$, say). To attach uncertainty to $\hat{\bar{p}}$, we train the MEnKF-ANN by supplying LSTM embeddings and Doc2Vec embeddings to the two arms of MEnKF-ANN but use $logit(\hat{\bar{p}})$ as the target response here. Table 3.6 shows the performance of MEnKF-ANN in this situation for two combinations of N and ν_x^2. The coverage is measured with respect to $\hat{\bar{p}}$ on the test sets. Although the average width and MAE are larger than those reported in Table 3.7, we observe that the LSTM weights ≈ 0.5, which is what we would expect because MEnKF-ANN is

Table 3.3: Performance of MEnKF-ANN using LSTM embeddings and Doc2Vec

N	ν_x^2	Coverage	Width	LSTM weight
216	16	90.25%	0.33	0.9997
216	32	89.25%	0.32	0.9999

Table 3.4: Dropout-induced coverage and width of prediction intervals obtained from *fitted LSTM* with two dropout layers

Rate	Reps	Coverage	Width
0.5	50	81.25%	0.53
0.5	200	84.50%	0.56

Table 3.5: Performance of MEnKF-ANN using Word2Vec and Doc2Vec

N	ν_x^2	Coverage	Width	Word2Vec weight
216	16	96.25%	0.83	0.9155
216	32	94.25%	0.84	0.9787

Table 3.6: Performance of MEnKF-ANN trained to predict the averaged probability obtained using LSTM and ANN

N	ν_x^2	Coverage	Width	LSTM weight	MAE
433	0.2	90.75%	0.2661	0.5239	0.0609
433	0.3	91.00%	0.3274	0.5370	0.0667

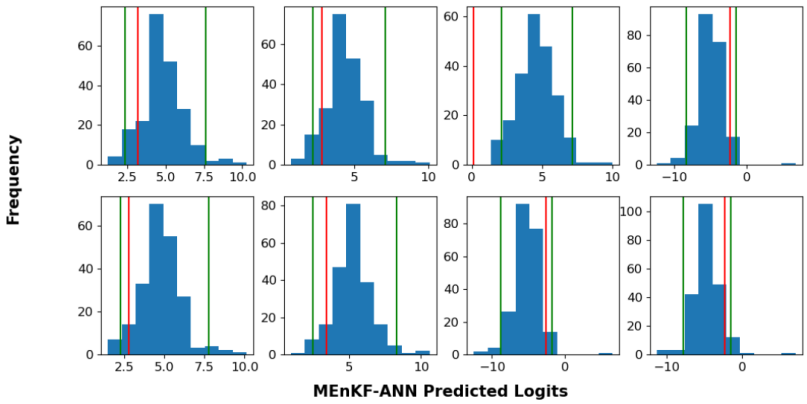

Figure 3.6: True logits superimposed on predicted logits from MEnKF-ANN using LSTM and Doc2Vec embeddings

seeing equally weighted outputs from LSTM and ANN.

3.5.3 Application II: Results on dbCAN-PUL data

We initialize the ensemble in the EnKF part of our model by drawing the members in the state vector (3.14) from $\mathcal{N}_{2n_g+2}(\mathbf{0}, \nu_x^2 I)$, where ν_x^2 is a tuning parameter that plays a crucial role in controlling the spread of the ensemble members and the dimension of I matches with the dimension of normal distribution. Following [12, 11], we assume the state transition is deterministic, i.e., $x_t = \phi_{t-1} x_{t-1}$ and hence we do not have the variance parameter corresponding to $\tilde{\eta}$ in the augmented state vector. When we reach the t^{th} batch of data, for the i^{th} member in the ensemble ($i = 1, 2, ..., N$), we update each element in the augmented state vector $w_t^{f,(i)}$, $w_t^{g,(i)}$, $a_t^{(i)}$, $b_t^{(i)}$ using the updating equation (2.12) suitably modified to handle deterministic state transition.

Our focal dataset consists of $n = 74$ samples belonging to Xylan and Pectin. However, training an LSTM on a small sample size would require aggressive regularization, even with this reduced label space. Therefore, we draw on an extensive collection of unlabelled data containing gene sequences associated with CAZyme gene clusters (CGC) computationally predicted from genomic data [29, 81]. Although this unlabelled data contains approximately 250K CGC gene sequences, unlike experimentally characterized PULs, these sequences do not have known carbohydrate substrate information. They hence cannot be directly used for classification purposes. We, therefore, use this unlabelled dataset to learn the Word2Vec embeddings of each gene appearing in the unlabelled dataset. These embeddings are then used to initialize the embedding layer of the target LSTM classifier.

Turning to the labeled dataset, instead of performing full cross-validation, we resort to subsampling procedure [56]. We take a subsample of sixty-six instances for training and hold eight instances for testing purposes. The subsample size (b) is chosen such that $b(n)/n \approx 8\sqrt{n}/n \to 0$, as $n \to \infty$. Although the subsampling

theory requires generating $\binom{n}{b}$ replicates, the computational cost for generating $\approx 10^{11}$ replicates, in our case, is prohibitive. Instead, we generate 50 independently subsampled replicates comprising training and testing sets of sizes 66 and 8, respectively. In each replication, two LSTMs ($LSTM_1$: two dropout layers - one in the LSTM layer and one before the final prediction layer with 50% dropout rate, and $LSTM_2$: one dropout layer in the LSTM layer with 50% dropout rate) are trained on the foregoing 66 training instances. Under this scheme, the probability that the i^{th} instance in our dataset appears at least once in the test set is $\approx 99.6\%$.

The LSTM-estimated probabilities of observing a *Pectin* substrate are extracted from $LSTM_1$ and $LSTM_2$ from each replicate. The average of these probabilities is logit transformed and used as the target response for our MEnKF-ANN approximator. We feed the embeddings of the gene sequences, obtained from $LSTM_1$ and $LSTM_2$, into the two arms of the MEnKF-ANN as two sets of features. We then generate predictions on the held-out test data points in each replicate. Finally, we compare the average LSTM prediction of probabilities with those generated by MEnKF-ANN predictions. The average MAE and the proportion of times a 95% prediction interval contains the LSTM-generated predictions in the held-out data set, under two different MEnKF-ANN hyperparameter choices are shown in Table 3.7 indicating that our approximator can be adequately used to generate the predictions. We do not report the LSTM weights estimated by MEnKF-ANN because, as we observed in the simulation (Table 3.3), the approximator overwhelmingly prefers the LSTM embeddings. Figure 3.7 shows the scatter plot of MEnKF-ANN-predicted and LSTM-predicted probabilities for the held-out data across 50 replicates. Figure 3.8 shows the boxplots associated with MEnKF-ANN predictions for the same set of test samples for which LSTM-generated prediction boxplots were shown in Figure 3.5. MEnKF-ANN can adequately

Table 3.7: Performance of MEnKF-ANN using $LSTM_1$ and $LSTM_2$ embeddings for dbCAN-PUL data

N	ν_x^2	Coverage	Width	MAE	CPU Time
108	2	83.00%	0.1029	0.0180	7.57 mins
108	6	85.00%	0.1183	0.0198	10.01 mins

Table 3.8: Comparison of the average width of prediction interval LSTM + MC dropout and MEnKF-ANN approximator for each LSTM

Target model	Average Width	Approximator	Average Width
$LSTM_1$	0.492	MEnKF-ANN_{11}	0.102
		MEnKF-ANN_{12}	0.085
$LSTM_2$	0.371	MEnKF-ANN_{21}	0.119
		MEnKF-ANN_{22}	0.108

approximate the target combination of LSTM.

Turning to the stability of prediction intervals, Table 3.8 shows the average width of the 95% prediction intervals obtained under individual base LSTMs. We activated the dropout layer(s) during prediction for each base learner and generated 200 predictions for each query instance. As discussed in the previous chapter, marginal versions of MEnKF-ANN are trained to approximate each based learner. MEnKF-ANN_{11} approximates $LSTM_1$ with 216 ensemble members and $\nu_x^2 = 16$, MEnKF-ANN_{12} also approximates $LSTM_1$, but now with 216 ensemble members and $\nu_x^2 = 32$. Similarly, MEnKF-ANN_{21} and MEnKF-ANN_{22} approximates $LSTM_2$ with 216 ensemble members and $\nu_x^2 = 16$ and $\nu_x^2 = 32$, respectively. Observe that the variation in the average width between $LSTM_1$ and $LSTM_2$ is considerably higher than the variation between MEnKF-ANN_{11} and MEnKF-ANN_{21} or between MEnKF-ANN_{12} and MEnKF-ANN_{22}. This indicates that the approximator produces more stable prediction intervals than obtaining prediction by activating the dropout layer during prediction.

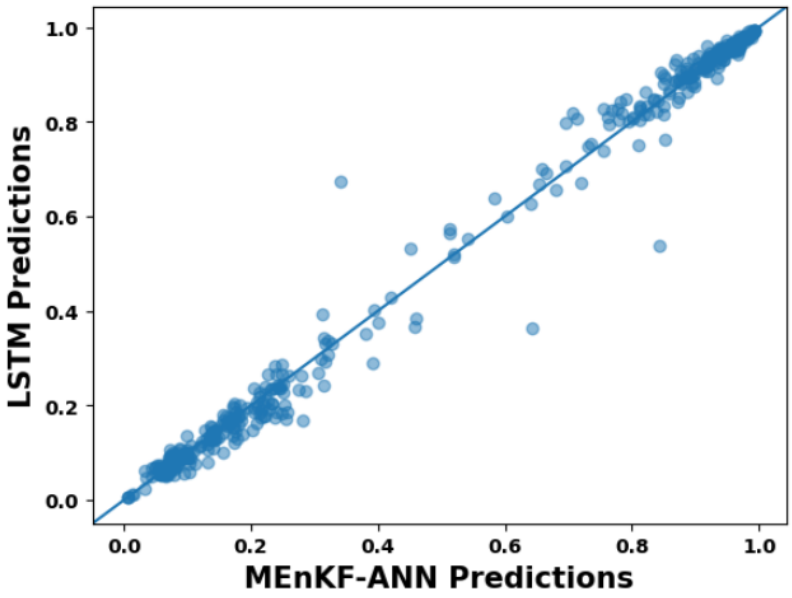

Figure 3.7: Scatterplot of MEnKF-ANN-predicted and LSTM-predicted probabilities for the test dataset

3.6 Conclusion

Our goal here was to develop the methodology of an EnKF-based multi-arm ANN that can simultaneously optimally estimate the *cross-learner* and *within-learner* weights and attach uncertainty to the model-averaged predictions. We designed the MEnKF-ANN to achieve that purpose. The augmented state space formulation allowed us to handle the non-linearity associated with neural networks. The matrix variate state variables allowed us to incorporate multiple learners and include all *cross-learner* and *within-learner* weights within the state matrix. The measurement model handled the model averaging aspect. As discussed in Chapter 2, the estimates of state variable produced by the KF technique are minimum mean square estimates. Since EnKF estimates asymptotically converge to their KF counterparts (see section 3.7), the estimates of the *cross-learner* and *within-learner* weights produced by our MEnKF-ANN are asymptotically jointly optimal under squared error loss. We re-iterate that such joint

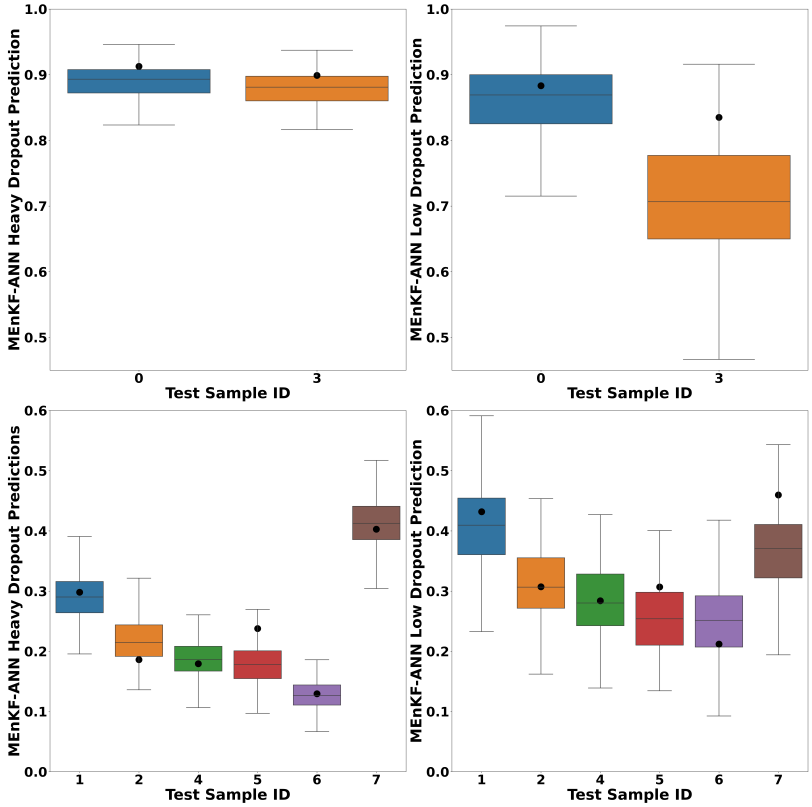

Figure 3.8: Boxplots showing the MEnKF-ANN predictions superimposed with the ground truth value for heavy and low dropout

optimality result is not available for the conventional stacking approach. Additionally, our EnKF construction automatically generates particles for predicted values of the response variable thereby allowing us to attach a statement of uncertainty associated with the model-averaged predictions. This constitutes the major methodological contribution of this chapter.

Turning to the application of MEnKF-ANN, we offered two illustrative examples. In the first example, we demonstrated that our technique could transfer information (to small datasets with potential covariate mismatch) from multi-arm deep learners trained on large datasets for regression tasks. We also demonstrated that the empirical coverage probability produced by our technique indicates it is an adequate predictive model. Quite surprisingly, when the fine-tuned deep learner was used to assimilate

the small dataset, the 95% empirical prediction intervals generated by activating the dropout layer contained the target values approximately 50% of the time. This raises questions about the reliability of the predictions generated by DL. As such, we posit that, since DLs are typically trained with dropouts, it is important to supply coverage and width of prediction interval along with the customary performance metrics of RSMPE and ρ.

Furthermore, we showed that the relative computational simplicity of our method allowed it to be retrained easily when new samples come in with a reduced set of features. This essentially indicates that MEnKF-ANN can be used as a vehicle to perform exhaustive ablation studies when the number of features is manageable. We emphasize that MEnKF-ANN is not a tool to perform feature imputation. If some of the input features are entirely missing in a dataset MEnKF-ANN simply switches off the arm that handles the set of missing features. If we encounter situations where values of the features are missing according to the classical MAR, NMAR setup, the current conceptualization of MEnKF-ANN cannot be directly applied.

State-augmented Kalman Filter and its variants provide a gradient-free method that can be extended to approximate popular neural network-based deep learners for regression and classification tasks. In the second application, our goal was to attach uncertainty to average probabilities generated by two different configurations of the LSTM binary classifier. We demonstrated how our method's in-built model averaging capability can be leveraged to attach uncertainty to these averaged predictions generated by two architectures. Our results suggest that this technique adequately captures the target probabilities to achieve coverage probabilities close to the nominal level. Since the domain of the target variable is bounded, we also see that the average width of the prediction interval is not too large to make these intervals meaningless. Our simulations suggest that the prediction intervals generated by our method are

less sensitive to the location and number of dropout layers in LSTM and hence provide more stable prediction intervals as compared to those obtained by activating the dropout layers within the LSTM itself. Admittedly, our procedure requires an additional round of training, but its fast computation time (see Table 3.7), along with its ability to emulate the approximand, adequately compensate for that. We used the information extracted from the dbCAN-PUL database and trained the base LSTMs to classify two carbohydrate substrates using the gene sequences characterized by the PULs of the gut microbiome and then used MEnKF-ANN to attach uncertainty to the predicted probabilities generated by a linear combination of two LSTMs. We anticipate this technique will be helpful to domain experts in assessing the reliability of predictions generated by an ensemble of learners.

3.7 Appendix

The size of the ensemble (N) and the batch size (which determines the number of updates T) are treated as tuning parameters. The following proposition offers a mathematically justifiable way to select these tuning parameters.

Proposition 1: If N and T are chosen such that the eigenvalues of $(I - K_T \mathcal{H}_T) < 1$, then the expected Kalman error attains steady state solution asymptotically.

To prove this proposition we first show that EnKF updates converge in probability to Kalman Filter updates. First, for notational convenience, we redefine some of the terms appearing in equations (2.5) - (2.10) that were derived from the state space model given by (2.1) and (2.2). Let the filtering distribution be represented by $x_t | y_{1:t} \sim \mathcal{N}(\hat{\mu}_{t|t}, \hat{\Sigma}_{t|t})$ where $\hat{\mu}_{t|t}$ and $\hat{\Sigma}_{t|t}$ are given by

$$\hat{\mu}_{t|t} := \hat{\mu}_{t|t-1} + K_t(y_t - H_t\hat{\mu}_{t|t-1}), \tag{3.15}$$

$$\hat{\Sigma}_{t|t} := \hat{\Sigma}_{t|t-1} - K_t H_t \hat{\Sigma}_{t|t-1}, \tag{3.16}$$

$$K_t := \hat{\Sigma}_{t|t-1} H_t^T (H_t \hat{\Sigma}_{t|t-1} H_t^T + R_t)^{-1} \tag{3.17}$$

The forecast distribution is given by $x_{t+1}|y_{1:t} \sim \mathcal{N}(\hat{\mu}_{t+1|t}, \hat{\Sigma}_{t+1|t})$ where $\hat{\mu}_{t+1|t}$ and $\hat{\Sigma}_{t+1|t}$ are given by

$$\hat{\mu}_{t+1|t} := M_t \hat{\mu}_{t|t}, \tag{3.18}$$

$$\hat{\Sigma}_{t+1|t} := M_t \hat{\Sigma}_{t|t} M_t^T + Q_t \tag{3.19}$$

In EnKF, N particles $x^1_{t|t'}, x^2_{t|t'}, \ldots, x^N_{t|t'}$ are drawn from either the forecast or the filtering distribution. So instead of estimating $\mu_{t|t'}$, the state estimates are given by

$$\overline{x}_{t|t'} = \frac{1}{N} \sum_{l=1}^{N} x^l_{t|t'} \tag{3.20}$$

with $t' = t$ or $t-1$ and the sample covariance matrix

$$\overline{\Sigma}_{t|t'} = \frac{1}{N-1} \sum_{l=1}^{N} (x^l_{t|t'} - \overline{x}_{t|t'})(x^l_{t|t'} - \overline{x}_{t|t'})^T \tag{3.21}$$

Let the distribution associated with the initial state be defined as $x_1|y_0 \sim \mathcal{N}(\mu_1, \Sigma_1)$. We initialize $x^l_1|y_0 \sim \mathcal{N}(\mu_1, \Sigma_1)$. Kalman Gain is estimated from the particles as

$$\overline{K_t} := \overline{\Sigma}_{t|t-1} H_t^T (H_t \overline{\Sigma}_{t|t-1} H_t^T + R_t)^{-1} \tag{3.22}$$

Particles are updated by the equation

$$x^l_{t|t} = x^l_{t|t-1} + \overline{K_t}(y^l_t - H_t x^l_{t|t-1}) \tag{3.23}$$

$$y^l_t \overset{\text{iid}}{\sim} \mathcal{N}(y_t, R_t) \tag{3.24}$$

Forecasted particles are given by

$$x^l_{t+1|t} = M_t x^l_{t|t} + \eta^l_t \tag{3.25}$$

$$\eta^l_t \overset{\text{iid}}{\sim} \mathcal{N}(0, Q_t) \tag{3.26}$$

Lemma 1. *Using the foregoing notations, as $N \to \infty$, the following holds*

a. $\overline{x}_{t|t} \xrightarrow{P} \hat{\mu}_{t|t}$

b. $\overline{\Sigma}_{t|t} \xrightarrow{P} \hat{\Sigma}_{t|t}$

c. $\overline{x}_{t+1|t} \xrightarrow{P} \hat{\mu}_{t+1|t}$

d. $\overline{\Sigma}_{t+1|t} \xrightarrow{P} \hat{\Sigma}_{t+1|t}$

Proof. See [42, 48] for compact proof of this Lemma 1.

Proof of Proposition 1: In our construction of MEnKF-ANN, when m_t is kept fixed, the state transition matrix $\phi_t = \phi$ remains constant in all updates (from (3.9)) and so does $\mathcal{H}_t$ after reparametrization (3.12). Now, from the time update step (immediately below (2.10)) we have $\Sigma_{t+1|t} = \phi\Sigma_{t|t}\phi' + Q$ (where Q is the covariance matrix for $\tilde{\eta}_t$). Now, using (2.9) to replace $\Sigma_{t|t}$, we have

$$\Sigma_{t+1|t} = \phi\Sigma_{t|t-1}\phi' + Q - \phi\Sigma_{t|t-1}\mathcal{H}'(\mathcal{H}\Sigma_{t|t-1}\mathcal{H}' + R)^{-1}\mathcal{H}\Sigma_{t|t-1}\phi'$$

KF theory uses this algebraic Ricatti equation to obtain the steady-state solution of $\Sigma_{t+1|t}$. Let $\hat{\Sigma}$ be the solution of the above equation, it can be interpreted as the steady state error covariance for estimating x_{t+1} conditional on $y_{1:t}$. To obtain $\hat{\Sigma}$ one can directly solve $\Sigma = \phi\Sigma\phi' + Q - \phi\Sigma\mathcal{H}'(\mathcal{H}\Sigma\mathcal{H}' + R)^{-1}\mathcal{H}\Sigma\phi'$ for Σ. We instead track the dominant eigenvalue of $(I - \tilde{K}_t\mathcal{H})$ because if the dominant eigenvalue of the above matrix is approximately 1, and the dominant singular value of $\tilde{K}_t\mathcal{H}_t$ converges to

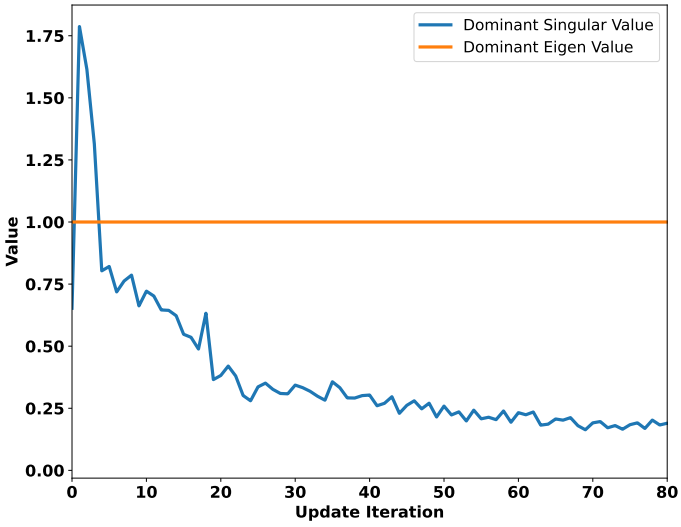

Figure 3.9: Trajectories of the singular values for the Kalman Gain matrix with the dominant eigenvalue of $I - K_t\mathcal{H}_t$

0, then using (2.9), we will have $\Sigma_{t|t} \approx \Sigma_{t|t-1}$ which essentially yields the stationary solution of the Ricatti equation. So, our reason for jointly tracking the trajectories of dominant eigenvalue of $(I - \tilde{K}_t\mathcal{H}_t)$ and dominant singular value of $\tilde{K}_t\mathcal{H}_t$ is to visually assess whether the EnKF has achieved an approximate steady-state. Figure 3.9 shows the foregoing trajectories empirically. □

Choosing N and m_t: Once steady-state is achieved, we look into the expression of $\mu_{t+1|t}$ (given below (2.10)) with $\hat{\Sigma}$ replacing $\Sigma_{t|t-1}$. Then, in steady-state we have

$$\begin{aligned}\mu_{t+1|t} &= \phi\mu_{t|t-1} + \phi\hat{\Sigma}\mathcal{H}'(\mathcal{H}\hat{\Sigma}\mathcal{H}' + R)^{-1}(y_t - \mathcal{H}\mu_{t|t-1}) \\ \mu_{t+1|t} &= \phi\mu_{t|t-1} + L(y_t - \hat{y}_{t|t-1})\end{aligned}$$

where $L = \phi\hat{\Sigma}\mathcal{H}'(\mathcal{H}\hat{\Sigma}\mathcal{H}' + R)^{-1}$ and $\hat{y}_{t|t-1}$ is predicted value of the observation given $y_{1:t-1}$. Define the state estimation error $e_{t|t-1} = x_t - \mu_{t|t-1}$ and the observation

prediction error to be $e^{(y)}_{t|t-1} = y_t - \hat{y}_{t|t-1}$. Then we have the following error dynamics:

$$\begin{aligned} e_{t+1|t} &= x_{t+1} - \mu_{t+1|t} \\ &= \phi x_t + \tilde{\eta}_t - \phi\mu_{t|t-1} + Le^{(y)}_{t|t-1} \\ &= \phi(x_t - \mu_{t|t-1}) + Le^{(y)}_{t|t-1} + \tilde{\eta}_t \\ &= \phi e_{t|t-1} + Le^{(y)}_{t|t-1} + \tilde{\eta}_t \end{aligned} \tag{3.27}$$

Now, observe $\phi = I$ in our construction then from (3.27) we have $e_{t+1|t} - e_{t|t-1} = Le^{(y)}_{t|t-1} + \tilde{\eta}_t$. Now suppose $E(e^{(y)}_{t|t-1}) = C$, where C is a $m_t \times 1$ vector whose each coordinate informs us about the expected one-step ahead prediction error at the observation scale. Suppose the elements of C are constants. Then, for each coordinate in the expected state estimation error vector, we have $E(e_{t+1|t} - e_{t|t-1}) = LC$. Now, under steady-state L is a constant over t, and therefore the expected difference in state estimation error between two successive updates is a linear combination of expected prediction error in the observation scale. Thus, if $E(e^{(y)}_{t|t-1})$ stabilizes in each coordinate, so does $E(e_{t+1|t} - e_{t|t-1})$

We exploit this observation to elicit the tuning parameters. We would like to estimate $E(e^{(y)}_{t|t-1})$ in steady state. So, we train multiple MEnKF-ANN with different values of N and m_t. We plot the trajectory of the one-step ahead training RMSE over the update iterations as a proxy for observation prediction error. Once we see the RMSE trajectories are converging we can reasonably expect $E(e^{(y)}_{t|t-1})$ to be approximately the same for each trajectory. Then our choosing criterion is given by the following: Out of all the trajectories entering the convergence region, choose the (N, m_t) combination that has the smallest N satisfying $N > m_t$.

Chapter 4

Scalability of Matrix Ensemble Kalman Filter-based stacker for combining two multi-arm deep learners

4.1 Introduction

Accurate cancer drug response (CDR) prediction has become a central problem in computational cancer-pharmacogenomics. Such computational models can potentially identify molecular signatures that determine CDR, at least *in-silico* setup, thereby offering guidance to anti-cancer drug discovery. Although several computational models exist for predicting CDR, deep learning models (DLs) have achieved state-of-the-art status because of their ability to capture the intrinsic chemical structure of drugs and integrate multi-omics data. Since there exist several DLs for predicting CDR (see [6, 13, 5] for a review of deep learning models developed for predicting CDR), an ensemble of deep learning models is immediately available that can potentially increase the overall robustness of the predictive method. However, how to quantify the uncertainty in the model ensemble?

Uncertainty quantification in individual DLs has been studied. Bayesian neural networks (BNN) and Bayesian deep learners have emerged as the default techniques for assessing uncertainty in DL predictions. These methods learn the posterior distribution of training weights, thereby producing an uncertainty estimate of the model output

[70, 71, 32]. Instead of sampling from the exact posterior distribution, variational inference and Laplacian approximation methods approximate the posterior distribution over BNN weights [39, 52, 66, 60, 63]. The famous Monte Carlo dropout technique also approximates the posterior distribution by running the trained network multiple times with a fraction of nodes randomly switched off every run, thereby producing a distribution of predictions. As an alternative to Bayesian procedures, [40] developed a frequentist method to quantify uncertainty in the DL estimates.

Ensemble techniques have also been used to quantify uncertainty in deep learners. Typically, such a technique requires the DLs to be trained under various choices of hyperparameters, thereby producing a *distribution* of predictions for the targets. However, these distributions are not as easily interpretable or probabilistically coherent as a posterior predictive distribution emerging from a Bayesian setup. To address this issue [43] introduced the nonparametric Bayes ensembling technique that can decompose various sources of uncertainty and generate point prediction and uncertainty interval in a theoretically rigorous fashion. Deep ensembling methods that provide accurate prediction intervals with acceptable coverage probability and reasonable width [57, 28] are also available. As such, [73, 21, 54] showed that a Bayesian conceptualization of deep ensemble yielded impressive accuracy and robustness. We direct the audience to [1, 50] for a more detailed review of uncertainty quantification in deep learners.

Observe that all the foregoing techniques are either geared to attach uncertainty to a focal DL or produce a stylized ensemble of models with an explicit goal of attaching uncertainty to predictions. Our scenario is slightly different because we have a catalog of base DLs trained to predict CDR, and we would like to develop a suitable technique to combine the predictions generated by the catalog of base learners and attach uncertainty to these predictions. Consequently, traditional model

averaging techniques, for instance, Bayesian Model Averaging [23, 47] or stacking [75], are more relevant for our purpose. However, neither of these techniques can be directly used to attach uncertainties because the available base DLs rarely characterize the probabilistic aspect of the weights and errors. Hence standard Bayesian Model Averaging may not be immediately implemented, and standard model stacking does not propagate the uncertainties in the output of the base learners [75, 34].

This chapter extends our generalized stacking approach, first proposed in [55], to suit the problem. In particular, we stack two recently developed DLs, DualGCN [46] and DeepCDR [44] - that predict CDR using chemical structures of cancer drugs and multi-omics data associated with the cell lines on which the drug screen is administered. We demonstrate how we can attach uncertainty, in a probabilistically coherent way, to predictions generated by stacking DeepCDR and DualGCN. We also show how this stacker can quickly adapt to attach uncertainty to a single base learner. Finally, we show how sequential training of our stacker can be used to handle large datasets, thereby demonstrating our approach's scalability.

The remainder of the chapter is organized as follows: Section 4.2 offers a brief description of the focal base learners, DeepCDR and DualGCN, along with a description of the data on which these models are trained. Section 4.3 details the construction of our generalized stacker. We reveal the result of stacking DeepCDR and DualGCN for predicting CDR and attaching uncertainty to these predictions in section 4.4. Finally, we offer concluding remarks in section 4.5.

4.2 Background

In the CDR prediction domain, DualGCN and DeepCDR are two multi-arm deep learning models that use the chemical structure of drugs and multi-omics profiles of cell

lines as inputs to predict the drug response, captured via logarithm of half-maximal inhibitory concentration ($\log IC_{50}$).

Both models process the drug features by using a Graph Convolutional Network (GCN) to leverage the graph-like representation typically used to describe the chemical structure of molecules. Each atom within a drug molecule is represented by a 75-dimensional feature vector [59] and an adjacency matrix that represents the connections among the atoms. Turning to the omics profile - DeepCDR extracts three types of omics features - genomic (genomic mutation), transcriptomic (gene expression), and epigenomic (DNA methylation) - for each cancer cell line from the CCLE database [7]. Transcriptomic and epigenomic features are processed using fully connected networks, and the genomic features are processed using a 1-D convolutional network because the chromosomes in the genomic features are believed to encode sequential information. DualGCN, on the other hand, uses gene expression and Copy Number Variation (CNV) to represent the omics profile of the cancer cell lines. This model extracts gene expression and CNV information associated with 697 genes from the CCLE database and obtains the adjacency matrix representing the association among these 697 genes from the STRING database [67]. In summary, the model architecture for two base deep learners are different - DeepCDR processes the multi-omics features using a 1-D Convolutional Neural Network (CNN) and the drug features using a GCN, DualGCN processes both multi-omics and drug features using two different GCNs.

The target response values ($\log IC_{50}$) corresponding to the drug-cell line combinations are extracted from the GDSC database [76]. A total of 86530 response values, associated with 208 unique drugs and 525 unique cancer cell lines, along with the chemical structures of the said drugs and the multi-omics profile of associated cell lines, form the dataset we analyze in this chapter. The base DLs use 80% of the instances for training purposes and the remaining 20% as validation samples to

determine early stopping. From the trained DeepCDR and DualGCN models, we extract the embeddings for the drugs and the multi-omics branch for both the training and the validation samples. In their original form, DualGCN learned 128 and 256 dimensional embeddings for the drugs and the multi-omics features, respectively. In contrast, the DeepCDR learned 100-dimensional embeddings for each omics feature and drug type. In this chapter, we modified the architecture of DeepCDR to make the output embedding dimensions match those produced by DualGCN. Both base learners are trained with dropout layers, but these layers are not activated during the prediction stage, so neither method directly produces any prediction intervals.

4.3 Methodology

Our generalized stacker uses a Matrix Ensemble Kalman Filter to train a multi-arm artificial neural network (MEnKF-ANN). In this protocol, a multi-arm ANN ingests different types of predictors, and the augmented state vector associated with the Ensemble Kalman Filter performs the stacking and updates the weights associated with the neural network, model weights, and variance parameters. The original construction of MEnKF-ANN for the two-arm neural network can be found in [55] along with the training algorithm. However, we need to extend the construction posited in [55] because we require the current stacker to ingest the drug embeddings and omics embeddings from DeepCDR and DualGCN in four arms so that we can determine the relative importance of each of these embeddings. Observe that identifying the importance can offer insight into which types of embeddings should be used to represent the omics profile (collection of genomic, transcriptomic, and epigenomic or gene expression and CNV).

We, therefore, begin with a generic construction of our MEnKF-ANN stacker

that can simultaneously train four different neural network architectures and perform *in-situ* stacking. Then, we show how this construction can leverage the learned embeddings from the base learners. We define our target responses as $Y \in \mathcal{R}$. We have a total of $m = \sum_{t=1}^{T} m_t$ training instances, with m_t being the number of training data points in the t^{th} batch. Consider two bi-arm DLs generically denoted as $f_{multi} = f_{multi}(\boldsymbol{U}_1^{(1)}, \boldsymbol{U}_2^{(1)})$ and $g_{multi} = g_{multi}(\boldsymbol{U}_1^{(2)}, \boldsymbol{U}_2^{(2)})$ that map the input features $\{\boldsymbol{U}_1^{(1)}, \boldsymbol{U}_2^{(1)}\}$ and $\{\boldsymbol{U}_1^{(2)}, \boldsymbol{U}_2^{(2)}\}$ to the response space, respectively. Let $v_t^{f_1} \in \mathcal{R}^{p_1}$ and $v_t^{f_2} \in \mathcal{R}^{q_1}$ denote two different learned embeddings extracted from two arms of the DL f_{multi} for the t^{th} batch of data. Similarly, let $v_t^{g_1} \in \mathcal{R}^{p_2}$ and $v_t^{g_2} \in \mathcal{R}^{q_2}$ denote the learned embeddings extracted from two arms of the DL g_{multi} for the t^{th} batch of data. We will assume that the embeddings extracted from the base DLs are of the same dimension, i.e., $p_1 = q_1 = p_2 = q_2$. Let f_1^*, f_2^*, g_1^*, g_2^* denote the architectures for the four ANNs that take v^{f_1}, v^{f_2}, v^{g_1}, v^{g_2} as inputs, respectively, and connect them to the response Y. All four ANN architectures $(f_1^*, f_2^*, g_1^*, g_2^*)$ have a single hidden layer of fixed size and a final prediction layer with one neuron. Thus, each ANN has the same number of learnable parameters, denoted by n_{multi}. Let $w_t^{f_1^*}$, $w_t^{f_2^*}$, $w_t^{g_1^*}$, and $w_t^{g_2^*}$ denote the updated weights corresponding to the ANNs f_1^*, f_2^*, g_1^*, and g_2^*, respectively, using the t^{th} batch of data. Let $a_t^{f_1^*}$, $a_t^{f_2^*}$, $a_t^{g_1^*}$, $a_t^{g_2^*}$ and b_t be real-valued scalar parameters and let $\boldsymbol{s}_t = [s_t^{f_1^*}, s_t^{f_2^*}, s_t^{g_1^*}, s_t^{g_2^*}] = [\sigma(a_t^{f_1^*}), \sigma(a_t^{f_2^*}), \sigma(a_t^{g_1^*}), \sigma(a_t^{g_2^*})]$ where $\sigma(.) : \mathcal{R}^d \to [0,1]^d$ is the usual *softmax* function.

4.3.1 The MEnKF-ANN stacker for multi-arm DLs

First, consider the state matrix of the matrix Kalman Filter, X_t, associated with the t^{th} batch of data:

$$X_t^{(m_t+n_{multi}+2)\times 4} = \begin{bmatrix} s_t^{f_1^*} f_1^*(v_t^{f_1}, w_t^{f_1^*}) & s_t^{f_2^*} f_2^*(v_t^{f_2}, w_t^{f_2^*}) & s_t^{g_1^*} g_1^*(v_t^{g_1}, w_t^{g_1^*}) & s_t^{g_2^*} g_2^*(v_t^{g_2}, w_t^{g_2^*}) \\ w_t^{f_1^*} & w_t^{f_2^*} & w_t^{g_1^*} & w_t^{g_2^*} \\ a_t^{f_1^*} & a_t^{f_2^*} & a_t^{g_1^*} & a_t^{g_2^*} \\ b_t & 0 & 0 & 0 \end{bmatrix} \tag{4.1}$$

Define $H_t^{m_t\times(m_t+n_{multi}+2)} = [I_{m_t}, 0_{m_t\times(n_{multi}+2)}]$, $G_t^{4\times 1} = [1,\ 1,\ 1,\ 1]^T$, $\Theta_{t-1} = I_{m_t+n_{multi}+2}$, and $\psi_{t-1} = I_4$. We can now define the measurement equation of a Matrix State Space Model as:

$$Y_t = H_t X_t G_t + \epsilon_t, \tag{4.2}$$

with the state evolution equation being

$$X_t = \Theta_{t-1} X_{t-1} \psi_{t-1} + \eta_t. \tag{4.3}$$

where ϵ_t and η_t are mutually independent zero-mean Gaussian error terms. In particular, we assume $\epsilon_t \sim \mathcal{N}_{m_t}(0, \gamma_y^2 I_{m_t})$ where $\gamma_y^2 = \log(1 + e^{b_t})$. We will discuss η_t in the context of efficient computation in section 4.3.2. Writing in *vec* format, (4.3) becomes

$$x_t = vec(X_t) = (\psi_{t-1}^T \otimes \Theta_{t-1}) vec(X_{t-1}) + vec(\eta_t) \tag{4.4}$$

and letting $\phi_{t-1} = \psi_{t-1}^T \otimes \Theta_{t-1}$ and $\tilde{\eta}_t = vec(\eta_t)$ we get from (4.4)

$$x_t = \phi_{t-1} x_{t-1} + \tilde{\eta}_t \tag{4.5}$$

(4.2) can similarly be compactified as

$$Y_t = \mathcal{H}_t x_t + \epsilon_t \tag{4.6}$$

where $\mathcal{H}_t = G_t^T \otimes H_t$. It is easy to verify that such a construction does explicit model averaging by expanding the $H_t X_t G_t$ in (4.2).

$$H_t X_t G_t = \left[s_t^{f_1} f_1(v_t^{f_1}, w_t^{f_1}) + s_t^{f_2} f_2(v_t^{f_2}, w_t^{f_2}) + s_t^{g_1} g_1(v_t^{g_1}, w_t^{g_1}) + s_t^{g_2} g_2(v_t^{g_2}, w_t^{g_2}), \right] \tag{4.7}$$

where the convex model weights $s_t^{f_1^*} + s_t^{f_2^*} + s_t^{g_1^*} + s_t^{g_2^*} = 1$ yield a weighted average of the predictions from the four ANNs. Recall that the inputs features for $f_k^*, g_k^*, (k = 1, 2)$ are extracted embeddings $v_t^{f_1}$, $v_t^{f_2}$, $v_t^{g_1}$, $v_t^{g_2}$ from the two base learners f_{multi} and g_{multi}. Therefore, by choosing X_t as in (4.1), we train an ensemble model and perform stacking *in-situ*, using the embeddings extracted from the trained multi-arm DLs.

We retain the scalar parameter b_t in (4.1), used to estimate the variance of the error term in the measurement equation (4.6), as a learnable parameter, and the remaining variance parameters are treated as tuning parameters.

4.3.2 An efficient solution for MEnKF-ANN stacker

The formulation of the state matrix X_t as in (4.1) brings up an interesting problem regarding the solution of the linear state space model. The standard solution of the model outlined in (4.5) and (4.6) requires the computation of the Kalman Gain term K_t. In Ensemble Kalman Filter the Kalman Gain is estimated by $\hat{K}_t = \tilde{S}_t \mathcal{H}_t^T (\mathcal{H}_t \tilde{S}_t \mathcal{H}_t^T + \sigma_y^2 I_{m_t})^{-1}$ where $\tilde{S}_t$ is the sample covariance matrix of the forecasted ensemble. In our case, $\mathcal{H}_t$ is $m_t \times 4(m_t + n_{multi} + 2)$ dimensional and $\tilde{S}_t$ has the same dimension as the covariance matrix of x_t in (4.4), i.e., $dim(\tilde{S}_t) = 4(m_t + n_{multi} + 2) \times 4(m_t + n_{multi} + 2)$. Consequently, both $\tilde{S}_t \mathcal{H}_t^T$ and $\mathcal{H}_t \tilde{S}_t \mathcal{H}_t^T$ are of relatively large dimensions and computation of Kalman Gain requires inverting and multiplying such large matrices in every update step, making the algorithm computationally expensive and increases

the memory requirement considerably.

However, closer inspection reveals that, unlike the standard Kalman Filter, not all elements of X_t need to be estimated in our case. Only w_t's, a_t's, and b_t need to be estimated. The top $m_t \times 4$ elements of X_t are deterministic functions of w_t, a_t, and v_t. Thus, instead of updating the entire X_t in a naive fashion, we can focus only on the unknown quantities. Therefore, following [38], we obtain the Ensemble Kalman Filter solution to a least squares minimization problem given by $\Phi(u, Y) = \|\mathcal{G}(u) - Y\|_\Gamma^2$, where u is the vector of unknown parameters that need to be estimated and $\mathcal{G}$ is any model architecture, Y is the target, and $\Gamma = Cov(Y)$. Following [77] we envision the set of particles $U^n = \{u_j^n\}_{j=1}^N$ as N estimates for the unknown parameter u at the n^{th} iteration. These estimates are recursively updated using the formula

$$u_j^{n+1} = u_j^n + C(U^n)(D(U^n) + \Gamma)^{-1}(y - \mathcal{G}(u_j^n)) \tag{4.8}$$

where $C(U) = \frac{1}{N}\sum_{j=1}^N (u_j - \bar{u}) \otimes (\mathcal{G}(u_j) - \bar{\mathcal{G}})^T$, $D(U) = \frac{1}{N}\sum_{j=1}^N (\mathcal{G}(u_j) - \bar{\mathcal{G}}) \otimes (\mathcal{G}(u_j) - \bar{\mathcal{G}})^T$, $\bar{u} = \frac{1}{N}\sum_{j=1}^N u_j$, $\bar{\mathcal{G}} = \frac{1}{N}\sum_{j=1}^N \mathcal{G}(u_j)$. Evidently, (4.8) only updates the unknown elements in the state matrix, thereby reducing the computational cost associated with solving the model described in section 4.3.1. Notice that in (4.8) y is the observed target vector. In contrast, the Ensemble Kalman Filter solution proposed in [33] uses perturbed target where they perturb y with a random noise vector z drawn from $\mathcal{N}(0, \Gamma)$. Therefore, the modified updating equation becomes

$$u_j^{n+1} = u_j^n + C(U^n)(D(U^n) + \Gamma)^{-1}(y + z - \mathcal{G}(u_j^n)) \tag{4.9}$$

More precisely, we define $u_t = [w_t^{f_1^*},\ w_t^{f_2^*},\ w_t^{g_1^*},\ w_t^{g_2^*},\ a_t^{f_1},\ a_t^{f_2},\ a_t^{g_1},\ a_t^{g_2},\ b_t]$ and reparameterize x_t in (4.4) by writing it in terms of u_t. Now, replacing $\mathcal{G}(u)$ with $\mathcal{H}_t x_t$ we can use (4.9) to solve the minimization problem $\Phi(x_t, Y_t) = \|\mathcal{H}_t x_t - Y_t\|_\Gamma^2$ to solve

the state space model defined in (4.5) and (4.6).

To obtain the EnKF solution to the minimization problem, we first draw N samples of $U_t^0 = \{u_{j,t}^0\}_{j=1}^N$, with $u_{j,t} = [w_{t,j}^{f_1^*},\ w_{t,j}^{f_2^*},\ w_{t,j}^{g_1^*},\ w_{t,j}^{g_2^*},\ a_{t,j}^{f_1^*},\ a_{t,j}^{f_2^*},\ a_{t,j}^{g_1^*},\ a_{t,j}^{g_2^*},\ b_{t,j}]$ from $\mathcal{N}_d(0, S_d)$, where $d = dim(u_{j,t}) = 4n_{multi} + 5$. The covariance matrix S_d is parameterized using three different variance terms σ^2_{ANNs}, $\sigma^2_{AvgWeights}$, and $\sigma^2_{TargetVar}$ corresponding to the variances for the weights associated with $f_1^*, f_2^*, g_1^*, g_2^*$, model averaging weights, and variance associated with the covariance matrix of the response, respectively.

$$S_d = \begin{bmatrix} \sigma^2_{ANNs} I_{4n_{multi}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \sigma^2_{AvgWeights} I_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \sigma^2_{TargetVar} I_1 \end{bmatrix} \tag{4.10}$$

These N particles of U^0 are then updated using the adapted version of (4.9) given by

$$u_{j,t}^{n+1} = u_{j,t}^n + C(U^n)(D(U^n) + \sigma^2_{y_t} I_{m_t})^{-1}(y_t + z_{j,t}^n - \mathcal{H}_t x_{t,j}^n) + \eta_{u;j,t} \tag{4.11}$$

with $\sigma^2_{y_t} = log(1 + e^{\frac{1}{N}\sum_{j=1}^N b_{t,j}^n})$, $z_{j,t}^n \sim \mathcal{N}(0, \sigma^2_{y_t} I_{m_t})$, $C(U) = \frac{1}{N}\sum_{j=1}^N (u_{j,t} - \bar{u}) \otimes (\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}})^T$, $D(U) = \frac{1}{N}\sum_{j=1}^N (\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}}) \otimes (\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}})^T$, $\bar{u} = \frac{1}{N}\sum_{j=1}^N u_{j,t}$, $\bar{\mathcal{H}} = \frac{1}{N}\sum_{j=1}^N \mathcal{H}_t x_{t,j}$ and $\eta_{u;j,t} \sim N(0, \sigma^2_{fudge} I_d)$ are Gaussian perturbations introduced to capture the stochasticity of the state equation (4.5). Once again, we treat all the foregoing variance parameters $(\sigma^2_{ANNs}, \sigma^2_{AvgWeights}, \sigma^2_{TargetVar}, \sigma^2_{fudge})$ as tuning parameters without explicitly estimating them.

We reiterate that S_d supplies the values of the variance parameters associated with the initial distributions of *within-learner weights* (w), *cross-learner* weights (a) and target variance (b). Thus, S_d controls the *spread* of the initial realization of the elements in the state matrix. As we mentioned in Chapter 2, the EnKF operation is

purely feed-forward, and hence, we do not consider a formal posterior of S_d. Instead, the elements in S_d are treated as tuning parameters and are chosen via tracking the validation error for multiple runs of the MEnKF-ANN under various choices of the foregoing tuning parameters.

4.3.3 Connecting MEnKF with DualGCN and DeepCDR

Instead of directly using the raw embeddings extracted from the DeepCDR and DualGCN, we perform a dimension reduction by running a PCA on these embeddings separately. We extract 64 PCs for each of DeepCDR-drugs, DeepCDR-omics, DualGCN-drugs, and DualGCN-omics. These four sets of PCs are passed as input features to the four ANNs f_1^*, f_2^*, g_1^*, and g_2^* that form the core of MEnKF-ANN stacker - as described in Section 4.3.1. All these ANNs have feed-forward neural network architectures with one hidden layer having eight neurons and a final prediction layer with one neuron. The batch size m_t is chosen *a-priori* depending on the training data size. Given the configuration of f_1^*, f_2^*, g_1^*, g_2^* and the size of the feature set, we have $n_{multi} = 529$ parameters that need to be estimated for each ANN. We, therefore, need to estimate $d = 4n_{multi} + 5 = 2121$ parameters in each batch. At $t = 0$, we generate N particles $U^0 = \{u_j^0\}_{j=1}^N$ from $\mathcal{N}_{2121}(0, S_{2121})$ to initialize the EnKF solution. We then compute X_t, H_t, G_t, and $\mathcal{H}_t$ as described in Sections 4.3.1, 4.3.2 for $t = 0$. The updated set of parameters $U^t, t = 1, 2, ...$ are obtained using the updating equation (4.11).

Observe that MEnKF-ANN explicitly estimates four weights corresponding to embeddings associated with DeepCDR-drugs, DeepCDR-omics, DualGCN-drugs, and DualGCN-omics, respectively. These weights assess the relative importance of each type of feature coming from the constituent deep learners. Now, for example, to estimate the weight for the DeepCDR model, we can sum $s^{f_1^*}$ and $s^{f_2^*}$. Similarly, to

estimate the weights for the drug features, we can sum $s^{f_1^*}$ and $s^{g_1^*}$. Additionally, the EnKF construction produces a range of predictions for each query point, enabling us to compute the prediction intervals empirically.

4.4 Application

We train the MEnKF-ANN stacker on the same 69214 data points used to train the base learners. We retain the remaining 17316 instances for test purposes. The stacker is trained sequentially on approximately equal-sized $T = 28$ batches, each with approximately $m_t = 2500$ samples. After each update of the MEnKF-ANN, we obtain the root mean squared training error computed on the entire training set. We stop training the stacker if both conditions are satisfied: (a) training RMSE only improves over ten successive batch updates, and (b) the MEnKF-ANN has run for at least one epoch. To assess sample size's impact on the stacker's predictive capability, we evaluate the root mean square prediction error (RMSPE) on the foregoing 17316 test samples after training over each batch of data. We expect that the predictive capability of MEnKF-ANN will improve as more data batches are added to the training set. We use the following hyper-parameter configuration for training MEnKF-ANN: $N = 106$, $\sigma^2_{fudge} = 0.01$, $\sigma^2_{ANNs} = 1$, $\sigma^2_{AvgWeights} = 1$, and $\sigma^2_{TargetVar} = 1$. To assess the sensitivity of the stacker on the initialization of ensemble particles (U^0), we train and evaluate the MEnKF-ANN 50 times under different realizations of U^0.

The final prediction is obtained by averaging over the 50 runs of MEnKF-ANN. RMSPE and the Pearson correlation coefficient (ρ) between the observed test values and predicted values are computed using this initialization-averaged prediction. Since the forecast distribution of each particle in the ensemble Kalman Filter is available in closed form, the width of the prediction interval and the coverage probability

associated with 95% prediction intervals are calculated using the quantile averaging technique [17, 18, 16] for each test point. We report RMSPE, ρ, coverage probability, and average width of the quantile-averaged prediction interval in Table 4.1. However, to demonstrate the benefit of stacking, we need to compare these metrics to those obtained from individual base learners. However, the vanilla base learners can only offer RMSPE and ρ. Hence, to make all candidate models comparable, we deploy the MEnKF-ANN to approximate each base learner by simply requiring the state matrix in (4.3) to contain two columns arising from either f_1^*, f_2^* or g_1^*, g_2^*. The MEnKF-ANN approximated individual base learners' prediction performance is also included in Table 4.1. To benchmark the performance of MEnKF-ANN, we train a multi-arm deep learner (joint learner) that combines the original construction of the subnetworks in DeepCDR and DualGCN.

First, the results show that the joint learner's performance (in terms of RMSPE and ρ) is almost the same as those reported in the original articles that introduced DualGCN and DeepCDR. Therefore, our joint learner reliably replicates previously published results and can be used for benchmarking. The MEnKF-ANN stacked learner has higher RMSPE and smaller ρ. However, the coverage probabilities associated with the joint leaner, obtained from the dropout-induced prediction interval, indicate its inability to achieve anything close to the nominal level. Our method consistently achieves a nominal level of coverage, indicating its adequacy in processing the uncertainty arising from the models.

We also observe that the stacked model outperforms the individual base learners with respect to RMSPE and ρ, thereby underscoring the utility of model averaging in improving the prediction accuracy in the CDR domain. The coverage probability of the stacked model practically achieves the nominal level, thus offering confidence in the predictions generated by the MEnKF-ANN. We observe that DeepCDR coupled

with MEnKF-ANN offers greater coverage but at the cost of a larger width, but the DualGCN-MEnKF-ANN combination offers lower coverage with a tighter width of the prediction interval. However, the stacked model provides a balance between the base learners. We also note that by attaching an inferential mechanism to DeepCDR and DualGCN, we enhance the statistical support of the predictions generated by these DLs.

Table 4.1: Performance metrics of MEnKF-ANN using DeepCDR and DualGCN embeddings

Models	RMSPE	Coverage	Width	ρ
Joint Learner	1.10	61.14%	1.85	0.92
DeepCDR-MEnKF-ANN	1.62	97.82%	8.36	0.82
DualGCN-MEnKF-ANN	1.60	93.48%	6.39	0.82
MEnKF-ANN stacked DualGCN & DeepCDR	1.37	95.71%	5.99	0.88

To further contextualize the precision of the predictions and to visually assess the predictive performance of MEnKF-ANN, we display, in Figure 4.1, the scatter plot between the MEnKF-ANN predictions and the observed values of $\log IC_{50}$ in the test set. The plot suggests no obvious bias in the predictions generated by MEnKF-ANN. Furthermore, even though the coverage probability exceeds the nominal level, given the range of the observed test data, we can conclude that the average width of the prediction intervals is not too large to render the prediction intervals meaningless. Together, we can surmise that the prediction intervals generated by MEnKF-ANN are reliable.

To get a detailed picture of the reliability of the prediction intervals, in Figure 4.2, we show the line plot of the quantile-averaged upper 97.5% and lower 2.5% prediction limits along with the observed values of the test points (in solid circles). We observe that the prediction interval covers the query points adequately. However, the coverage deteriorates in the extreme upper and lower tail regions of the data.

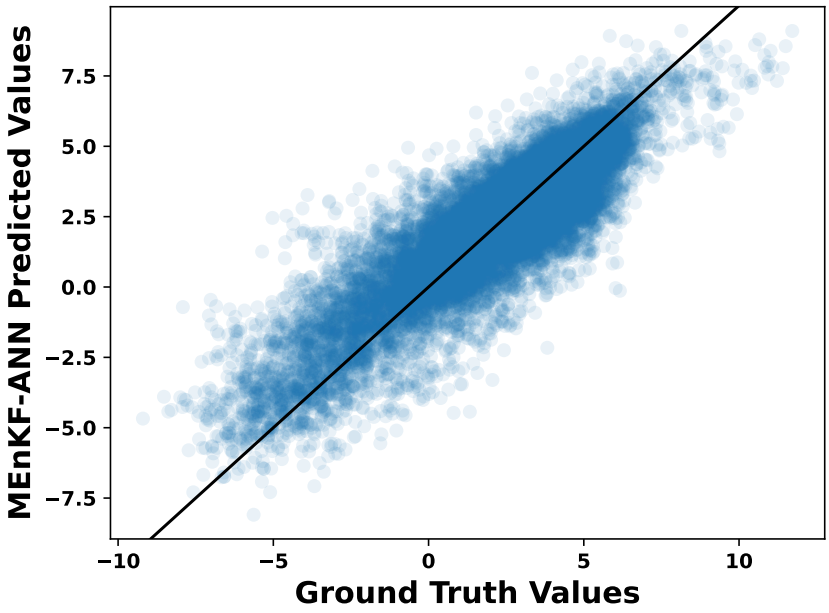

Figure 4.1: Scatterplot for MEnKF-ANN predictions with the observed log IC_{50} values in the test set.

This underscores the hazard of making predictions on the boundary or performing extrapolation with models that are not explicitly designed to capture the quantiles of the response variables.

To demonstrate the effect of ensemble size, we plot the trajectories of mean training RMSE and RMSPE across the batch updates, along with the point-wise standard deviation in Figure 4.3. The MEnKF-ANN method can learn quickly from the data and reach the best RMSE within the first few updates. The variation in the mean square errors also becomes negligible when MEnKF-ANN reaches the second epoch. The training RMSE and RMSPE curves indicate no obvious overfitting or underfitting issues - agreeing with the scatter plot displayed in Figure 4.1. To assess the validity of Gaussian assumption in the measurement equation (4.6), we plot the histogram of the initialization-averaged residuals extracted from the training set in Figure 4.4 and overlay the pdf of the fitted Gaussian distribution on these residuals.

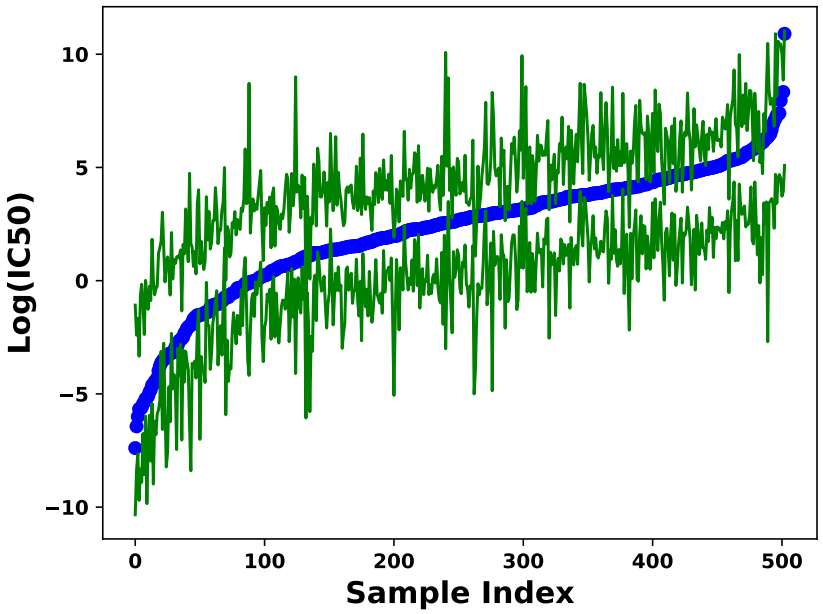

Figure 4.2: Prediction intervals for test samples with the observed log IC_{50} values.

It appears that the Gaussian assumption is tenable in our case.

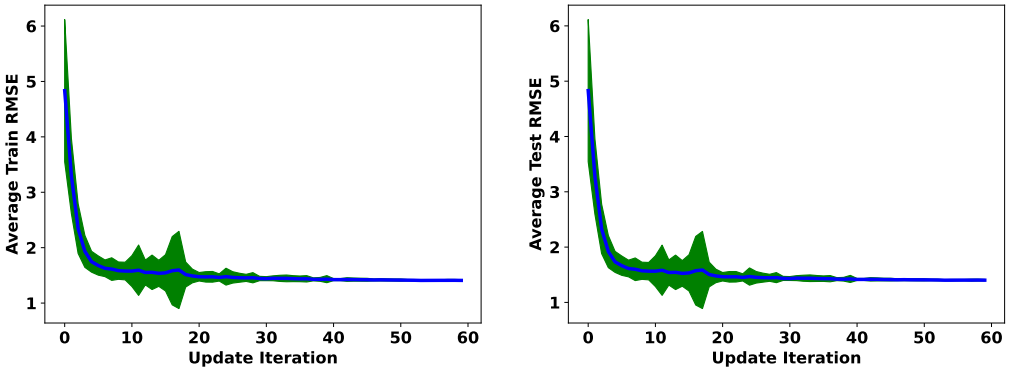

Figure 4.3: Training and testing RMSE curves for the MEnKF updating iterations

Finally, we report the estimated model weights $(s^{f_1^*}, s^{f_2*}, s^{g_1^*}, s^{g_2^*})$, averaged over the MEnKF-ANN runs, in Table 4.2. The stacker gives considerably more weightage to DualGCN, indicating that the architecture of DualGCN is more optimized for predicting CDR than DeepCDR's configuration. Drug embeddings extracted from DualGCN have the highest weight by a large margin. The omics embeddings extracted

from DualGCN and DeepCDR have essentially the same weightage - indicating both models are equally adept at processing the omics data. Closer inspection reveals that DualGCN drug embeddings dominate the omics embedding, but the reverse seems to be the case in DeepCDR, where omics embeddings dominate the drug embedding. To assess whether this curious reversal in weight pattern holds up for the base learners individually, we report the relative weightage of each type of embedding for each base learner approximated separately by MEnKF-ANN. We observe that the pattern of weights revealed in the stacked version agrees with those obtained from individual base learners. This is a key advantage of our stacking approach - it offers insight into the relative merits of different aspects of the base DLs without training every base learner separately.

Table 4.2: Model weights for drugs and omics features extracted from DeepCDR and DualGCN

Models	DeepCDR Drugs	DeepCDR Omics	DualGCN Drugs	DualGCN Omics
DeepCDR	0.41	0.59	NA	NA
DualGCN	NA	NA	.66	.34
DualGCN, DeepCDR	0.07	.15	.62	.16

4.5 Conclusion

Our goal here was to develop an extended version of the generalized stacking approach that can be used to generate predictions by stacking two DLs and attaching uncertainty to the predictions obtained from the stacked model. The proposed approach used a matrix Ensemble Kalman Filter-based neural network to perform model averaging and automatically generate prediction intervals. We utilized four neural networks, each ingesting drugs or omics features extracted from the base DLs. This helped us understand the relative predictive capability of different model/feature combinations. We also demonstrated that the coverage of the prediction interval, although exceeding

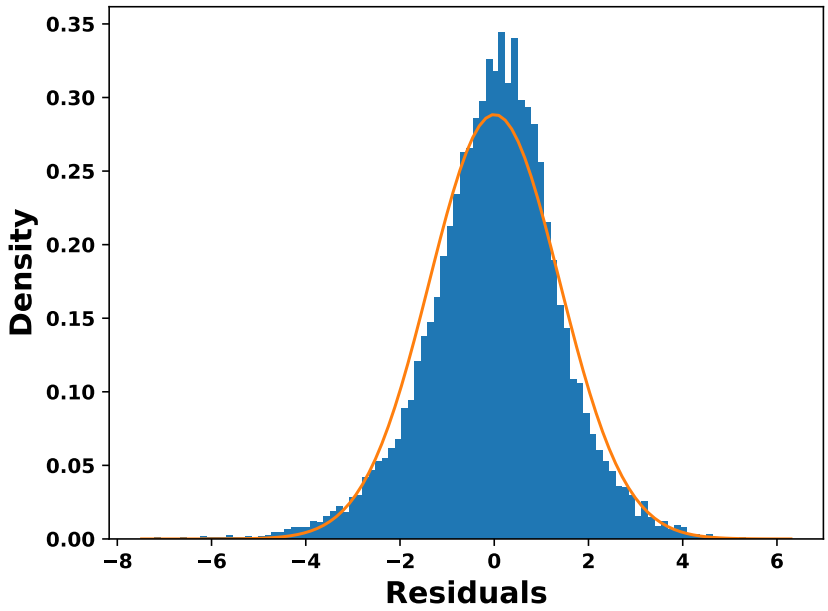

Figure 4.4: Normalized histogram of test set residuals with normal density curve

the nominal level, was reasonable, and the width of the prediction intervals was not too wide to render them meaningless.

The results obtained from stacking DeepCDR and DualGCN suggested that the predictive performance of the stacked model was better and more robust than the individual MEnKF-ANN approximated individual base learners. Although the prediction performance of the DLs (originally reported in [46, 44]) was numerically superior to our stacked model (potentially due to the more complex architecture), we posit that our approach is more robust in the sense that it can synthesize multiple models operating on different data types. For example, consider a test sample with genomic, transcriptomic, epigenomic, and CNV information. If we chose to use either DeepCDR or DualGCN, we would not be able to process at least one type of omics feature. The stacked model would offer the capability to handle all these features by synthesizing both these DLs. Furthermore, since we train the stacker sequentially, we can use this approach for online training when small batches of data are added

sequentially. This would alleviate the need to retrain the deep learners every time new data points are added - thereby saving computational resources when dealing with dynamic databases.

On the limitation side, the current architecture of our stacking neural network is simple. However, extending to more complex architecture is conceptually straightforward (see [77]). Additionally, the assumption of Gaussian errors in our state-space formulation may be too restrictive for skewed or heavy-tailed data. Regardless, our explicit error characterization allows us to perform residual analysis and thus allows us to assess specific aspects of model adequacy. The diagonal construction of the covariance matrix S_d is also simplistic. For a better representation, we can specify a more dense block diagonal matrix that induces covariance among the weights coming from the arms f_1^*, f_2^* and g_1^*, g_2^*.

The versatile nature of our stacking approach opens up some immediate future research directions. We can repurpose this approach for transfer learning. For example, if we wish to utilize the constituent deep learners on a small dataset, perhaps the most popular approach is to train the deep learners on the focal dataset with a *warm start*. With the stacker, we can pass the feature set associated with the focal dataset through the deep learners and extract the embeddings, which are then used to train the stacker. Another potential utility of our approach is that it allows us to infer changes in the data-generating model. For instance, suppose the first half of the data is generated by a particular model (DeepCDR, say), and the last half is generated by another model (DualGCN, say). Suppose both the data-generating models are a part of the ensemble. In that case, the trajectory of the convex model weights across the batches can tell us whether there is a switch in the data-generating process. Observe that traditional static model averaging protocols cannot be used to ascertain such switches in the data-generating regime.

Chapter 5

Extending Matrix Ensemble Kalman Filter-based stacker for predicting multivariate responses

5.1 Introduction

So far, the Matrix Ensemble Kalman Filter method has been developed for regression models with only one dependent variable (single output). This chapter extends the multi-arm MEnKF method for developing regression models with multiple dependent variables (multi-output). This method can also learn the covariance matrix for the multi-output dependent variables. We start by developing the theoretical framework and extend the MEnKF method developed for multivariate response. We then show how this method can be extended to accommodate and identify changes in the data-generating process.

We deploy the proposed multivariate MEnKF-ANN to predict the chemical properties of drug molecules from their Simplified Molecular Input Line Entry System (SMILEs) representations. We use the ChEMBL [25] database containing chemical properties for millions of potential drug compounds and their SMILEs representations. Researchers use this database frequently to discover new compounds that can act as effective drugs in treating various health conditions and illnesses. In particular, we focus on predicting the n-octanol-water partition coefficient (P) and the Polar

Surface Area (PSA). These two chemical properties of the compounds are often used to determine the therapeutic efficacy of drug compounds.

The P coefficient measures the relative solubility in fat (lipophilicity) and water (hydrophilicity). P is less than one if the drug compound is more soluble in water and greater than one if it is more soluble in fat. Consequently, any drug compounds with a higher value for P can accumulate in the fatty tissues of organisms (bio-accumulating) and can be detrimental. The Stockholm Convention, an international environmental treaty, deems any drug compounds with a value of P greater than five as having a serious risk of bio-accumulating and discourages their use. Lipinski's "Rule of 5" provides instructions to find favorable drug compounds that can be taken orally. One of the rules is to favor drug compounds with a value of P less than 5, similar to what the Stockholm Convention recommends. PSA is the area of the drug compounds' polar atoms (Oxygen, Nitrogen, and attached Hydrogen atoms). It measures the ability of the drug to permeate the cell membranes and penetrate the blood-brain barrier. Drug compounds having a PSA greater than 140 angstroms squared are considered to be poor at penetrating cell membranes. Compounds having a PSA of less than 90 angstroms squared can penetrate the blood-brain barrier and act on the central nervous system.

Calculating P and PSA for drug compounds experimentally is highly resource-intensive. Predictive models (for example, QSAR) are often used to predict P and PSA from experimentally collected data [20]. Our method for predicting the PSA and LogP (Logarithm of the n-octanol-water partition coefficient) is motivated by the QSAR method, which uses the molecular descriptors of the drug compounds as features for developing the prediction models. In addition to the molecular properties, we also use the SMILE representation of the chemical structure of the drug compounds. Therefore, the SMILE representation and the molecular descriptors function as the two sets of

features available for all drug compounds. The target variables we would be predicting are the LogP and the PSA for the drug compounds, making it a multivariate regression problem.

First, we design a multi-arm DL that simultaneously takes the SMILEs representations of the drug compounds and the molecular descriptors from the RDKit library that generates numerical features from the SMILEs. Since SMILEs are string representations of the chemical structure of the drug compounds, our multi-arm DL processes the SMILEs using an LSTM layer. The quantified features produced by RdKit are processed using fully connected dense layers. The embeddings obtained from the SMILEs subnetwork and RDKit subnetwork are concatenated and passed to the prediction layer, which comprises two neurons corresponding to the dependent variables of PSA and LogP.

We use the multi-arm DL as a feature extractor to extract the embeddings of SMILEs and RdKit features and train the multivariate MEnKF-ANN using these features. Our goal here is to assess the predictive capability of multivariate MEnKF-ANN when applied to the ChemBL database. We then offer two simulation studies to assess how accurately our stacking approach estimated the model weights. In the first simulation, we generate data by assigning fixed model weights to the SMILEs and the RDKit arm and then assess MEnKF's ability to recover those model weights. In the second simulation, we assume two sets of model weights, i.e., the training data comes from two data-generating processes that differ only with respect to the weights assigned to the SMILEs network vis-a-vis the RdKit network. We train the multivariate MEnKF-ANN on the resulting dataset and track the trajectory of the model-averaging weights to assess whether our method can detect this change in the data-generating process and dynamically adapt the model-averaging weights.

The remainder of this chapter is organized as follows: Section 5.2 describes the

structure of the various base models we use as feature extractors for input to the MEnKF method. Section 5.3 describes the construction of multivariate MEnKF-ANN. In section 5.4, we apply the multivariate stacker to predict the PSA and LogP values on a ChemBL dataset. Section 5.5 presents the simulation results that assess the performance of MEnKF in recovering the true model weights and its ability to identify switches in the model weights. We offer the concluding remarks in Section 5.6.

5.2 Base Models

First, we describe the base learners that take each feature type as input, generate the features' embeddings, and predict the target variables. The lower dimensional representations for SMILEs and the molecular descriptors, learned by the base models, are supplied to the MEnKF method. We explore two base model architectures, one with multi-arm architecture and one with single-arm architecture. The multi-arm DL ingests SMILEs and RDKit features in two subnetworks within a single consolidated architecture. Single-arm models predict the PSA and LogP using the SMILEs and the RDKit features separately.

In the multi-arm DL, we process the SMILE strings using an embedding and an LSTM layer and the molecular descriptors using a combination of fully connected dense layers. The input SMILE strings are first converted to their one-hot encoded representation. The dimensions of the one-hot encoded vectors are the same as the number of unique tokens in the SMILE strings population (also known as the vocabulary). The embedding layer learns a d_1 dimension vector representation (embeddings) for each unique token in the vocabulary. These learned embeddings are then used to create the input feature representation of the SMILE strings. A collection of these learned embeddings then represents each SMILE. More specifically, if a SMILE has l

tokens, it is represented by a $l \times d_1$ dimension array of features. This $l \times d_1$ dimension array of features is then passed to an LSTM layer, which learns a d_2 dimension vector representation for the SMILE. d_1 and d_2 are hyperparameters that can be chosen during the model-building stage. The LSTM embedding of the SMILEs is further processed by a block of two fully connected dense layers, creating a latent representation of the LSTM embedding. This latent representation is then concatenated with the embeddings produced by the fully connected layers for the RDKit representation and then passed to a prediction layer with two neurons for the two dependent variables of PSA and LogP. The resulting multi-arm architecture is trained by minimizing the mean squared error loss between the ground truth PSA and LogP values and the model predictions. After the model is trained, SMILEs and molecular descriptor embeddings can be extracted by simply passing the new training samples through the trained model.

In the second base model, we train two single-arm deep learners. One of the base model architectures uses the SMILEs strings to predict the logP and PSA, while the other uses the RDKit molecular descriptors to predict the same. The architecture of these two individual single-arm learners is similar to their corresponding subnetworks in the multi-arm DL. The final predictions are obtained via stacking the predictions generated by the pair of single-arm base learners.

The reasoning behind having two different base model architectures is motivated by the fact that we can use multivariate MEnKF-ANN to emulate the multi-arm DL to generate predictions. However, such multi-arm base learners do not explicitly assign individual weights to different types of representation. Therefore, we cannot assess our stacker's ability to recover the true model weights. Hence, in the simulation study, we use known weights to stack the estimates generated by two single-arm base learners and then deploy MEnKF-ANN to track the model weights.

5.3 Methods

We extend the method developed in Section 4.3 to make it suitable for multi-output regression. We define our target response as $Y \in \mathcal{R}^2$. We have a total of $m = \sum_{t=1}^{T} m_t$ training instances, with m_t being the number of training data points in the t^{th} batch. Let $v_t^{f_1} \in \mathcal{R}^{p_1}$ and $v_t^{f_2} \in \mathcal{R}^{q_1}$ denote two different learned embeddings extracted from the multi-arm DL f_{multi} for the t^{th} batch of data. Similarly, let $v_t^{g_1} \in \mathcal{R}^{p_2}$ and $v_t^{g_2} \in \mathcal{R}^{q_2}$ denote the learned embeddings extracted from the multi-arm DL g_{multi} for the t^{th} batch of data. Let f_1, f_2, g_1, g_2 denote the model architectures for DLs corresponding to the inputs $v_t^{f_1}$, $v_t^{f_2}$, $v_t^{g_1}$, $v_t^{g_2}$, respectively. All four model architectures have a single hidden layer and a final prediction layer with one neuron. Let $w_t^{f_1}$, $w_t^{f_2}$, $w_t^{g_1}$, and $w_t^{g_2}$ denote the updated weights corresponding to the model architectures of f_1, f_2, g_1, and g_2, respectively, using the t^{th} batch of data. Let $a_t^{f_1}$, $a_t^{f_2}$, $a_t^{g_1}$, $a_t^{g_2}$ and b_t be real-valued scalar parameters. We will assume that it is possible to extract embeddings from the multi-arm DLs, which are of the same dimension, implying that $p_1 = q_1 = p_2 = q_2$. We also assume that the model architectures of f_1, f_2, g_1, and g_2 have the same number of learnable parameters, n_{multi}. Define *softmax* as $\sigma(.) : \mathcal{R}^K \to [0,1]^K$ function which takes as input a vector z of K dimensions and applies the following formula to each element of z: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ for $i = 1, 2, \ldots, K$. Define a vector a_t as $a_t = [a_t^{f_1}, a_t^{f_2}, a_t^{g_1}, a_t^{g_2}]$. Let $s_t = [s_t^{f_1}, s_t^{f_2}, s_t^{g_1}, s_t^{g_2}]$ be a vector obtained after applying the *softmax* operation on the vector a_t. Let $s_t^{f_{multi}} = s_t^{f_1} + s_t^{f_2}$ and $s_t^{g_{multi}} = s_t^{g_1} + s_t^{g_2}$. $s_t^{f_{multi}}$ and $s_t^{g_{multi}}$ can be thought of as the weights given by the MEnKF-ANN method to the two multi-arm DL architectures of f_{multi} and g_{multi}.

5.3.1 Multi-Output Matrix Ensemble Kalman Filter utilizing features from two Multi-Arm DLs

Consider the state matrix, $X_t^{(m_t+n_{multi}+3)\times 4}$, associated with the t^{th} batch of data given by

$$X_t = \begin{bmatrix} s_t^{f_{multi}} f_1(v_t^{f_1}, w_t^{f_1}) & s_t^{f_{multi}} f_2(v_t^{f_2}, w_t^{f_2}) & s_t^{g_{multi}} g_1(v_t^{g_1}, w_t^{g_1}) & s_t^{g_{multi}} g_2(v_t^{g_2}, w_t^{g_2}) \\ w_t^{f_1} & w_t^{f_2} & w_t^{g_1} & w_t^{g_2} \\ a_t^{f_1} & a_t^{f_2} & a_t^{g_1} & a_t^{g_2} \\ c_t^{1,1} & c_t^{1,2} & c_t^{2,1} & c_t^{2,2} \\ d_t^{1,1} & 0 & 0 & d_t^{2,2} \end{bmatrix} \tag{5.1}$$

Define

$$H_t^{m_t\times(m_t+n_{multi}+3)} = [I_{m_t}, 0_{m_t\times(n_{multi}+3)}] \tag{5.2}$$

and

$$G_t^{4\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.3}$$

Additionally, define $\Theta_{t-1} = I_{m_t+n_{multi}+3}$ and $\psi_{t-1} = I_4$. Using the foregoing parameterizations of H_t, X_t, and G_t, we can define a Matrix State Space model similar to the one defined in Section 2.4 where the measurement equation and the state transition equation have the same form as (2.14) and (2.13). The measurement and state transition equations for the corresponding vector state space model also have

the same form as (2.19) and (2.17). It is easy to verify that such a construction does explicit model averaging by expanding the $H_t X_t G_t$ matrix multiplication in (2.14), which will result in a $m_t \times 2$ dimensional matrix.

$$\left[s_t^{f_{multi}} f_1(v_t^{f_1}, w_t^{f_1}) + s_t^{g_{multi}} g_1(v_t^{g_1}, w_t^{g_1}) \quad s_t^{f_{multi}} f_2(v_t^{f_2}, w_t^{f_2}) + s_t^{g_{multi}} g_2(v_t^{g_2}, w_t^{g_2}) \right] \tag{5.4}$$

Since $s_t^{f_{multi}} + s_t^{g_{multi}} = 1$, (5.4) is a weighted average of the predictions from the two pairs of models of f_1, g_1 and f_2, and g_2. Recall that the inputs to the four model architectures were the four sets of extracted embeddings $v_t^{f_1}$, $v_t^{g_1}$, $v_t^{f_2}$, and $v_t^{g_2}$ from the two multi-arm DLs of f_{multi} and g_{multi}. Therefore, by choosing X_t as in (5.1), we can train an ensemble model using the embeddings extracted from the trained multi-arm DLs. All the weights of the constituent four models in this ensemble and the averaging weights a_t are learned simultaneously.

The scalar parameters $c_t^{1,1}, c_t^{1,2}, c_t^{2,1}, c_t^{2,2}, d_t^{1,1}, d_t^{2,2}$ in (5.1) are used to estimate the covariance matrix of the error term ϵ_t in (2.19). Since $Y \in \mathcal{R}^2$, ϵ_t will have dimensions $2m_t \times 1$ and its covariance matrix will be a $2m_t \times 2m_t$ dimensional matrix. Consider matrices L_t and $D_t \succ 0$ parameterized as

$$L_t = \begin{bmatrix} c_t^{1,1} & c_t^{1,2} \\ c_t^{2,1} & c_t^{2,2} \end{bmatrix} \tag{5.5}$$

$$D_t = \begin{bmatrix} d_t^{1,1} & 0 \\ 0 & d_t^{2,2} \end{bmatrix} \tag{5.6}$$

$$R_t = L_t L_t^T + D_t \tag{5.7}$$

The above-chosen configurations of L_t, D_t would ensure that R_t is a symmetric positive semi-definite matrix of dimensions 2×2. To get the covariance matrix S_t of the $2m_t \times 1$ dimensional ϵ_t in (2.19) we can compute $S_t = R_t \otimes I_{m_t}$.

5.3.2 Solution for the Multi-Output Matrix Ensemble Kalman Filter

Proceeding similarly to Section 4.3.2, we first define the vector of unknown quantities that need to be estimated

$$u = [w_t^{f_1},\ w_t^{f_2},\ w_t^{g_1},\ w_t^{g_2},\ a_t^{f_1},\ a_t^{f_2},\ a_t^{g_1},\ a_t^{g_2},\ c_t^{1,1},\ c_t^{1,2},\ c_t^{2,1},\ c_t^{2,2},\ d_t^{1,1},\ d_t^{2,2}]$$

The first step is to generate N samples from $\mathcal{N}_d(0, S_d)$, where $d = 4n_{multi} + 10$ is the dimensionality of the vector u. S_d is a covariance matrix that needs to be specified. Denote these N samples as $U_t^0 = \{u_{j,t}^0\}_{j=1}^N$, where $u_{j,t}$ is a vector of the form $u_{j,t} = [w_{t,j}^{f_1},\ w_{t,j}^{f_2},\ w_{t,j}^{g_1},\ w_{t,j}^{g_2},\ a_{t,j}^{f_1},\ a_{t,j}^{f_2},\ a_{t,j}^{g_1},\ a_{t,j}^{g_2},\ c_{t,j}^{1,1},\ c_{t,j}^{1,2},\ c_{t,j}^{2,1},\ c_{t,j}^{2,2},\ d_{t,j}^{1,1},\ d_{t,j}^{2,2}]$. The first $4n_{multi}$ elements of $u_{j,t}$ are parameters for the four constituent ANN models, the next four elements are the model averaging weights, the next four elements are used to define the L_t, and the last two elements are used to define the D_t matrix. S_d can, therefore, be parameterized using four different variances σ^2_{ANNs}, $\sigma^2_{AvgWeights}$, σ^2_L and σ^2_D corresponding to the variances for the parameters of the ANN model weights, model averaging weights, and parameters that define the L_t and R_t matrices required to compute the covariance matrix of the batch of targets S_t.

$$S_d = \begin{bmatrix} \sigma^2_{ANNs} I_{4n_{multi}} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \sigma^2_{AvgWeights} I_4 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \sigma^2_L I_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \sigma^2_D I_2 \end{bmatrix} \tag{5.8}$$

Such a parameterization for S_d allows us to separately initialize the elements of the N samples corresponding to the constituent ANN weights, model averaging weights, and the parameters corresponding to the covariance matrix of the targets, respectively. These N samples are then updated using a modified version of the equation (4.11). The modified update equation using (4.11) can then be defined as

$$u_{j,t}^{n+1} = u_{j,t}^{n} + C(U^n)(D(U^n) + R_t)^{-1}(y_t + z_{j,t}^{n} - \mathcal{H}_t x_{t,j}^{n}) + z_{j,t}^{fudge} \tag{5.9}$$

where $\mathcal{H}_t = G_t^T \otimes H_t$, $C(U) = \frac{1}{N}\sum_{j=1}^{N}(u_{j,t} - \bar{u}) \otimes (\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}})^T$, $\bar{u} = \frac{1}{N}\sum_{j=1}^{N} u_{j,t}$, $\bar{\mathcal{H}} = \frac{1}{N}\sum_{j=1}^{N} \mathcal{H}_t x_{t,j}$, $x_{t,j}^{n} = \text{vec}\,(X_{t,j}^{n})$,$D(U) = \frac{1}{N}\sum_{j=1}^{N}(\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}}) \otimes (\mathcal{H}_t x_{t,j} - \bar{\mathcal{H}})^T$. $X_{t,j}^{n}$ is simply the X_t in (5.1) constructed using $u_{j,t}^{n}$, $z_{j,t}^{n} \sim \mathcal{N}(0, R_t)$, where R_t is defined as in the equation (5.7), using

$$L_t = \begin{bmatrix} \frac{\sum_{j=1}^{N} c_{t,j}^{1,1}}{N} & \frac{\sum_{j=1}^{N} c_{t,j}^{1,2}}{N} \\ \frac{\sum_{j=1}^{N} c_{t,j}^{2,1}}{N} & \frac{\sum_{j=1}^{N} c_{t,j}^{2,2}}{N} \end{bmatrix} \tag{5.10}$$

$$D_t = \begin{bmatrix} \frac{\sum_{j=1}^{N} d_{t,j}^{1,1}}{N} & 0 \\ 0 & \frac{\sum_{j=1}^{N} d_{t,j}^{2,2}}{N} \end{bmatrix} \tag{5.11}$$

and $z_{j,t}^{fudge} \sim \mathcal{N}(0, \sigma_{fudge}^2 I_d)$ where σ_{fudge}^2 is a user-defined variance parameter.

5.3.3 Connecting Multi-Output Matrix Ensemble Kalman Filters with Base Model Architectures

Recall that our goal is to predict the PSA and LogP for drug compounds using their SMILE representations and molecular descriptors. In Section 5.2, we presented two base model DL architectures that we would use to predict PSA and LogP.

We will first show how to parameterize (5.1) to process the embeddings extracted

from the multi-arm DL. Consider equation (5.4) which gives the matrix product $H_t X_t G_t$. To configure the X_t matrix in the MEnKF-ANN setup, we need to figure out how the terms in equation (5.4) relate to the multi-arm base model architecture. First, we extract the SMILE string embeddings and the molecular descriptor embeddings emanating from the last fully connected dense layers in the multi-arm DL. Let $s^{f_{multi}}$ and $s^{g_{multi}}$ be the weights assigned to SMILE string embeddings and molecular descriptor embeddings, respectively. Let v^{f_1} and v^{f_2} be the SMILE string embeddings and v^{g_1} and v^{g_2} be the molecular descriptor embeddings. f_1, f_2, g_1, and g_2 are all fully connected feed-forward neural networks with one hidden layer and one prediction layer with a single neuron. The matrix product in (5.4) has dimensions $m_t \times 2$ for a batch of size m_t. Comparing this with (2.14), we observe that (5.4) has the same dimensions as the batch targets, Y_t. Assume that the first column of Y_t contains the observed LogP values and the second column contains observed PSA values. Consider the first column in the matrix product of (5.4) which is $s_t^{f_{multi}} f_1(v_t^{f_1}, w_t^{f_1}) + s_t^{g_{multi}} g_1(v_t^{g_1}, w_t^{g_1})$. $s^{f_{multi}}$ and $s^{g_{multi}}$ are the weights assigned to SMILE string embeddings and molecular descriptor embeddings and $v_t^{f_1}$, $v_t^{g_1}$ are the SMILE string embeddings and the molecular descriptor embeddings, respectively. Since $s^{f_{multi}} + s^{g_{multi}} = 1$, $s_t^{f_{multi}} f_1(v_t^{f_1}, w_t^{f_1}) + s_t^{g_{multi}} g_1(v_t^{g_1}, w_t^{g_1})$ can be interpreted as the learned weighted average prediction for LogP using two simultaneously trained ANNs which use SMILE string embeddings and the molecular descriptor embeddings, respectively. Similarly, we can interpret $s_t^{f_{multi}} f_2(v_t^{f_2}, w_t^{f_2}) + s_t^{g_{multi}} g_2(v_t^{g_2}, w_t^{g_2})$ as the learned weighted average prediction for PSA using two simultaneously trained ANNs which use SMILE string embeddings and the molecular descriptor embeddings, respectively. Therefore, we can interpret (5.4) as a learned weighted average prediction for the LogP and PSA using the learned SMILE and molecular descriptor embeddings from the multi-arm base model architecture.

We have two single-arm DL architectures for the second base model that predict

LogP and PSA using SMILE strings and molecular descriptor features separately. The parameterization of (5.4) for the second base model architecture is similar to its multi-arm DL base model architecture parameterization. We would still have $s^{f_{multi}}$ and $s^{g_{multi}}$ as the weights assigned to SMILE string embeddings and molecular descriptor embeddings, respectively, but now the model averaging done in $H_t X_t G_t$ approximates the stacking of the two single-armed base learner. v^{f_1} and v^{f_2} be the SMILE string embeddings extracted from the single-arm DL that uses SMILE strings and v^{g_1} and v^{g_2} be the molecular descriptor embeddings extracted from the single-arm DL that uses molecular descriptors. f_1, f_2, g_1, and g_2 are all fully connected feed-forward neural networks with one hidden layer having 16 neurons and one prediction layer with a single neuron. The SMILE string embeddings and the molecular descriptor embeddings from both the multi-arm DL architecture and the two single-arm DL architecture are vectors of size 32.

To start the MEnKF method, we first calculate the number of parameters that need to be estimated, which is $d = 4n_{multi} + 10$. Based on the model architectures of f_1, f_2, g_1, and g_2 they all have the same number of learnable parameters $n_{multi} = 545$, therefore $d = 2190$. The number of samples for the EnKF solution in (5.9) is $N = 274$. The first step in the EnKF solution is to draw 274 samples from $\mathcal{N}_{2190}(0, S_{2190})$. S_{2190} is a covariance matrix of the form (5.8) and is defined using σ^2_{ANNs}, $\sigma^2_{AvgWeights}$, σ^2_L , and σ^2_D. For $t = 0$, denote these 274 samples as $U^0_t = \{u^0_{j,t}\}^{274}_{j=1}$, where $u_{j,t}$ is as described in Section 5.3.2. Compute X_t, H_t, and G_t, and $\mathcal{H}_t$ as described in Sections 5.3.1 and 5.3.2. These 274 samples are then updated using the equation (5.9) for $t = 1, 2, \ldots$.

5.4 Applications

We first describe the data used to train the base DLs and the multivariate MEnKF-ANN and offer results that demonstrate the predictive accuracy of our method along with the coverage and width of the prediction intervals that it generates.

5.4.1 Data Description

We use the ChEMBL database to extract drug molecules' chemical structure and properties. First, we extract 2.26 million SMILE strings and their observed LogP and PSA values from this database. These 2.26 million SMILE strings were split into 75% (1.70 million) training and 25% validation samples (0.56 million). This forms the dataset ($\mathcal{D}_b$) with which we train our base DLs - multi-arm and two single-arm DLs. Next, we also extracted 959 SMILE strings, corresponding to small molecules, which were not included in the foregoing 2.26 million SMILE strings used to train and validate the base DL architectures. These 959 SMILE strings and their logP and PSA values form the dataset ($\mathcal{D}_M$) that we used to train and evaluate the MEnKF-ANN method. The base DLs do not *see* small molecules. We would like to predict the LogP and PSA associated with small molecules. Therefore, our MEnKF-ANN acts as a transfer learner.

We use the *RDKit* library to numerically quantify the molecular features corresponding to the SMILE strings. This library computes 207 molecular features for each SMILE string. We use variance thresholding to remove features that show minimal variability and cannot explain variation in the response variables. Retaining molecular features with a variance of at least 1 leaves us with 104 (out of the 207) RDKit extracted molecular descriptors.

Training multivariate MEnKF-ANN: The 959 SMILE strings not used in the

training of the base models are first split into 75% training (719) and 25% testing (240) samples. The trained multi-arm base DL is used to forward propagate the SMILE strings and the molecular descriptors present in $\mathcal{D}_M$ to extract their respective embeddings. These embeddings are then used as input features to train MEnKF-ANN. We fix the dimension of the extracted embeddings (for both SMILE strings and the molecular descriptors) to be 32. These SMILE and molecular descriptor embeddings are then used as input features to train the MEnKF-ANN method. We used a batch size $m_t = 719$, $\sigma^2_{ANNs} = 0.1$, $\sigma^2_{AvgWeights} = 0.1$, $\sigma^2_L = 0.1$, and $\sigma^2_D = 0.1$. We stop training the MEnKF-ANN if the training RMSE does not improve for 20 successive epochs at the same level of coverage.

5.4.2 Results

Figure 5.1 shows the scatterplot of the observed vs. predicted LogP and PSA from the trained MEnKF-ANN model for the test samples. The MEnKF-ANN model achieves a high level of accuracy in predicting the LogP and PSA values. The high prediction accuracy is also evident from the Pearson correlation scores in Table 5.1 between the observed vs. predicted LogP and PSA, which is 0.99 for both. The coverage of the prediction intervals from MEnKF-ANN for both LogP and PSA testing samples are 93.33% and 97.92%, respectively. The average widths of the prediction intervals are 0.13 and 3.15, respectively. Figure 5.2 shows the histogram of the MEnKF-ANN predictions (LogP on the left and PSA on the right) superimposed with the observed value (in green) and the empirical 95% prediction intervals (in red) for four randomly chosen test samples. It can be seen that the widths of the prediction intervals are quite tight for both LogP and PSA predictions. Figure 5.3 shows the average weight for SMILEs embeddings estimated by MEnKF-ANN across the training epochs. The empirical convergence of model weight is evident from this figure. Additionally, the

blue background tracks the standard error of the model weights. It appears that the standard error converges as well.

Since we have bivariate responses, we can also estimate the response covariance matrix. The original sample covariance matrix in $\mathcal{D}_M$ is $\begin{bmatrix} 7.17 & -59.79 \\ -59.79 & 3088.22 \end{bmatrix}$. Since the N particles in the multivariate MEnKF-ANN generate N fitted values in the training samples, we can obtain N copies of empirical covariance matrices associated with the target response variable. The average of these N empirical covariance matrices can be used to estimate the covariance between LogP and PSA. This covariance estimate turns out to be $\begin{bmatrix} 7.16 & -59.65 \\ -59.65 & 3085.51 \end{bmatrix}$ indicating MEnKF-ANN's ability to recover original sample covariance.

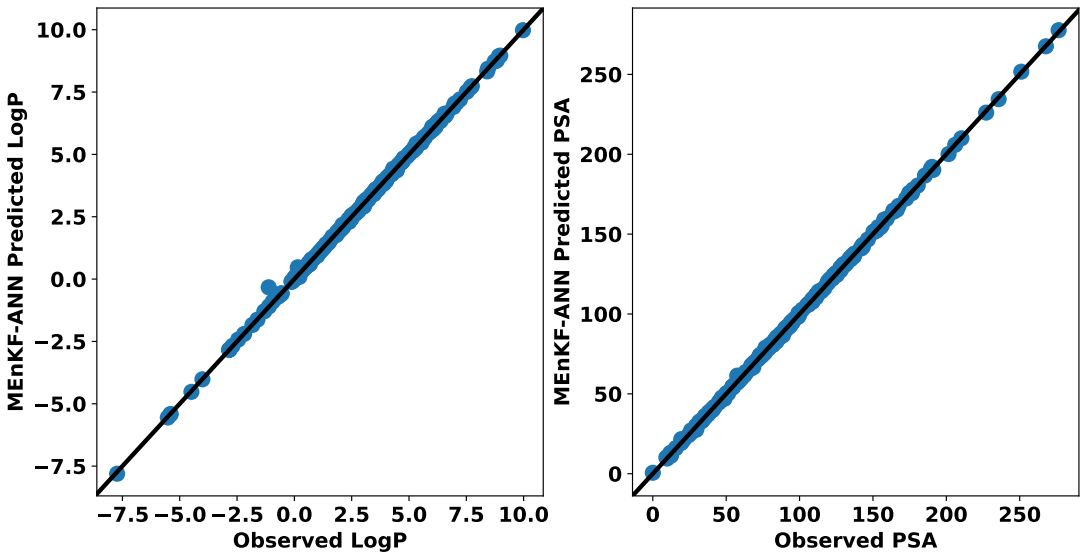

Figure 5.1: Scatterplot showing the average MEnKF LogP and PSA prediction with their corresponding ground truth values for the test set

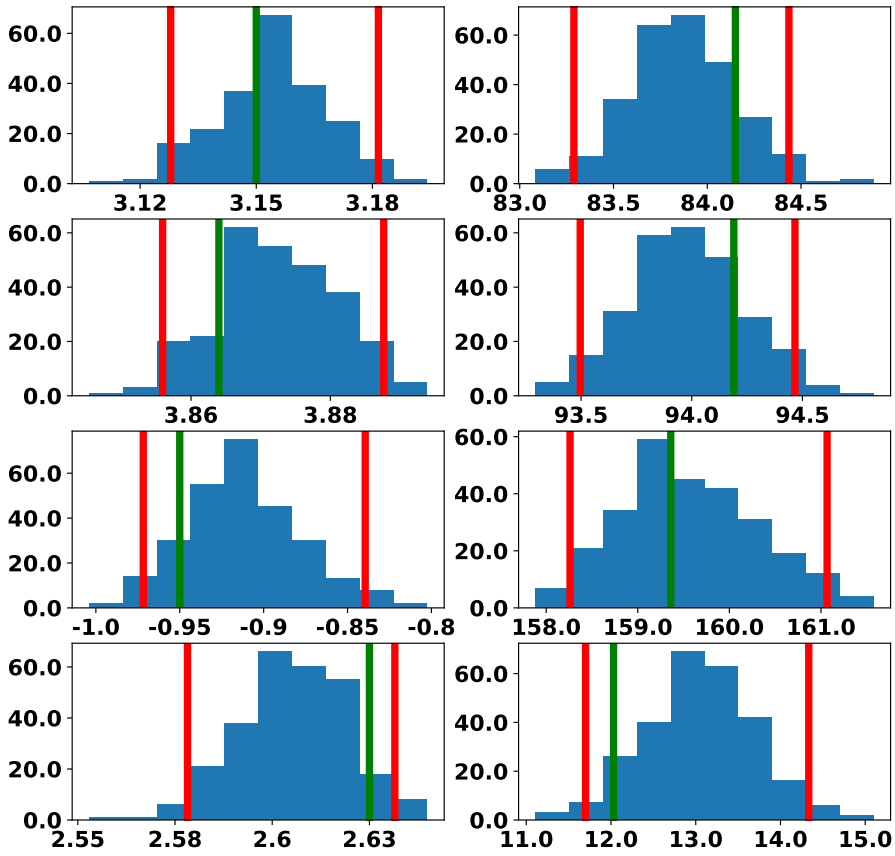

Figure 5.2: MEnKF-ANN predictions superimposed with the ground truth value and the empirical 95% prediction intervals

Table 5.1: Mean performance metrics of MEnKF-ANN for the prediction of LogP and PSA in the test set.

Target	RMSE	Coverage	Width	ρ
LogP	0.07	93.33%	0.13	0.99
PSA	0.74	97.92%	3.15	0.99

5.5 Simulations

In this section, we demonstrate the ability of MEnKF-ANN to identify a shift in the weights assigned to the base learners. In particular, we are interested in the scenario where a part of the dataset is generated by a particular weighted combination of the outputs from the SMILE embeddings and the outputs from the molecular descriptor

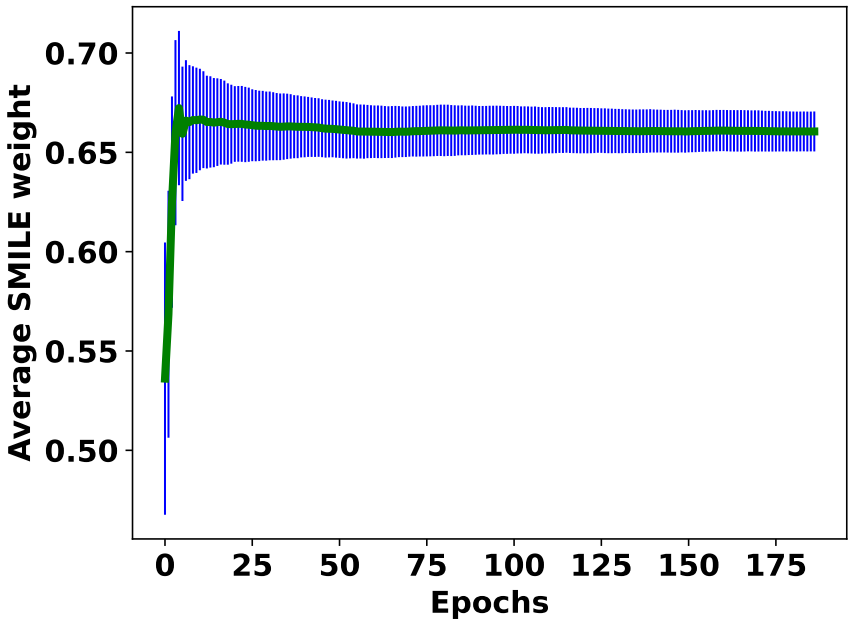

Figure 5.3: Trajectory of the average SMILE embedding weights from MEnKF-ANN over the epochs.

embeddings. However, the magnitude of the model weights changes in the subsequent batches.

Since the MEnKF-ANN estimates model averaging weights for each training batch, we hypothesize that by tracking the model weight across batches, we can identify if the true model weights change dynamically. We designed two synthetic scenarios to investigate this hypothesis. In the first scenario, the model weight is kept constant. In the second scenario, we make one switch in the model weight. In all simulations, we report the point estimate, the average width, and the coverage probability of the empirical 95% prediction intervals.

5.5.1 Fixed case scenario

Data generation scheme: Recall from Section 5.2 that the last layer in the multi-arm DL before the prediction layer was a concatenation layer which concatenates embeddings

for the SMILE strings (v_t^{SMILES}) and molecular descriptors ($v_t^{MolDescriptors}$), each a 32-dimensional vector. Consider a single training sample such that the batch size $m_t = 1$ and therefore v_t^{SMILES}, $v_t^{MolDescriptors}$ both are having dimensions 1×32. The predicted LogP and PSA for this training sample can be obtained using $(v_t^{SMILES} \times w_t^{SMILES}) + (v_t^{MolDescriptors} \times w_t^{MolDescriptors}) + (b_t)^T$, where b is the bias vector that produces the mean of the bivariate responses. Therefore, the predicted LogP and PSA can be seen as having some contribution from the SMILEs embeddings and the molecular descriptor embeddings. We modify the prediction equation by introducing two new parameters, $weight_t^{SMILEs}$ and $(1 - weight_t^{SMILEs})$, which are weights given to the contribution stemming from SMILE embeddings and molecular descriptor embeddings, respectively. Therefore, the mean values of the synthetic responses are obtained using the formula $weight_t^{SMILEs}(v_t^{SMILES} \times w_t^{SMILES}) + (1 - weight_t^{SMILEs})(v_t^{MolDescriptors} \times w_t^{MolDescriptors}) + (b_t)^T$. The final target response data are obtained by perturbing the above means by noises generated from $\mathcal{N}_2(0, R_t)$.

We assess the performance under three combinations of $weight^{SMILEs}$ and noise covariances. We refer to these combinations as C_1, C_2 and C_3, with $C_1 = (0.7, R_1), C_2 = (0.7, R_2)$, and $C_3 = (0.8, R_3)$, where

$$R_1 = \begin{bmatrix} 0.3 & 0.06 \\ 0.06 & 0.3 \end{bmatrix} \; R_2 = \begin{bmatrix} 0.3 & -0.27 \\ -0.27 & 0.3 \end{bmatrix} \; R_3 = \begin{bmatrix} 0.2 & -0.18 \\ -0.18 & 0.2 \end{bmatrix}$$

We generate 50 synthetic datasets for each C_1, C_2, and C_3, comprising 959 instances under each covariance specification. We use the SMILEs and molecular descriptors associated with small molecules that were not supplied to the multi-arm DL. In each simulated replicate, we use 719 samples for training MEnKF-ANN and the remaining 240 samples for testing.

Results: Fixed case scenario

In this simulation exercise, we aim to recover $weight^{SMILEs}$ under three different

combinations C_1, C_2, and C_3. Table 5.2 gives the coverage and width of the confidence intervals for $weight^{SMILEs}$ parameter from MEnKF-ANN. $\widehat{weight}^{SMILEs}$ is the average estimated model weight for SMILE embeddings from MEnKF-ANN. It can be seen that the $\widehat{weight}^{SMILEs}$ from MEnKF-ANN is close to the true $weight^{SMILEs}$. The coverage of the confidence intervals exceeds the nominal level, and the widths are also reasonable for all three combinations.

Table 5.3 gives the coverage and width of the prediction intervals for LogP and PSA yielded by MEnKF-ANN on the test data. It can be seen that the RMSEs, coverages, and widths are all reasonable. Therefore, the evidence in Table 5.2 and Table 5.3 indicate that the MEnKF-ANN method has good predictive performance and can accurately recover the true model weights.

Figure 5.4 shows the trajectory of $\widehat{weight}^{SMILEs}$ for combination C_2. Observe how the weight converges to a steady state within the first 20 epochs. We expect to see such stability in weight trajectory when the data generation process remains the same.

Table 5.2: Average estimated SMILE weight by MEnKF-ANN along with the coverage and widths from its empirical 95% confidence interval.

Combination	$weight^{SMILEs}$	R_t	$\widehat{weight}^{SMILEs}$	Coverage	Width
C_1	0.7	R_1	0.76	96%	0.20
C_2	0.7	R_2	0.75	96%	0.20
C_3	0.8	R_3	0.83	96%	0.15

Table 5.3: Mean performance metrics of MEnKF-ANN for predicting LogP and PSA in the test set along with coverage and widths from its empirical 95% prediction interval.

Combination	LogP RMSE	PSA RMSE	LogP Cov	PSA Cov	LogP Width	PSA Width
C_1	0.27	5.79	91.31%	93.60%	0.82	17.82
C_2	0.29	5.95	91.47%	93.68%	0.83	17.98
C_3	0.22	4.68	91.95%	94.13%	0.62	13.66

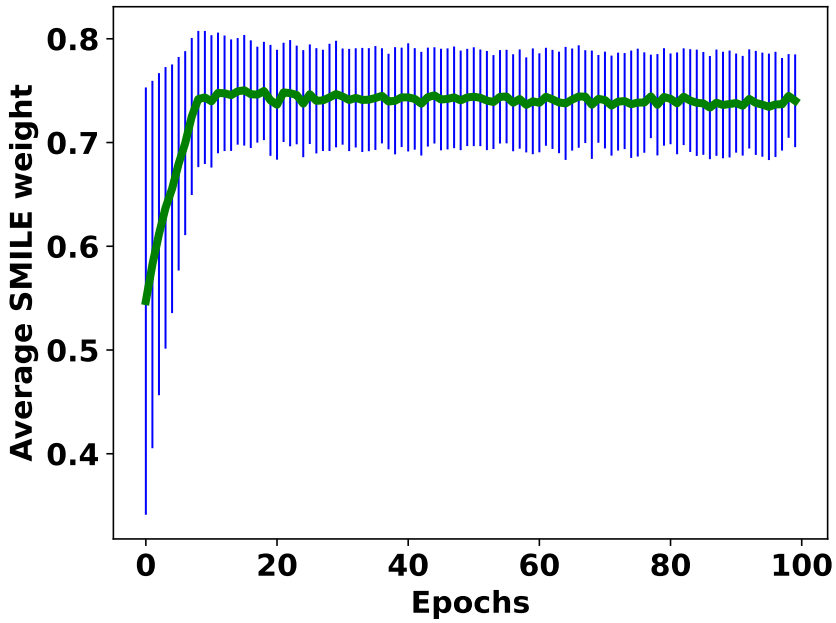

Figure 5.4: Trajectory of the average SMILE embedding weights from MEnKF-ANN for combination C_2 over the epochs.

5.5.2 Dynamic weight scenario

Data generation scheme: In this scenario, we extract the 32-dimensional embeddings associated with the SMILE strings (v_t^{SMILES}) and molecular descriptors ($v_t^{MolDescriptors}$) from the two single-arm DLs whose architectures were described in Section 5.2. Just like in Section 5.5.1, true target data in the simulations is obtained by adding m_t random noise vectors drawn from $\mathcal{N}(0, R_t)$ to the final prediction equation given by $weight_t^{SMILEs}(v_t^{SMILES} \times w_t^{SMILES}) + weight_t^{MolDescriptors}(v_t^{MolDescriptors} \times w_t^{MolDescriptors}) + (b_t)^T$. R_t, $weight_t^{SMILEs}$, and $weight_t^{MolDescriptors}$ are user specified. We use one R_t configuration in this simulation.

$$R_4 = \begin{bmatrix} 0.3 & -0.06 \\ -0.06 & 0.3 \end{bmatrix}$$

We create a weight switch scenario in the following way: To generate the first batch, we use 959 SMILE strings and associated RdKit descriptors and simulate mean

values of LogP and PSA using $weight_t^{SMILEs} = 0.7$. We generate another 959 instances for the second batch using the same feature set as before, but now $weight_t^{SMILEs}$ is set to be 0.4. We then perturb the 1918 data points with bivariate normal noises with mean zero and covariance R_4. Observe that the only systematic difference between the first 959 instances and the following 959 instances is the change in the model averaging weights assigned to the output generated by the DL trained on the SMILEs feature.

Results: dynamic weights

Instead of letting MEnKF-ANN scan the entire training data in each epoch, we partition 1918 instances into a few equally sized blocks. Each block is treated as an independent training set and trained over multiple epochs. When we transition to a new block, we restart the training but use the learned parameters from the previous block to initialize the state matrix for the new block. We do not specify any specific exit criteria but observe the trajectories of the model weights over the epochs until they reveal stability, as observed in the fixed weight case. We used the following MEnKF-ANN hyperparameter values for this simulation $N = 219$, $\sigma^2_{ANNs} = 1$, $\sigma^2_{AvgWeights} = 3$, $\sigma^2_L = 1$, and $\sigma^2_D = 1$.

Figure 5.5 shows the average estimated model weight trajectory for SMILE embeddings when the entire dataset is partitioned into two blocks. The blue curve shows the trajectory for the first block, and the orange curve shows the trajectory for the second block. Note the transition in the trajectory when MEnKF-ANN starts retraining with the second block. This indicates that blocking the training data and multiple restarts can enable MEnKF-ANN to identify a potential switch in the data-generating regime, at least in terms of weights allocated to the constituent base models.

Left panel of Figure 5.6 shows the ratio of the training error associated with

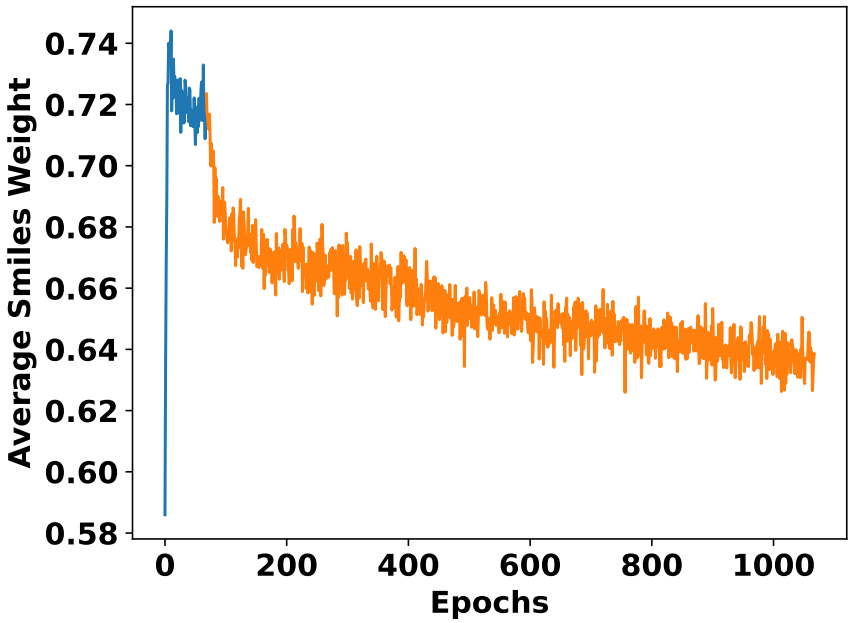

Figure 5.5: Trajectory of the average SMILE embedding weights from MEnKF-ANN over the epochs for dynamic weights scenario

LogP from the SMILEs-only arm of MEnKF-ANN to the molecular descriptors-only arm of MEnKF-ANN, and the right panel shows the same but now for PSA. Recall that the true model weight corresponding to molecular descriptor features was higher in the second block. Consequently, the ratio of the training errors becomes smaller over the epochs, signifying that the predictions from the molecular descriptor-only arms of the MEnKF-ANN are getting more accurate than the predictions from the SMILEs-only arm. This further supports the trajectory observed in Figure 5.6.

5.6 Conclusion

In this chapter, we have extended our MEnKF-ANN method to accommodate multivariate responses. We demonstrated that the multivariate extension is conceptually straightforward mainly due to the utilization of matrix normal theory in constructing our stacker. We have shown how this stacker performs transfer learning, transmitting

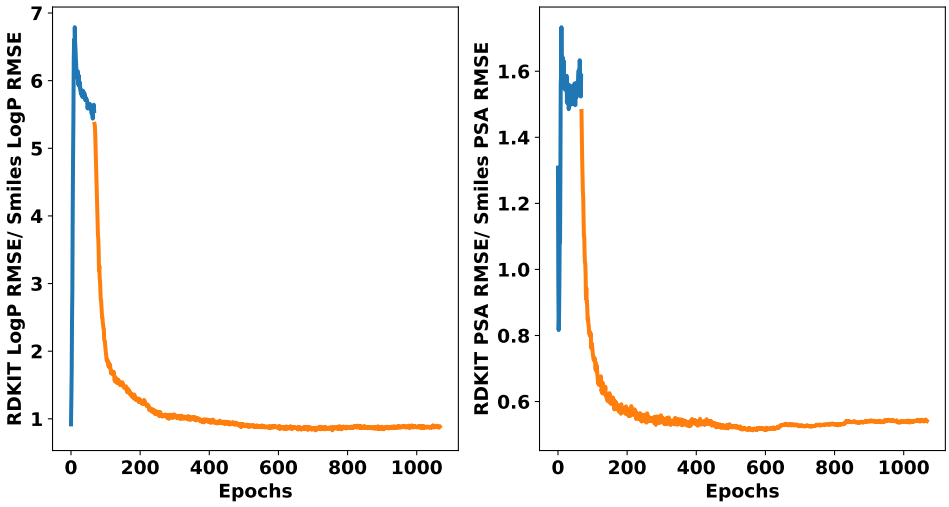

Figure 5.6: Trajectory of the ratio of RMSPE from molecular descriptor embeddings only model to the RMSPE from SMILE string embeddings only model for dynamic weights scenario

knowledge from base deep learners trained on a different dataset. We have also shown how multiple restarts in the MEnKF-ANN algorithm enable us to identify a switch in the model averaging weights. We note that this aspect of our stacker is important because, as shown in Chapter 3, the property of EnKF forces the trajectory of the forecast estimates to converge in the long run regardless of the initialization. With a switch in the model weight, we would expect the forecasts to have different distributions under different regimes; hence, vanilla EnKF cannot handle such a situation.

On the limitation side, this approach relies on sequential training of each data partition and hence cannot be parallelized. Additionally, in order to identify regime switch, it requires stability in the regime in most partitions. In other words, if the dataset rows are randomly permuted, MEnKF-ANN will fail to identify which data point comes from which regime. We emphasize that MEnKF-ANN, in its current version, is not a model that can detect change points. It is simply an exploratory tool that exploits the asymptotic properties of EnKF and KF to assess the stability of

the steady-state solution of the state variable. The guiding hypothesis is that if the data-generating process remains stationary, then the steady-state solution of the state variable in EnKF will not depend upon the initialization. Therefore, if MEnKF-ANN is trained sequentially with multiple restarts, the state variables will converge to the same steady-state distribution. If we observe that the steady state distribution of the state variables depends on the initialization, then it simply indicates non-stationarity in the data-generating process. Our simulation study simply offers a proof-of-concept that if the non-stationarity lies in the specification of mean in the measurement model, then tracking the trajectories of the *cross-learner* weights under multiple restarts can potentially reveal the dependence of the steady solution of that state variable on the initialization.

Future research will be devoted to alleviating the foregoing limitations. For computational benefit, we propose to train each data partition using multiple overdispersed initializations parallelly. We can then track the model weight trajectory for each initialization. We will thus end up with a collection of trajectories for each partition. A functional ANOVA on these trajectories could be used to determine whether the population mean of these trajectories remains the same across the data partition.

Turning to regime switch integration, we propose using the regime-switching state-space model [15] instead of vanilla EnKF. Such integration will offer further generalization to the stacking approach.

Chapter 6

Conclusion

In this dissertation, we have proposed a generalized approach for model averaging multiple ANNs. Although there is a rich repository of literature discoursing EnKF theory and model averaging procedures, this dissertation, to the best of our knowledge, is the first one to demonstrate that the EnKF framework could be naturally extended to perform model averaging of multiple ANNs. We used feed-forward ANN architecture for the base learners and trained individual ANNs and their convex combinations simultaneously using Ensemble Kalman Filter updating equations. The utility of our EnKF-trained ANN is that it captures the epistemic uncertainty in the model parameters in a more coherent way and under certain conditions generates the minimum mean square error estimates of all the model parameters that enter into the state matrix. Additionally, EnKF explicitly characterizes the joint distributional profile of the state variable and therefore the predictive distribution is explicitly written in the form of the conditional distribution of the query points given the training dataset and the input features associated with the query points. Hence, the conditional expectation of this predictive density is the minimum mean square error estimator of the point prediction and the conditional quantiles form genuine predictive intervals. We empirically demonstrate that the coverage probability associated with these predictive intervals achieves a nominal level. Additionally, when the response space is compact,

we demonstrate that the average width of the prediction interval is small enough to make these intervals useful. On the passing, we note that the uncertainty intervals generated by the customary Monte Carlo dropout procedure falls far short of the nominal level.

The connection between our approach and deep learners is revealed when we allocate different columns of the state matrix to the feature embeddings coming from different base deep learners (or from different arms of a multi-arm deep learner). MEnKF-ANN now essentially approximates the dense prediction layers in the constituent deep learners by feed-forward ANNs and estimates the *within-learner* and *cross-learner* weights simultaneously. Unlike the conventional stacking approach, MEnKF-ANN does not use the predicted values generated by the DLs as covariates for the meta-learner, thereby ignoring the uncertainty associated with the output generated by each of these DLs. We posit that the assumption of non-stochasticity in the input of meta-learners is not tenable, particularly when we have deep learners as base models because almost all deep learners are trained with dropout layers which structurally induce stochasticity in the predictions. Our method utilizes the embeddings extracted from the deep learners and does not make any assumption regarding the stochasticity of the output generated by the base models.

We showed that MEnKF-ANN construction is useful in performing transfer learning, particularly when the target domain requires a reduced model to be trained as compared to more complex models that were trained in the source domain. Observe that conventional stacking would first require retraining of the source domain models in the target domain and then estimate the stacking weights on hold-out validation data in the target domain. MEnKF-ANN does not require retraining of source domain models. It simply extracts the required embeddings for the features in the target domain and trains base ANNs and their convex combination simultaneously on the

target domain data. We also develop a multivariate extension of MEnKF-ANN and demonstrate that this extension simply requires expansion in the dimension of the state matrix. All distribution specifications and updating equations retain the same form with appropriate changes in the matrix dimensions. Finally, we show that instead of letting MEnKF-ANN scan the entire training set in a single go, if we perform multiple restarts across different partitions of the training data, the convergence properties of EnKF could be exploited to identify if there is an abrupt change in the mean function of the observation model. We offered a proof-of-concept simulation exercise to assess whether MEnKF-ANN could detect if the model-averaging weights change from one training set to another.

There are several limitations of the proposed approach. First, MEnKF-ANN produces optimal (in minimum mean square error sense) estimates of the *within-learner* and *cross-learner* weights under the assumptions specified in Chapter 2. If any of those assumptions are violated, optimality of the estimates is not guaranteed, MEnKF-ANN simply boils down to an algorithm that can perform *in-situ* model averaging. Second, in its current formulation, MEnKF-ANN cannot perform model averaging on deep learners directly. That is, the current computation protocol cannot have $f(.)$ and $g(.)$ to be deep learners themselves. More restrictively, it cannot handle non-numeric input features. That is, we need to supply numeric values for v_t^f and v_t^g. Therefore, to handle object-type input features (say, images), MEnKF-ANN would require an autoencoder-type technique to convert the object inputs to vector-valued features. Third, as the name of our approach suggests, we only consider feed-forward ANNs as the base learners. In its current state, MEnKF-ANN can handle different ANN architectures, but it cannot handle two very different base learners - for instance, an ANN and a regression tree. Observe that conventional stacking does not suffer from this problem because the stacking formulation (as shown in (1.1)) is agnostic to the architecture of

the base models that generate the predictions. Fourth, the uncertainty statement that MEnKF-ANN attached to the predicted values of the target variables only captures the uncertainty arising due to our lack of knowledge about the model parameters. It assumes that the true data-generating model appears in the set of models that are being averaged. Consequently, uncertainty due to our lack of knowledge about the model itself is not properly quantified in the distribution profile of the model parameters and predictive distributions. Finally, some of the illustrative examples should be interpreted with caution- particularly when it comes to transfer learning and detection of model switch. Our intention in deploying MEnKF-ANN in those situations was to demonstrate that this framework is flexible enough to perform several functions. But, there exist multiple techniques that are explicitly developed to handle those tasks [19, 49]. Consequently, we cannot claim that MEnKF-ANN generates optimal predictions under these situations. It only offers a potential candidate model.

In terms of future research directions, introducing particle filters in this context is an obvious way to relax the Gaussianity assumption hard-wired in MEnKF-ANN. Additionally, formally introducing switching Kalman Filters to draw inferences on the switching regime model will add more rigor to our current approach. Theoretically, we can generalize this approach to multilayer ANN instead of the current single-layer feed-forward ANN. With this generalization, we can prove that we can consistently approximate any deep learner because [27, 8] proved that sufficiently complex multilayer feed-forward networks can accurately approximate arbitrary mappings from input space to the response space.

Bibliography

[1] M. Abdar, F. Pourpanah, S. Hussain, D. Rezazadegan, L. Liu, M. Ghavamzadeh, P. Fieguth, X. Cao, A. Khosravi, U. R. Acharya, et al. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion*, 76:243–297, 2021.

[2] J. L. Anderson. An ensemble adjustment kalman filter for data assimilation. *Monthly weather review*, 129(12):2884–2903, 2001.

[3] C. Ausland, J. Zheng, H. Yi, B. Yang, T. Li, X. Feng, B. Zheng, and Y. Yin. dbcan-pul: a database of experimentally characterized cazyme gene clusters and their substrates. *Nucleic Acids Research*, 49(D1):D523–D528, 2021.

[4] P. Badjatiya, S. Gupta, M. Gupta, and V. Varma. Deep learning for hate speech detection in tweets. In *Proceedings of the 26th international conference on World Wide Web companion*, pages 759–760, 2017.

[5] P. J. Ballester, R. Stevens, B. Haibe-Kains, R. S. Huang, and T. Aittokallio. Artificial intelligence for drug response prediction in disease models, 2022.

[6] D. Baptista, P. G. Ferreira, and M. Rocha. Deep learning for drug response prediction in cancer. *Briefings in bioinformatics*, 22(1):360–379, 2021.

[7] J. Barretina, G. Caponigro, N. Stransky, K. Venkatesan, A. A. Margolin, S. Kim, C. J. Wilson, J. Lehár, G. V. Kryukov, D. Sonkin, et al. The cancer cell line

encyclopedia enables predictive modelling of anticancer drug sensitivity. *Nature*, 483(7391):603–607, 2012.

[8] A. R. Barron. Statistical properties of artificial neural networks. In *Proceedings of the 28th IEEE Conference on Decision and Control,*, pages 280–285. IEEE, 1989.

[9] M. K. Bjursell, E. C. Martens, and J. I. Gordon. Functional genomic and metabolic studies of the adaptations of a prominent adult human gut symbiont, bacteroides thetaiotaomicron, to the suckling period. *Journal of biological chemistry*, 281(47):36269–36279, 2006.

[10] Cancer Genome Atlas Research Network, J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart. The cancer genome atlas pan-cancer analysis project. *Nat Genet*, 45(10):1113–1120, 2013.

[11] C. Chen, X. Lin, Y. Huang, and G. Terejanu. Approximate bayesian neural network trained with ensemble kalman filter. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[12] C. Chen, X. Lin, and G. Terejanu. An approximate bayesian long short-term memory algorithm for outlier detection. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 201–206. IEEE, 2018.

[13] Y. Chen and L. Zhang. How much can deep learning improve prediction of the responses to drugs in cancer cell lines? *Briefings in bioinformatics*, 23(1):bbab378, 2022.

[14] D. Choukroun, H. Weiss, I. Y. Bar-Itzhack, and Y. Oshman. Kalman filtering for matrix estimation. *IEEE Transactions on Aerospace and Electronic Systems*, 42(1):147–159, 2006.

[15] J. Y. Chow. Nonlinear learning underpinning pedagogy: evidence, challenges, and implications. *Quest*, 65(4):469–484, 2013.

[16] P. Christensen, K. Gillingham, and W. Nordhaus. Uncertainty in forecasts of long-run economic growth. *Proceedings of the National Academy of Sciences*, 115(21):5409–5414, 2018.

[17] R. M. Cooke. Averaging quantiles, variance shrinkage, and overconfidence. *Futures & Foresight Science*, 5(1):e139, 2023.

[18] E. Y. Cramer, E. L. Ray, V. K. Lopez, J. Bracher, A. Brennen, A. J. Castro Rivadeneira, A. Gerding, T. Gneiting, K. H. House, Y. Huang, et al. Evaluation of individual and ensemble probabilistic forecasts of covid-19 mortality in the united states. *Proceedings of the National Academy of Sciences*, 119(15):e2113561119, 2022.

[19] H. Daumé III. Frustratingly easy domain adaptation. *arXiv preprint arXiv:0907.1815*, 2009.

[20] P. Ertl, B. Rohde, and P. Selzer. Fast calculation of molecular polar surface area as a sum of fragment-based contributions and its application to the prediction of drug transport properties. *Journal of medicinal chemistry*, 43(20):3714–3717, 2000.

[21] E. Fersini, E. Messina, and F. A. Pozzi. Sentiment analysis: Bayesian ensemble learning. *Decision support systems*, 68:26–38, 2014.

[22] H. J. Flint, K. P. Scott, S. H. Duncan, P. Louis, and E. Forano. Microbial degradation of complex carbohydrates in the gut. *Gut microbes*, 3(4):289–306, 2012.

[23] T. M. Fragoso, W. Bertoli, and F. Louzada. Bayesian model averaging: A systematic review and conceptual classification. *International Statistical Review*, 86(1):1–28, 2018.

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

[25] A. Gaulton, L. J. Bellis, A. P. Bento, J. Chambers, M. Davies, A. Hersey, Y. Light, S. McGlinchey, D. Michalovich, B. Al-Lazikani, et al. Chembl: a large-scale bioactivity database for drug discovery. *Nucleic acids research*, 40(D1):D1100–D1107, 2012.

[26] A. K. Gupta and D. K. Nagar. *Matrix variate distributions*. Chapman and Hall/CRC, 2018.

[27] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[28] R. Hu, Q. Huang, S. Chang, H. Wang, and J. He. The mbpep: a deep ensemble pruning algorithm providing high quality uncertainty prediction. *Applied intelligence*, 49:2942–2955, 2019.

[29] L. Huang, H. Zhang, P. Wu, S. Entwistle, X. Li, T. Yohe, H. Yi, Z. Yang, and Y. Yin. dbcan-seq: a database of carbohydrate-active enzyme (cazyme) sequence and annotation. *Nucleic Acids Research*, 46(D1):D516–D521, 2018.

[30] M. A. Iglesias, K. J. Law, and A. M. Stuart. Ensemble kalman methods for inverse problems. *Inverse Problems*, 29(4):045001, 2013.

[31] A. E. Kaoutari, F. Armougom, J. I. Gordon, D. Raoult, and B. Henrissat. The abundance and variety of carbohydrate-active enzymes in the human gut microbiota. *Nature Reviews Microbiology*, 11(7):497–504, 2013.

[32] T. Karaletsos and T. D. Bui. Hierarchical gaussian process priors for bayesian neural network weights. *Advances in Neural Information Processing Systems*, 33:17141–17152, 2020.

[33] M. Katzfuss, J. R. Stroud, and C. K. Wikle. Understanding the ensemble kalman filter. *The American Statistician*, 70(4):350–357, 2016.

[34] W. Khan, S. Walker, and W. Zeiler. Improved solar photovoltaic energy generation forecast using deep learning-based ensemble stacking approach. *Energy*, 240:122812, 2022.

[35] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[36] L. Kleeman. Understanding and applying kalman filtering. In *Proceedings of the Second Workshop on Perceptive Systems, Curtin University of Technology, Perth Western Australia (25-26 January 1996)*, 1996.

[37] N. M. Koropatkin, E. A. Cameron, and E. C. Martens. How glycan metabolism shapes the human gut microbiota. *Nature Reviews Microbiology*, 10(5):323–335, 2012.

[38] N. B. Kovachki and A. M. Stuart. Ensemble kalman inversion: a derivative-free technique for machine learning tasks. *Inverse Problems*, 35(9):095005, 2019.

[39] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei. Automatic differentiation variational inference. *Journal of machine learning research*, 2017.

[40] V. Kuleshov, N. Fenner, and S. Ermon. Accurate uncertainties for deep learning using calibrated regression. In *International conference on machine learning*, pages 2796–2804. PMLR, 2018.

[41] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[42] F. Le Gland, V. Monbet, and V.-D. Tran. *Large sample asymptotics for the ensemble Kalman filter*. PhD thesis, INRIA, 2009.

[43] J. Liu, J. Paisley, M.-A. Kioumourtzoglou, and B. Coull. Accurate uncertainty estimation and decomposition in ensemble learning. *Advances in neural information processing systems*, 32, 2019.

[44] Q. Liu, Z. Hu, R. Jiang, and M. Zhou. DeepCDR: a hybrid graph convolutional network for predicting cancer drug response. *Bioinformatics*, 36(Supplement_-2):i911–i918, 2020.

[45] V. Lombard, H. Golaconda Ramulu, E. Drula, P. M. Coutinho, and B. Henrissat. The carbohydrate-active enzymes database (cazy) in 2013. *Nucleic acids research*, 42(D1):D490–D495, 2014.

[46] T. Ma, Q. Liu, H. Li, M. Zhou, R. Jiang, and X. Zhang. Dualgcn: a dual graph convolutional network model to predict cancer drug response. *BMC bioinformatics*, 23(4):1–13, 2022.

[47] W. Maddox, T. Garipov, P. Izmailov, D. Vetrov, and A. G. Wilson. Fast uncertainty estimates and bayesian model averaging of dnns. In *Uncertainty in Deep Learning Workshop at UAI*, volume 8, 2018.

[48] J. Mandel, L. Cobb, and J. D. Beezley. On the convergence of the ensemble kalman filter. *Applications of Mathematics*, 56(6):533–541, 2011.

[49] V. Manfredi, S. Mahadevan, and J. Kurose. Switching kalman filters for prediction and tracking in an adaptive meteorological sensing network. In *2005 Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2005. IEEE SECON 2005.*, pages 197–206. IEEE, 2005.

[50] J. Mena, O. Pujol, and J. Vitria. A survey on uncertainty estimation in deep learning classification systems from a bayesian perspective. *ACM Computing Surveys (CSUR)*, 54(9):1–35, 2021.

[51] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[52] W. Morningstar, S. Vikram, C. Ham, A. Gallagher, and J. Dillon. Automatic differentiation variational inference with mixtures. In *International Conference on Artificial Intelligence and Statistics*, pages 3250–3258. PMLR, 2021.

[53] T. O'Hagan. Dicing with the unknown. *Significance*, 1(3):132–133, 2004.

[54] T. Pearce, M. Zaki, A. Brintrup, N. Anastassacos, and A. Neely. Uncertainty in neural networks: Bayesian ensembling. *Stat*, 1050:12, 2018.

[55] V. Piyush, Y. Yan, Y. Zhou, Y. Yin, and S. Ghosh. A matrix ensemble kalman filter-based multi-arm neural network to adequately approximate deep neural networks, 2023.

[56] D. N. Politis, J. P. Romano, and M. Wolf. *Subsampling*. Springer Science & Business Media, 1999.

[57] R. Pop and P. Fulop. Deep ensemble bayesian active learning: Addressing the mode collapse issue in monte carlo dropout via ensembles. *arXiv preprint arXiv:1811.03897*, 2018.

[58] N. A. Pudlo, K. Urs, R. Crawford, A. Pirani, T. Atherly, R. Jimenez, N. Terrapon, B. Henrissat, D. Peterson, C. Ziemer, et al. Phenotypic and genomic diversification in complex carbohydrate-degrading human gut bacteria. *Msystems*, 7(1):e00947–21, 2022.

[59] B. Ramsundar, P. Eastman, P. Walters, and V. Pande. *Deep learning for the life sciences: applying deep learning to genomics, microscopy, drug discovery, and more*. " O'Reilly Media, Inc.", 2019.

[60] H. Ritter, A. Botev, and D. Barber. A scalable laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning, 2018.

[61] I. Rivals and L. Personnaz. A recursive algorithm based on the extended kalman filter for the training of feedforward neural models. *Neurocomputing*, 20(1-3):279–294, 1998.

[62] M. Roth, G. Hendeby, C. Fritsche, and F. Gustafsson. The ensemble kalman filter: a signal processing perspective. *EURASIP Journal on Advances in Signal Processing*, 2017:1–16, 2017.

[63] K. Shinde, J. Lee, M. Humt, A. Sezgin, and R. Triebel. Learning multiplicative interactions with bayesian neural networks for visual-inertial odometry. *arXiv preprint arXiv:2007.07630*, 2020.

[64] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[65] R. D. Stewart, M. D. Auffret, R. Roehe, and M. Watson. Open prediction of polysaccharide utilisation loci (pul) in 5414 public bacteroidetes genomes using pulpy. *BioRxiv*, page 421024, 2018.

[66] J. Swiatkowski, K. Roth, B. Veeling, L. Tran, J. Dillon, J. Snoek, S. Mandt, T. Salimans, R. Jenatton, and S. Nowozin. The k-tied normal distribution: A compact parameterization of gaussian mean field posteriors in bayesian neural networks. In *International Conference on Machine Learning*, pages 9289–9299. PMLR, 2020.

[67] D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, et al. String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613, 2019.

[68] A. M. Valdes, J. Walter, E. Segal, and T. D. Spector. Role of the gut microbiota in nutrition and health. *Bmj*, 361, 2018.

[69] E. A. Wan and R. Van Der Merwe. The unscented kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*, pages 153–158. Ieee, 2000.

[70] H. Wang and D.-Y. Yeung. Towards bayesian deep learning: A framework and some existing methods. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3395–3408, 2016.

[71] H. Wang and D.-Y. Yeung. A survey on bayesian deep learning. *ACM computing surveys (csur)*, 53(5):1–37, 2020.

[72] M. West and J. Harrison. *Bayesian forecasting and dynamic models*. Springer Science & Business Media, 2006.

[73] A. G. Wilson and P. Izmailov. Bayesian deep learning and a probabilistic perspective of generalization. *Advances in neural information processing systems*, 33:4697–4708, 2020.

[74] M. Wortsman, G. Ilharco, S. Y. Gadre, R. Roelofs, R. Gontijo-Lopes, A. S. Morcos, H. Namkoong, A. Farhadi, Y. Carmon, S. Kornblith, et al. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *International Conference on Machine Learning*, pages 23965–23998. PMLR, 2022.

[75] N. B. Yahia, M. D. Kandara, and N. B. BenSaoud. Integrating models and fusing data in a deep ensemble learning method for predicting epidemic diseases outbreak. *Big Data Research*, 27:100286, 2022.

[76] W. Yang, J. Soares, P. Greninger, E. J. Edelman, H. Lightfoot, S. Forbes, N. Bindal, D. Beare, J. A. Smith, I. R. Thompson, et al. Genomics of drug sensitivity in cancer (gdsc): a resource for therapeutic biomarker discovery in cancer cells. *Nucleic acids research*, 41(D1):D955–D961, 2012.

[77] A. Yegenoglu, K. Krajsek, S. D. Pier, and M. Herty. Ensemble kalman filter optimizing deep neural networks: An alternative approach to non-performing gradient descent. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 78–92. Springer, 2020.

[78] R. Zanetti and K. J. DeMars. Joseph formulation of unscented and quadrature filters with application to consider states. *Journal of Guidance, Control, and Dynamics*, 36(6):1860–1864, 2013.

[79] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[80] H. Zhang, T. Yohe, L. Huang, S. Entwistle, P. Wu, Z. Yang, P. K. Busk, Y. Xu, and Y. Yin. dbcan2: a meta server for automated carbohydrate-active enzyme annotation. *Nucleic acids research*, 46(W1):W95–W101, 2018.

[81] J. Zheng, B. Hu, X. Zhang, Q. Ge, Y. Yan, J. Akresi, V. Piyush, L. Huang, and Y. Yin. dbcan-seq update: Cazyme gene clusters and substrates in microbiomes. *Nucleic Acids Research*, 51(D1):D557–D563, 2023.