North American Actuarial Journal

ISSN: 1092-0277 (Print) 2325-0453 (Online) Journal homepage: www.tandfonline.com/journals/uaaj20

Poisson Mixture Deep Learning Neural Network Models for the Prediction of Drivers' Claims with Excessive Zero Claims Using Telematics Data

Farha Usman, Jennifer S. K. Chan, Alice X. D. Dong & Udi E. Makov

To cite this article: Farha Usman, Jennifer S. K. Chan, Alice X. D. Dong & Udi E. Makov (30 Oct 2025): Poisson Mixture Deep Learning Neural Network Models for the Prediction of Drivers' Claims with Excessive Zero Claims Using Telematics Data, North American Actuarial Journal, DOI: 10.1080/10920277.2025.2570289

To link to this article: https://doi.org/10.1080/10920277.2025.2570289

Published online: 30 Oct 2025.

North American Actuarial Journal, 0(0), 1–29, 2025

ISSN: 1092-0277 print / 2325-0453 online
DOI: 10.1080/10920277.2025.2570289

Routledge
Taylor & Francis Group

OPEN ACCESS Check for updates

Poisson Mixture Deep Learning Neural Network Models for the Prediction of Drivers' Claims with Excessive Zero Claims Using Telematics Data

Farha Usman[1], Jennifer S. K. Chan[1], Alice X. D. Dong[2], and Udi E. Makov[3]
[1]*School of Mathematics and Statistics, University of Sydney, Camperdown, Australia*
[2]*Transdisciplinary School, University of Technology Sydney, Ultimo, Australia*
[3]*Department of Statistics, University of Haifa, Haifa, Israel*

This article explores the role of neural networks in insurance claim prediction using Poisson mixture (PM) deep learning neural networks. The model is designed to handle drivers' insurance claims with excessive zero occurrences by setting a prior probability of a safe group (low claim). The article evaluates PM networks using the negative log-likelihood (NLL) loss function instead of the common choice of mean square error loss function applicable for symmetric data. The NLL loss function captures the asymmetric distribution of claim counts with excessive zeros. The meticulous search for network architecture, employing both manual and Bayesian optimization search techniques, further improves the prediction accuracy of PM density networks. The commitment of this research to pushing the boundaries of predictive analytics is evident throughout the modeling, architecture selection, and evaluation process, positioning the PM deep learning neural network as a noteworthy advancement in insurance claim prediction.

1. INTRODUCTION

In the traditional insurance industry, premiums are calculated based on predictions using only historical annualized numbers of claims, supplemented with demographic information such as driver age, gender, district, and car make. Since the introduction of telematics and related devices, auto insurance has undergone earthshaking changes as these devices enrich two classes of driving data: driving habit data and driving style data. The former includes information on when, where, and how much the insured drives, whereas the latter describes how they drive, such as accelerating, braking, sharp turning, and so on. Driving habit data facilitate premium calculation for the *pay-as-you-drive* (PAYD) usage-based insurance (UBI) policies. Adding further driving habit data, the PAYD policy was later extended to *pay-how-you-drive* (PHYD) using driving behaviors to assess driver claims propensities when telematics is incorporated with data from the global positioning system (GPS) (Williams et al. 2022).

Developing predictive models for auto insurance is challenging due to the increasing volume and complexity of data received from telematic devices. Other confronting factors include policy changes, regulatory scrutiny, and privacy protection. Studies on the methodologies for analyzing telematics data began with traditional statistical models, but attention has been drawn to machine learning (ML) in recent years (Hanafy and Ming 2021). ML offers more advantages over state-of-the-art generalized linear models (GLMs) in terms of its potential to capture nonlinear relationships in the data, thereby providing more accurate claim predictions. Clustering, decision trees, random forests, gradient boosting, explainable boosting machines,

Address correspondence to Farha Usman, School of Mathematics and Statistics, University of Sydney, Camperdown, NSW 2050, Australia. E-mail: farhausman@gmail.com; Jennifer S. K. Chan, School of Mathematics and Statistics, University of Sydney, Camperdown, NSW 2050, Australia. E-mail: jennifer.chan@sydney.edu.au

and neural networks (NNs) are popular ML techniques. These ML techniques can also be combined with statistical models to provide more efficient predictive models.

These models include the decision trees that were incorporated into Poisson predictive models for the frequency of claims (Makov and Weiss 2016). Moreover, Smith et al. (2000) applied decision trees and NNs to study the probability that a policyholder will submit a claim and discussed the impact of prediction accuracy on insurance companies. Gao et al. (2019) proposed Poisson generalized additive models (GAMs) using the two-dimensional speed-acceleration heat maps in addition to some classical risk factors to predict claim frequencies. Gao and Wüthrich (2018) used the K-medoids cluster method to reduce the size of the components in the grid to group drivers with similar heatmaps. They also used principal component analysis to reduce the dimension of the design matrix. Among these ML models and techniques, Paefgen et al. (2013) found that NN outperforms logistic regression and decision tree classifiers using 15 predictor variables in claim events. Weerasinghe and Wijegunasekara (2016) classified claims frequency as low, fair, and high, compared NN with decision tree and multinomial logistic regression models, and found that NN has the best predictive performance.

NN emerges as an efficient method for modeling claim data. This is due to NNs' remarkable capacity for both linear and nonlinear mapping. They surpass the limitations of rigid model structures found in traditional statistical models. NNs excel over GLMs and lasso regression in learning the intricate logic between input features and output targets empirically throughout the training process. They adeptly construct linear and nonlinear relationships, leveraging high-order interactions across layers to discern correlations between input (features) and output (target) variables. NNs proved highly proficient in recognizing complex patterns and relationships within extensive data sets. By adopting customer-specific features, such as age, location, and driving behavior, as inputs, NNs can efficiently identify hidden driving patterns that drive auto insurance claims (Dong et al. 2016).

The idea of NNs is inspired by neuroplasticity models of the human brain. In the simple form of multilayer perceptron (MLP; also called feedforward) without loop connections, NNs form a finite acyclic graph, a finite directed graph without directed cycles (Goodfellow et al. 2016). Sze et al. (2017) believed that the brain learns through changes to the weights associated with the synapses, and different weights respond to different inputs. In a close-up, the structure of the NN resembles a layered cobweb with a series of neurons that take one or more inputs on the left that come from raw data, and the values are fed forward from the input layer to the hidden and output layers. Each layer contains one or more neurons, also called perceptrons. The operation of a neuron involves two steps. In each layer, the input is first linearly combined with layer weights. Then a nonlinear activation function is applied to the result. These hierarchical abstract layers of neurons, as latent variables, perform pattern matching and forecasting.

Basic feedforward NNs have been advanced to increase their flexibility and adaptiveness. Popular extensions include deep learning neural networks (DLNNs) with multiple hidden layers between the input and output layers.

These multiple processing layers reshape the data using hierarchically interconnected layers of artificial neurons (LeCun et al. 2015). Through this high-level abstraction of data, they automatically learn to extract salient features of the input data. The common DLNNs include MLP, recurrent neural network (RNN), long-short-term memory (LSTM), convolutional neural networks (CNN), restricted Boltzmann machines (RBM), autoencoder (AE), and others. Among these models, RNN and LSTM models are particularly suitable for financial time series and sequential trip data; CNN and AE models are widely used in data compression, data denoising, and image recognition for supervised and unsupervised learning, respectively; and RBM models are applied to natural language processing.

In auto insurance claim prediction, Guo et al. (2018) considered DLNNs to capture hidden driving patterns across heterogeneous drivers with telematics data. Dong et al. (2016) used one-dimensional CNNs and RNNs to analyze a large GPS trip dataset. They found that deep learning CNNs and RNNs dramatically outperform traditional ML methods. Simoncini et al. (2018) investigated RNNs with GPS trip data for vehicle classification. To allow frequent updates of driving behavior and, hence, a premium to encourage safe driving, sequential trip data for each update will cover a shorter period. Instead of using sequential trip data, this article considers cross-sectional driving behaviors by creating driving variables (DVs) from telematics data. Specifically, diverse driving events, such as acceleration to 30 miles per hour in 5–11 seconds during rush hours, or smooth and even cornering within the posted speed at the junction during weekday morning and afternoon in a non-rush hour, and so on, are defined, and the frequencies of events are aggregated over a certain period for each driver. The resulting DVs are then standardized across drivers. Hence, networks such as LSTMs, CNNs, and RNNs, which are particularly designed to capture serial dependencies, are not suitable. This article chooses DLNNs, which are popular in various auto insurance problems (Kim et al. 2022).

Although DLNNs can be very flexible in capturing the complex nonlinear effects of DVs on claim counts, they are also subject to the challenge of overfitting, a problem prevalent in complex NN architectures that perform well on training data but may falter on other data sets (Goodfellow et al. 2016; Ying 2019). Regularization in NN mitigates overfitting by creating a robust NN architecture with good generalization when dealing with intricate data (Mishra et al. 2019; Li et al. 2020). Several

strategies are used, including batch normalization (Ioffe and Szegedy 2015; Garbin et al. 2020), dropout (Srivastava et al. 2014; Garbin et al. 2020), and early stopping (Prechelt 2002), which are employed to address the overfitting concerns. Both batch normalization and dropout require tuning of hyperparameters (Mishra et al. 2019; Li et al. 2020) to work well with the network; hence, models should be trained with different combinations of hyperparameters to find an optimal set of hyperparameters. However, this also increases training time (Srivastava et al. 2014; Garbin et al. 2020). Despite this, batch normalization significantly reduces training time by normalizing the input of each layer in the network. It proves to be effective with high learning rates and reduces the number of training steps required for convergence (Ioffe and Szegedy 2015; Garbin et al. 2020). These regularization techniques play a pivotal role in enhancing the generalization capacity of NNs, ensuring their adaptability and resilience when handling complex data sets.

As NNs can be viewed as some complicated and dynamic regression models, they bear some similarities with regression models in which regression parameters are estimated to optimize certain objectives or loss functions. From a statistical modeling perspective, the common mean square error (MSE) loss function is applied to nonparametric regression, essentially matching the first-order moment in the L2 norm and corresponding to a normal distribution for symmetric data. This assumption fails to capture the zero-inflated claim data, often with zero inflation, even though nonlinear activation functions such as sigmoid or, equivalently, logistic link functions can be adopted. More importantly, the MSE loss function does not include data distribution information, and hence, it fails to provide uncertainties of the target variable estimates. Uncertainty quantification has received growing interest in NNs (Amini et al. 2020; Wong et al. 2025).

To tackle this shortcoming, density networks were proposed, adopting negative loglikelihood (NLL) as the loss function to capture different distribution assumptions. In this approach, Fallah et al. (2009) proposed Poisson regressions for claim counts based on NN with six hidden layers. Then they compared the performances in three simulated data sets, one linear and two nonlinear cases, and found that Poisson NNs only improve the prediction in nonlinear environments. They urged NN to have a more refined set of parameters and programming in a linear setting. Yunos et al. (2016) proposed DLNNs for two claim components: claim count and claim severity (claim cost per claim). They used nine different network structures and revealed that the network architecture affects the accuracy of the prediction. Wüthrich and Merz (2019) introduced the art of model blending of the simple generalized linear (GLM) model in a neural net architecture named the combined actuarial neural net (CANN) approach. Sakthivel and Rajitha (2017) compared the zero-inflated hurdle models with the NN in the claim count modeling. Kim et al. (2022) found DLNN is superior in prediction accuracy and faster in computation times than gradient boosting machines, principal component analysis (PCA)-based Poisson regression, PCA-based negative binomial regression, and PCA-based zero inflated Poisson regression models with a combination of two data sets, speeding ticket and insurance claim count data.

Clearly, Poisson density networks using the NLL loss function extend DLNNs to capture the asymmetric claim count distribution, and enable safe and risky driver classification if the predicted claim frequency is lower or exceeds a certain threshold, respectively.

However, this method requires a search for a good threshold as it is not a model-driven parameter. On the other hand, mixture models such as PM facilitate classification using posterior probability estimates and allow differential DVs in the mean functions of the safe and risky driver groups. Following the idea of PM LASSO regression by Usman et al. (2024), this article proposes two-group PM DLNNs. The inclusion of the nonregression weight parameter π for the first mixture component presents significant implementation challenges. To address this, techniques are introduced to iteratively estimate the DLNN parameters $\boldsymbol{\theta}$ conditioned on π and subsequently reestimate π conditioned on $\boldsymbol{\theta}$. To the best of our knowledge, the application of PM DLNNs to telematics data represents a pioneering effort in using density networks to uncover hidden driving patterns for claim frequency prediction and driver classification.

Comparing PM DLNNs to any nonmixture type ML techniques, such as gradient boosting and explainable boosting machines, PM DLNNs integrate both claim prediction and driver classification into a unified modeling approach. This information is used in the premium calculation discussed later. Comparing PM DLNNs with the PM LASSO regression in Usman et al. (2024), the advantage of DLNN lies in its ability to capture complex nonlinear relationships through both nonlinear activation functions and high-order interaction terms. Without having to shrink some features to zero, DLNNs can effectively handle large numbers of features in detecting hidden patterns, especially with techniques like dropout and regularization that prevent overfitting.

This article contributes on multiple fronts, as outlined here. First, it proposes innovative PM DLNNs to learn hidden driving patterns for claim prediction and driver classification. Due to the complexity of PM DLNNs, the architecture search is crucial and is guided by practical considerations, with a nuanced interplay of factors or fine-tuning parameters, including epochs, batch size, learning rates, and optimizers. DLNNs with five hidden layers are meticulously chosen, each playing a pivotal role in discerning intricate patterns within DVs. Apart from architecture search, other techniques, such as batch normalization, early stop, and dropout, are also adopted to mitigate underfitting and overfitting, as well as to enable speedy convergence of parameters.

Second, a notable challenge is to define the mean parameters of the two Poisson distributions in the NLL loss function by incorporating constraints that can distinguish and separate the two distinct mixture groups of safe and risky drivers. These constraints need to align with the data structure to provide efficient separation between the two groups.

Third, the inclusion of the nonregression weight parameter π for the mixture component presents significant implementation challenges. To address this, this article proposes several methodologies, including the iterative estimation of the DLNN parameters denoted by $\boldsymbol{\theta}$ conditioned on π, followed by reestimation of π conditioned on the DLNN parameters $\boldsymbol{\theta}$ until convergence or early stop criteria are attained. The whole PM DLNNs are implemented via `Keras` and `TensorFlow` within the Python environment. The program is provided in the Appendix. Model performance is visualized by the scatter plot of predicted annual claim frequency against the observed and numerical measures, such as NLL and MSE, that measure model fit and prediction accuracy, respectively.

Lastly, the claim frequency and driver group classification estimated from PM DLNNs can generate more accurate risk profiles, assisting insurers in personalized pricing and underwriting by considering individual risk factors. Specifically, following the idea of Usman et al. (2024), we apply the predicted claim frequency and group classification for new drivers to the UBI experience-rating premium pricing method to calculate the risk premium in the pay-as-you-go pricing scheme. As drivers exhibit riskier behaviors, as measured by telematics, the premium will increase, even in the absence of previous claims, for new drivers. To monitor driving behavior and incentivize safe driving, premiums can be recalculated weekly, but the predictive model can be retrained monthly or over other suitable periods, considering the size of the telematics data and training cost. This method offers a more proactive approach to safeguarding the company's financial stability by providing customized premiums that can be regularly updated to reflect actual driving risk. In addition, the estimates of claim frequency and safe/risky driver classification are easily interpretable, supporting transparent and justifiable pricing decisions. As a result, the method enhances the accuracy of auto insurance premium calculations, ultimately contributing to the overall profitability of insurance companies.

This article is structured as follows. Section 2 details the PM DLNN model's proposed structure, the architecture's tuning, and the regularization. Section 3 provides an empirical study with model estimation and selection. Section 4 presents the experimental results of DLNNs. Section 5 concludes the article and discusses future research directions.

2. POISSON MIXTURE DEEP LEARNING NEURAL NETWORKS

To predict annual claim frequency and classify drivers into safe and risky groups, DLNNs are derived within the PM modeling framework. Let y_i denote the observed claim count of driver i, $y_i, i = 1..., N$, N be the number of drivers, and n_i be the policy exposure for driver i. The input features or covariates are J selected DVs, and they are presented as $\boldsymbol{X}_{(1:N)\times(1:J)} = (\boldsymbol{x}_{1.}, ..., \boldsymbol{x}_{N.})^{\top} \in \mathbb{R}^N \times \mathbb{R}^J$. The target variable is the claim count $\boldsymbol{y}_{1:N}$ with exposure $\boldsymbol{n}_{1:N}$ and is used to train the DLNN to predict the claim counts $\boldsymbol{A}_{1:N,\,1:2} = (\boldsymbol{a}_{1.}, ..., \boldsymbol{a}_{N.})^{\top} \in \mathbb{R}^N \times \mathbb{R}^2$ as output where $\boldsymbol{a}_i = (a_{i1}, a_{i2})$ represents the annual predicted claim frequency for the driver i in the safe and risky driver groups.

2.1. Poisson Mixture Regression Model

Poisson regression model is commonly applied to count data, like the number of claims. It is defined as

$$Y_i \sim \text{Poisson}(\mu_i), \mu_i = n_i a_i = n_i \exp(\boldsymbol{x}_{i\bullet}\boldsymbol{\beta}) = \exp(\boldsymbol{x}_{i\bullet}\boldsymbol{\beta} + \log(n_i)) \tag{1}$$

where the observed $\boldsymbol{y} = y_{1:N}$, $\boldsymbol{\beta} = \beta_{0:J}$, $a_i = \exp(\boldsymbol{x}_{i\bullet}\boldsymbol{\beta})$ estimated the annual claim $a_i = y_i/n_i$ for driver i, and $\log(n_i)$ is the off-set parameter in the regression model.

Poisson regression assumes equidispersion. For overdispersed data, the negative binomial (NB) distribution provides extra dispersion. With NB regression, the distribution "Poisson(μ_i)" in Equation (1) is replaced with NB distribution $\text{NB}(\nu, q_i) = \text{NB}(\nu, \mu_i/(\nu - \mu_i))$ where ν is the shape parameter and q_i is the success probability of each trial. NB distribution converges to the Poisson distribution if ν tends to infinity.

The PM model is also popular for modeling unobserved heterogeneity that causes overdispersion. It also facilitates classification. The model assumes G unobserved groups, each with probability π_g, $0 < \pi_g < 1, g = 1, ..., G$ and $\sum_{g=1}^{G} \pi_g = 1$. To classify drivers into safe and risky groups, we set $G = 2$. Assuming that each driver claim Y_i from the driver group g follows a Poisson distribution with mean μ_{ig}, that is, $Y_i \sim \text{Poisson}(\mu_{ig})$ at probability π_g, $g = 1, 2$, the density of PM model is

$$f_{\text{PM}}(y_i|\pi, \mu_{i1}, \mu_{i2}) = \pi f_1(y_i|\mu_{i1}) + (1 - \pi) f_2(y_i|\mu_{i2}) \tag{2}$$

where $\pi = \pi_1$ and $f_g(\cdot|\mu_{ig}), g = 1, 2$ is the probability mass function of Poisson distribution with mean μ_{ig}. The observed data likelihood function is

$$\mathcal{L}_o(\boldsymbol{\theta}) = \prod_{i=1}^{N} \sum_{g=1}^{2} \pi_g f_g(y_i|\mu_{ig}). \tag{3}$$

The marginal predicted total number of claims is

$$\hat{y}_i = \hat{\pi}_{i1}\mu_{i1} + (1 - \hat{\pi}_{i1})\mu_{i2} \tag{4}$$

where μ_{ig} is the predicted claim frequency condition on group g and the posterior probability for driver i in group g is estimated by

$$\eth_{ig} = \frac{\pi_g f_g(y_i|\mu_{ig})}{\sum_{g'=1}^{2} \pi_{g'} f_{g'}(y_i|\mu_{ig'})}. \tag{5}$$

Based on $\eth_{ig}$, driver i can be classified to group g^* if

$$g^* = \arg\max_g \eth_{ig}.$$

However, for a new driver i with input features $\boldsymbol{x}_i$ from the telematic device but unknown claim frequency y_i, the posterior probability $\eth_{ig}$ cannot be evaluated directly. Then one can simulate $y_i^{(h)}, , 1, ..., H$ from the PM model $f_{\text{PM}}(y_i|\pi, \mu_{i1}, \mu_{i2})$ in Equation (2) with parameters $(\mu_{i1}, \mu_{i2}, \pi_{i1})$ determined from the trained model and calculate $\eth_{ig}^{(h)}|y_i^{(h)}$. Then the posterior probability $\eth_{ig}$ estimate can be the mean or median of $\eth_{ig}^{(h)}, 1, ..., H$.

Equation (5) also forms the E-step of the expectation–maximization (EM) algorithm (Moon 1996) for estimating the missing group membership. When the missing is estimated, the M-step estimates all model parameters using complete data. However, for NNs with large number of parameters, this method fails and a gradient descent algorithm (see Section 2.4) is used instead.

Note that apart from using Equation (4) to predict the total claim frequency, an alternative estimate making use of the more informative posterior probability is given by

$$\hat{y}_i = \hat{\eth}_{i1}\mu_{i1} + (1 - \hat{\eth}_{i1})\mu_{i2}. \tag{6}$$

Results from data application show that Equation (6) provides better in-sample MSE by adopting more driver information. Accordingly, the annual number of claims is given by

$$\hat{a}_i = \hat{\eth}_{i1}\hat{a}_{i1} + (1 - \hat{\eth}_{i1})\hat{a}_{i2} \tag{7}$$

where $\hat{a}_{ig}$ is the predicted annual claim from the network output in (29). Lastly, Usman et al. (2024) conducted a test for the overdispersion and got p values 0.0482 and 0.0477, which are just marginally significant for the two Poisson models. Moreover, they also considered the zero-inflated Poisson (ZIP) model, as a special case of the PM model with a degenerated first component for the zero claims. Results show that the PM model outperforms the ZIP model, as the PM model offers greater model flexibility. Considering the additional variability introduced by PM models, they concluded that PM models are suitable for capturing the overdispersion in the data.

2.2. Architecture of DLNNs

For the PM model in Equation (2), the means $\mu_{ig} = n_i a_{ig}$ where a_{ig} adopt the DLNN structure. Assume that there are $\mathfrak{Q}$ hidden layers and each hidden layer contains m_l neurons where $\boldsymbol{m} = (m_1, ..., m_{\mathfrak{Q}})$ and $m_0, m_{\mathfrak{Q}+1}$ are the size of input features (layer 0) and output targets (layer $\mathfrak{Q} + 1$), respectively. To find the predicted annual claim matrix $\boldsymbol{A}$, DLNNs use forward propagation by departing from the left input layer to the hidden layers and arriving at the output layer using the MLP approach. The algorithm needs good weights linking the layers to get accurate predictions.

In each layer l, let the weights be $\mathrm{w}_{jk}^{(l)}, j = 1, ..., m_{l-1}, k = 1, ..., m_l$, the biases be $b_k^{(l)}$ and the k-th output neuron $a_{ik}^{(l)}$, which is obtained by first weighting $\boldsymbol{a}_i^{(l-1)} = (a_{i1}^{(l-1)}, ..., a_{im_{l-1}}^{(l-1)})^\top \in \mathbb{R}^{m_{l-1}}$ with the weight vector $\mathbf{w}_k^{(l)} = (\mathrm{w}_{1k}^{(l)}, ..., \mathrm{w}_{m_{l-1}k}^{(l)})^\top \in \mathbb{R}^{m_{l-1}}$. Then summing and bias-adjusting are applied as follows

$$z_{ik}^{(l)} = \sum_{j=1}^{m_{l-1}} \mathrm{w}_{jk}^{(l)} a_{ij}^{(l-1)} + b_k^{(l)} = \mathbf{w}_k^{(l)\top} \boldsymbol{a}_i^{(l-1)} + b_k^{(l)} = \mathsf{f}(\boldsymbol{a}_i^{(l-1)} | \mathbf{w}_k^{(l)}, b_k^{(l)}), \;\; k = 1, ..., m_l \tag{8}$$

where $\mathsf{f}(\cdot|\mathbf{w}_k^{(l)}, b_k^{(l)})$ is the linear function and $a_{ik}^{(0)} = x_{ik}$ for the input layer. Lastly, $z_{ik}^{(l)}$ is nonlinearly transformed to $a_{ik}^{(l)}$ as follows:

$$f_l(z_{ik}^{(l)}) = a_{ik}^{(l)}$$

where the activation function $f_l(\cdot)$ is a nonlinear function to provide a reasonable output range. The choices of $f_l(\cdot)$ are reviewed in Equations (14) to (19).

In summary,

$$a_{ik}^{(l)} = f_l(\mathbf{w}_k^{(l)\top} \boldsymbol{a}_i^{(l-1)} + b_k^{(l)}) = f_l\left(\sum_{j=1}^{m_{l-1}} \mathrm{w}_{jk}^{(l)} a_{ij}^{(l-1)} + b_k^{(l)}\right) = f_l \circ \mathsf{f}(\boldsymbol{a}_i^{(l-1)} | \mathbf{w}_k^{(l)}, b_k^{(l)}) \tag{9}$$

where $f_l \circ \mathsf{f}$ is the composite mapping in each layer l and

$$\boldsymbol{a}_i^{(l)} = f_l(\boldsymbol{W}^{(l)\top} \boldsymbol{a}_i^{(l-1)} + \boldsymbol{b}^{(l)}) = f_l \circ \mathsf{f}(\boldsymbol{a}^{(l-1)} | \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}) := f_{l, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}}(\boldsymbol{a}^{(l-1)}) \tag{10}$$

where $\boldsymbol{a}_i^{(l)} = (a_{i1}^{(l)}, ..., a_{im_l}^{(l)})^\top$ is an $m_l \times 1$ dimensional neuron vector, $\boldsymbol{W}^{(l)} = (\mathbf{w}_1^{(l)}, ..., \mathbf{w}_{m_l}^{(l)}) = (\mathrm{w}_{jk}^{(l)}) \in \mathbb{R}^{m_{l-1}} \times \mathbb{R}^{m_l}$ is a weight matrix, and $\boldsymbol{b}^{(l)} = (b_1^{(l)}, ..., b_{m_l}^{(l)})^\top \in \mathbb{R}^{m_l}$ is a bias vector. Finally, the output $\boldsymbol{a}_i^{(l)}$ is taken as the input for the next layer $l+1$, and so on.

In the output layer $\mathfrak{L}+1$ with $m_f = G = 2$ neurons, the output $\boldsymbol{a}_i^{(\mathfrak{L}+1)} = (a_{i1}^{(\mathfrak{L}+1)}, a_{i2}^{(\mathfrak{L}+1)})$ predicts the annual claim for driver i in group g, $g = 1, 2$. The predicted annual claim and claim for driver i in group g are given by

$$\hat{a}_{ig}(\boldsymbol{\theta}) = f_{cg}(a_{ig}^{(\mathfrak{L}+1)}) \quad \text{and} \quad \hat{y}_{ig} = \mu_{ig} = n_i \hat{a}_{ig}(\boldsymbol{\theta}) \tag{11}$$

where $a_{ig}^{(\mathfrak{L}+1)}$ are the output neurons and $f_{cg}(\cdot)$ in Equation (29) are transformation functions to ensure that the two groups are well separated. Then the overall claim $\hat{y}_i$ is given by Equation (6).

Over all layers, DLNN defines a function f_{DLNN} as a mapping from input data features to the desired annual number of claims by group, $\mathbf{X} \to \mathbf{A}$ via all hidden layers with activation functions $f_{l, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}}(\cdot)$ in Equation (10). The deep predictor becomes a composite mapping:

$$\boldsymbol{a}_i^{(\mathfrak{L}+1)} = f(\boldsymbol{x}_i|\boldsymbol{\theta}) = f_{\mathfrak{L}+1, \boldsymbol{W}^{(\mathfrak{L}+1)}, \boldsymbol{b}^{(\mathfrak{L}+1)}} \circ ... \circ f_{1, \boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}}(\boldsymbol{a}^{(0)}) \in \mathbb{R}^{+,2} \tag{12}$$

where $\boldsymbol{a}^{(0)} = \boldsymbol{x}_i$. Here, the mapping $f(\boldsymbol{X})$ can be modeled as a superposition. Figure 1 visualizes the DLNN structure as just described.

The vector of all regression parameters is $\boldsymbol{\theta} = (\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, ..., \boldsymbol{W}^{(\mathfrak{L}+1)}, \boldsymbol{b}^{(\mathfrak{L}+1)})$. The number M of all regression parameters in $\boldsymbol{\theta}$ is calculated as

$$M = \sum_{l=1}^{\mathfrak{L}+1} [m_l \times (m_{l-1} + 1)] \tag{13}$$

and they are estimated through backward propagation by optimizing the loss function as described in Sections 2.3 and 2.4. Table A1 in Appendix A illustrates the calculation for our choice of architecture discussed in Section 2.5, except the batch normalization, which is also explained in Section 2.5.

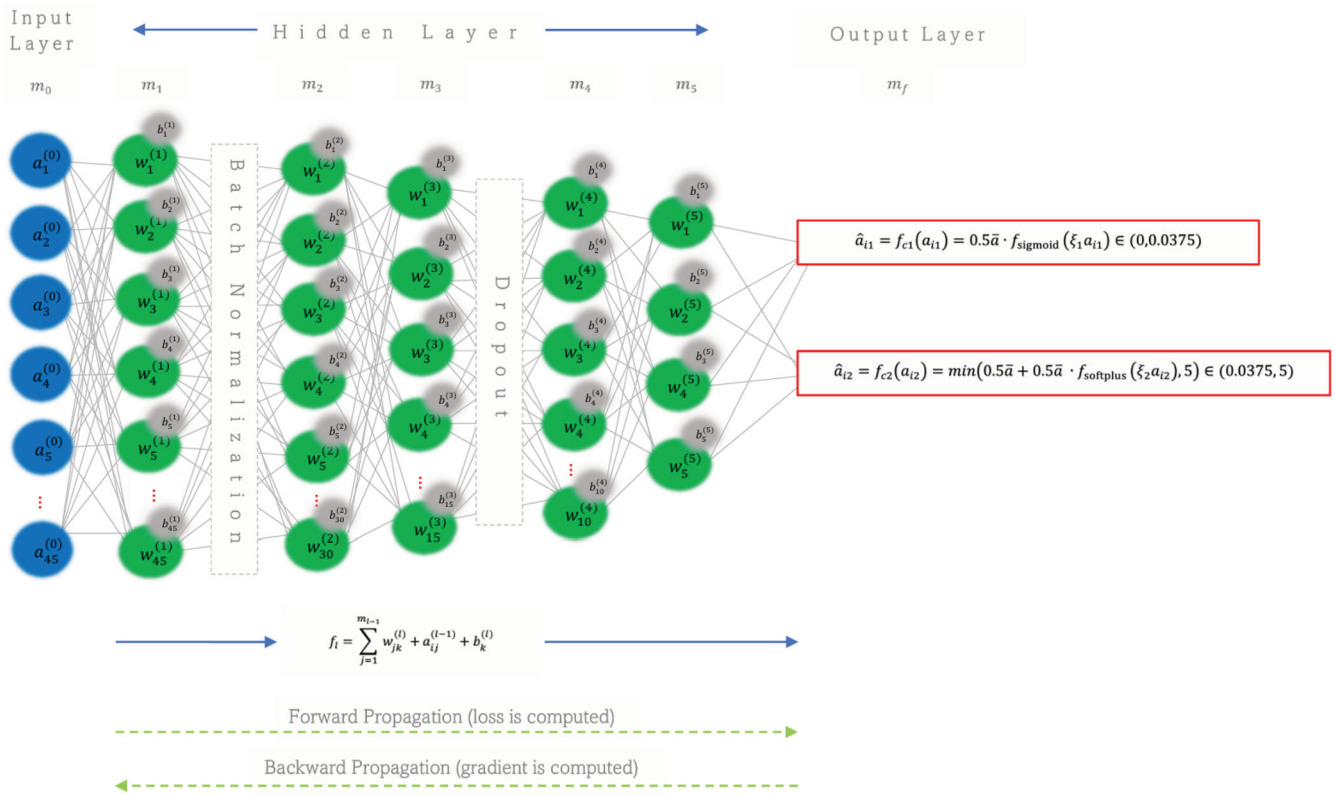


FIGURE 1. Poisson Mixture Deep Learning Neural Network (PM-DLNN) with $m_0 = 45$ Features, $\mathfrak{L} = 5$ Hidden Layers, $\boldsymbol{m} = (45, 30, 15, 10, 5)$ Neurons in the Hidden Layers, and $m_f = 2$ Neuron in the Output Layer.

The activation function $f_l(\cdot)$ captures nonlinearity in the hidden layers and transforms the output to a desirable range, such as positive for the predicted annual number of claims in the output layer. The activation function prefers to be continuous, monotonically increasing, differentiable for gradient calculation in backward propagation, and smooth to reduce the oscillations and noise during the training. The choices of activation functions include `linear`, rectified linear unit (`ReLU`), exponential linear unit (`ELU`), leaky rectified linear unit (`LeakyReLU`), `Softplus`, and `Sigmoid`:

$$f_{linear}(z) = z \in (-\infty, \infty), \tag{14}$$

$$f_{ReLU}(z) = \max(0, z) \in (0, \infty), \tag{15}$$

$$f_{\texttt{ELU}}(z) = \begin{cases} z, & \text{if } z > 0, \\ \xi(e^z - 1), & \text{if } z \leq 0, \end{cases} \tag{16}$$

$$f_{LeakyReLU}(z) = \begin{cases} z, & \text{if } z \geq 0, \\ \xi z, & \text{if } z < 0, \end{cases} \tag{17}$$

$$f_{\texttt{softplus}}(z) = \log(1 + e^z), \tag{18}$$

$$f_{\texttt{sigmoid}}(z) = \frac{e^z}{1 + e^z} \in (0, 1), \tag{19}$$

where ξ is a hyperparameter. The `ReLU` function is faster to train because it has a fixed derivative (slope). However, when $z < 0$, $f_{ReLU}(z) = 0$ always, making the network unable to update the weights, a phenomenon called "neuron death." The `ELU` function solves this problem when $z < 0$ by allowing a negative output range $(-1, 0)$, whereas `LeakyReLU` allows a small amount of information to flow when $z < 0$ and sets ξ at 0.01 (Sharma et al. 2017). `Softplus` is a smoothed version of `ReLU` that avoids the zero gradients when $z \leq 0$.

Section 2.6.1 provides details of hyperparameter tuning, including the number of hidden layers $\mathfrak{L}$ and the number of neurons $\boldsymbol{m}$ in each layer. The results find $\mathfrak{L} = 5$ hidden layers and $\boldsymbol{m} = (45, 30, 15, 10, 5)$ neurons in the hidden layers with $m_0 = 45$ and $m_{\mathfrak{L}+1} = 2$ neurons in the input and output layers, respectively, providing a good architecture to predict annual number of claims of each component of the mixture. The `Keras`'s `Sequential()` model in `Python` is used to build the models. The choice of architecture $\mathfrak{L} = 5$ and $\boldsymbol{m}$ is explained in Sections 2.6.1 and 2.6.2. The dropout layer for regularization and batch layer are explained in Section 2.5. The dropout layer does not involve new parameters.

2.3. Loss Function

The regression parameters $\boldsymbol{\theta}$ are estimated to optimize some objective functions $\mathcal{J}(\boldsymbol{\theta})$, that is,

$$\hat{\boldsymbol{\theta}} = \arg\min_{\theta} \mathcal{J}(\boldsymbol{\theta})$$

for training data. The MSE is a common metric for evaluating prediction accuracy by comparing observed and predicted number of claims. Hence, the MSE is defined as

$$\mathcal{J}_{\text{MSE}}(\boldsymbol{\theta}, \pi) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i(\boldsymbol{\theta}))^2, \text{ where } \hat{y}_i(\boldsymbol{\theta}) = n_i[\pi\hat{a}_{i1}(\boldsymbol{\theta}) + (1-\pi)\hat{a}_{i2}(\boldsymbol{\theta})] \tag{20}$$

is the predicted claim according to the mixture distribution in Equation (2) and $\hat{a}_{ig}(\boldsymbol{\theta})$ is given by Equation (11).

However, when MSE is used as a loss function, it essentially assumes a normal distribution for symmetric data, since the negative loglikelihood (NLL) function $-\ell$ for normal distribution includes the MSE term apart from a normalizing constant, and σ^2 is also assumed to be a constant. Hence, the shortcomings of the MSE loss function are obvious when dealing with skewed claim data exhibiting zero inflation, as shown in the right plot of Figure 2. More importantly, from the moment-matching perspective, the use of the MSE loss function does not provide information regarding the uncertainty of the predicted claim estimates.

To capture the asymmetry of the claim counts, Poisson and PM models are proposed in Section 2.1, and the PM NLL can be adopted for the loss function. The NLL loss function for PM models is defined as

$$\mathcal{J}_{\text{NLL}}(\boldsymbol{\theta}, \pi) = -\frac{1}{N}\sum_{i=1}^{N}\log f_{\text{PM}}(y_i; \pi, \mu_{i1}, \mu_{i2}) = -\frac{1}{N}\sum_{i=1}^{N}\log\left[\pi\frac{\mu_{i1}^{y_i}\mathrm{e}^{-\mu_{i1}}}{y_i!} + (1-\pi)\frac{\mu_{i2}^{y_i}\mathrm{e}^{-\mu_{i2}}}{y_i!}\right] \tag{21}$$

where $f_{\text{PM}}(y_i; \pi, \mu_{i1}, \mu_{i2})$ is also given in Equation (2).

Lastly, instead of updating the model's weights based on $\mathcal{J}(\boldsymbol{\theta})$ computed from the entire training dataset, training is often performed on smaller subsets of the training data, known as *batches* or *mini-batches*. The optimization algorithm using gradient descent and applying to batches is called batch gradient descent.

The advantages of using mini-batches include computational efficiency, taking advantage of parallelization, especially when working with large datasets, regularization effects by introducing randomness of mini-batch to prevent overfitting, and memory efficiency to reduce memory load. The process of training the DLNN based on $\mathcal{J}(\boldsymbol{\theta})$ using mini-batches with batch size $\mathcal{B}$ is commonly referred to as *mini-batch training*. The mini-batch size $\mathcal{B}$ is with respect to the training data size $N_T = \mathsf{p}N$, where p is the proportion of data assigned to training, and the validation data size $N_V = (1-\mathsf{p})N$. The `Python` codes to calculate the NLL and MSE losses are given in code list 1 in Appendix B and code list 2 in Appendix C.

2.4. Model Optimization

This article considers some commonly used optimizers, including stochastic gradient descent (`SGD`), adaptive gradient algorithm (`AdaGrad`), adaptive learning rate method (`AdaDelta`), and adaptive moment estimation (`Adam`) in the architecture search in Equation (27) to optimize $\mathcal{J}(\boldsymbol{\theta})$. All of these methods use mini-batch training.

When replacing the second-order derivatives of the loss function with a constant learning rate, `SGD` is known as the first-order Newton–Raphson method (Drori 2022). The algorithm is formulated as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \zeta \left. \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \tag{22}$$

where $\boldsymbol{\theta}_t$ are the model parameters in the t-th iteration, $\mathcal{J}(\boldsymbol{\theta})$ is the loss function, ζ is the learning rate, and $\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t}$ is the gradient of the loss function $\mathcal{J}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta} = \boldsymbol{\theta}_t$. With mini-batch training, this method is called mini-batch `SGD` (MSGD).

`AdaGrad` is an improved version of the `SGD` algorithm that increases ζ for sparser parameters and decreases ζ for less sparse ones. This strategy often improves convergence performance where the data are sparse and the sparse parameters are more informative. This optimizer adjusts ζ depending on the squared sum of past partial derivatives. The update rule is given by

$$\theta_{j,t+1} = \theta_{i,t} - \left(\frac{\zeta}{\sqrt{v_{jt}} + \epsilon} \right) \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j}|_{\theta_j=\theta_{jt}} \tag{23}$$

where $v_{j,t+1} = v_{j,t} - \left(\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j}|_{\theta_j=\theta_{jt}} \right)^2$ and ϵ is a smoothing term to avoid zero value. However, the main flaw of `AdaGrad` is its accumulation of squared gradients in the denominator, which causes the learning rate to shrink so that the model will not learn more when the learning rate is almost zero (Zeiler 2012).

To resolve these issues, `AdaDelta` reduces its aggressive, monotonically decreasing learning rate by using a decay rate in

$$v_{t+1} = \varphi v_t - (1 - \varphi) \left(\left. \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j} \right|_{\theta_j=\theta_{jt}} \right)^2, \tag{24}$$

where φ is a decay rate, and the updating formula is given by Equation (23). In this way, it restricts the window of accumulated past gradients to a fixed size. It avoids storing previously squared gradients by recursively defining the sum of gradients as a decaying average of all past squared gradients so that it depends only on the previous standard and current gradient. See Qu et al. (2019) for an application of `AdaDelta`.

Lastly, the `Adam` algorithm (Zhang 2018) is another advanced algorithm, defined as

$$\theta_{j,t+1} = \theta_{jt} - \zeta \frac{\hat{\varsigma}_t}{\sqrt{\hat{v}_{jt}} + \epsilon} \tag{25}$$

where the first-order and second-order moment estimates of the gradient $\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j}$ are $\varsigma_t = \varphi_1 \varsigma_{j,t-1} + (1 - \varphi_1) \frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j}|_{\theta_j=\theta_{jt}}$ and $v_t = \varphi_2 v_{j,t-1} + (1 - \varphi_2)(\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j})^2|_{\theta_j=\theta_{jt}}$, respectively, their average values are $\hat{\varsigma}_t = \varsigma_t/(1 - \varphi_1^t)$ and $\hat{v}_t = v_t/(1 - \varphi_2^t)$, and φ_1 and φ_2 are the decay rates taking values 0.9 and 0.999, respectively.

2.5. Regularization

Regularization is important in NNs because it helps prevent overfitting, a phenomenon in which a model learns the training data too well, capturing noise, specific patterns, and overly complex representations that do not generalize well to new unseen data. Normalization, batch normalization (`BatchN`), dropout (`DropOut`), and early stopping are introduced as powerful regularization techniques to mitigate the challenges of both under- and overfitting in NNs and to ensure stability and efficiency of the learning process.

First, we apply standardization $\boldsymbol{x}_{\text{stand}} = (\boldsymbol{x} - \text{mean}(\boldsymbol{x}))/\text{sd}(\boldsymbol{x})$ as one normalization way to normalize all features before fitting to DLNN so that each feature will have $\mu = 0$ and $\sigma = 1$. `Python` code `StandardScaler()` is used to perform the standardization.

Another regularization is the batch normalization `BatchN`, an important technique in mini-batch training. By adding noise in forming random batches during training, it allows the model to train more consistently and converge more quickly for each

input variable across layers over a mini-batch (Ioffe and Szegedy 2015; Santurkar et al. 2019). It also helps to address the vanishing and exploding gradient problems, making it easier for gradients to flow through the network, reducing sensitivity to initialization and enabling higher learning rates.

`BatchN` is suggested to apply to layer 1 so that for any batch with size $\mathcal{B}$, it transforms each input $a_{ik}^{(1)}, k = 1, ..., m_1$ neurons, $i = 1, ..., \mathcal{B}$ observations within the batch, by standardization and transformation using four parameters, namely, the shifting parameters $\mathfrak{a}_k$, the scaling parameter $\mathfrak{b}_k$, the mean moving average (MA) parameter μ_k, and the standard deviation MA parameter σ_k given by Ioffe and Szegedy (2015):

$$f_{BatchN}(a_{ik}^{(1)}) = \mathfrak{b}_k \left(\frac{a_{ik}^{(1)} - \mu_k}{\sigma_k} \right) + \mathfrak{a}_k,$$

where $f_{BatchN}(\cdot)$ is the `BatchN` transformation function and the estimates of μ_k and σ_k condition on the current batch are

$$\hat{\mu}_k = \frac{1}{\mathcal{B}} \sum_{i=1}^{\mathcal{B}} a_{ik}^{(1)} \text{ and } \hat{\sigma}_k = \sqrt{\frac{1}{\mathcal{B}-1} \sum_{i=1}^{\mathcal{B}} (a_{ik}^{(1)} - \hat{\mu}_k)^2}.$$

Parameters μ_k, σ_k are nontrainable parameters, whereas $\mathfrak{a}_k$, $\mathfrak{b}_k$ are trainable, similar to the weight and bias parameters $\boldsymbol{W}^{(l)}$, $\boldsymbol{b}^{(l)}$. In Table A1, $m_1 = 45$; hence the number of parameters for `BatchN` is $4 \times 45 = 180$.

`DropOut` is also a popular regularization technique where randomly selected neurons in a hidden layer are dropped out during training at a probability $\mathfrak{p}$ such as 5%, 10%, 20%, and so on. Their contribution to the activation of downstream neurons is temporally removed on the forward pass, and any weight updates are not applied to the neuron on the backpropagation.

Lastly, as the epoch t increases, the loss function $\mathcal{J}(\cdot)$ often decreases continuously, resulting in a nearly perfect fit of the training data. Early stopping refers to some auto-stop criteria such as a certain total number of iterations or epochs $\mathcal{E}$ hasreached or the predetermined lower threshold value that the loss function $\mathcal{J}(\cdot)$ has attained (You et al. 2019). The parameter *patience* $\mathcal{P}$ in early stopping refers to the number of consecutive epochs with no improvement in the performance metric of the validation set before the training process is halted. The performance metric can be the loss function or accuracy measures, such as MSE. *Determining the appropriate patience value can be challenging and crucial for achieving the best results.*

In detecting early stopping, one can exclude initial epochs, which could exhibit transient behavior, and focus on stabilizing phases of the training process. This strategy provides flexibility in managing early stopping, allows more informed decisions regarding model convergence, and prevents premature stopping during the warm-up period.

2.6. Network Architecture

2.6.1. Hyperparameter Tuning

Hyperparameters of DLNNs, including the number of hidden layers $\mathfrak{L}$, the number of neurons m_l in each layer, the activation function $f_l(\cdot)$, the mini-batch size $\mathcal{B}$, the number of epoch $\mathcal{E}$, the learning rate ζ, the dropout rate $\mathfrak{p}$, and the patience $\mathcal{P}$, need to be determined. The following are some guidelines.

First, $\mathfrak{L}$ is an important parameter that describes the architecture of a NN. DLNNs need multiple layers to learn more detailed and abstract relationships within the data. However, more complicated networks require more data and regularization techniques, such as early stopping and dropout, to avoid overfitting the data. If the data have large dimensions or features, $\mathfrak{L}$ is suggested to be 3 to 5. Moreover, a better loss function can increase network power and potentially reduce $\mathfrak{L}$.

Second, the number of hidden neurons $\boldsymbol{m} = (m_1, ..., m_{\mathfrak{L}})$ in each layer should be determined. Some researchers suggest it should be between the size of the input layer m_0 and the output layer $m_{\mathfrak{L}+1}$. In the first hidden layer (m_1), the size should be less than twice the size of the input layer ($J = m_0$), that is, $m_1 < 2m_0$ (Boger and Guterman 1997; Berry and Linoff 2004) or $1 < m_1 < 66$ (Tang and Fishwick 1993) or $m_1 = (2/3)m_0$ (Karsoliya 2012) or $m_1 = m_0/2$ (Kang 1991) or $0.7m_0 < m_1 < 0.9m_0$ (Buyrukoğlu et al. 2021). On the other hand, Liu et al. (2007) proposed a new criterion based on the estimated signal-to-noise-ratio figure (SNRF) to optimize the number of hidden neurons $\boldsymbol{m}$ to avoid overfitting. Instead of using a separate validation set to detect overfitting, SNRF can quantitatively measure the useful information left unlearned so that overfitting can be automatically detected from the training error. However, the calculation of SNRF is more complicated.

The third choice is the learning rate ζ, which is fixed for the `SGD` algorithm according to Equation (22). A large ζ allows the model to learn more quickly, at the cost of skipping the optimal weights and arriving at a suboptimal set of weights (Ioffe and Szegedy 2015). On the other hand, reducing ζ improves the likelihood of convergence but requires more iterations to find

the optimal solution. Learning rates ζ can also be searched. Goodfellow et al. (2016) suggest searching ζ from the set {0.1, 0.01, 0.001, 0.0001, 0.00001}. Bengio (2012) suggests that a default ζ value of 0.01 typically works for standard DLNNs. Alternatively, optimizers defined in Equations (23) to (25) update the learning rate for each network weight individually.

The fourth choice is the batch size $\mathcal{B}$. Masters and Luschi (2018) stated that training of DLNNs is typically based on mini-batches to reduce memory traces and allow parallel running to shorten training times. The batch size $\mathcal{B}$ often depends on data, model architecture, and the trade-off of training time, memory usage, regularization, and accuracy. Common batch sizes $\mathcal{B}$ are powers of 2 (e.g., 32, 64, 128) for efficiency reasons, and generally, $\mathcal{B} = 32$ is a good initial choice. Larger $\mathcal{B}$ leads to shorter training time, higher memory usage, and possibly lower accuracy. Another issue is the vanishing or exploding gradients when estimating the weight parameters. These problems can be addressed using techniques such as batch normalization, gradient clipping (rescaling the gradient to keep it small), or other optimizer techniques, discussed in Section 2.4.

Lastly, setting the patience $\mathcal{P}$ in early stopping is a trade-off. A smaller $\mathcal{P}$ may stop training early and save computational resources but might result in premature stopping if the model has not converged. On the other hand, a larger $\mathcal{P}$ allows more epochs for potential improvement but may waste resources and risk overfitting. The appropriate value for $\mathcal{P}$ depends on various factors, such as model complexity, epoch, batch size, learning rates, convergence speed, and so on. Moreover, the exclusion of certain initial epochs allows for a better assessment of models' convergences and prevents premature stopping at very early epochs.

In conclusion, searching for these hyperparameters $\mathfrak{9}$ is very important to optimize some objective functions. These optimal hyperparameters provide some good DLNN architectures for optimizing the loss function $\mathcal{J}(\boldsymbol{\theta})$ with respect to the network parameters $\boldsymbol{\theta}$ in Equation (21).

2.6.2. Hyperparameter Optimization

The tuning of hyperparameters $\mathfrak{9}$ including $(\mathfrak{L}, \boldsymbol{m}, \mathfrak{p}, f_l, \mathcal{E}, \mathcal{B}, \zeta, \mathcal{P}, \texttt{optimizer})$ is to an optimal set $\mathfrak{9}$ that also optimizes the loss function $\mathcal{J}(\boldsymbol{\theta}|\mathfrak{9})$ from Section 2.4, conditioned at a given architecture defined by $\mathfrak{9}$. To find the optimal set of hyperparameters, there are four methods. *Manual* search picks hyperparameter sets based on the guidelines in Section 2.6.1 to find some good sets of $\mathfrak{9}$. *Grid* search refers to a search over a grid of hyperparameter space for a set of $\mathfrak{9}$ over all combinations that achieves the optimal of some objective functions, such as optimal validation NLL or accuracy. *Random* search is similar to grid search but considers a prespecified number of $\mathfrak{9}$ sets randomly drawn from the hyperparameter space. Lastly, *Bayesian optimization* (BO) uses advanced optimization techniques to improve the search speed using past performances. This article employs two search methods.

2.6.2.1. Manual Search In this method, the manual search (MAN) is combined with the guidelines in Section 2.6.1 to set $\mathfrak{L} = 5$ and $\boldsymbol{m} = (45, 30, 15, 10, 5)$. Activation functions are set as $f_{1:4} = f_{\texttt{Linear}}$ in Equation (14) and $f_5 = f_{\texttt{ELU}}$ in Equation (16) to ensure the output neurons are nonnegative. Then a large number of hyperparameter combinations are experimented with, with early stop activated only after the 50th epoch, providing the model with an opportunity to stabilize and avoid interference with the initial exploratory phase of training. The manual search process starts with identifying the optimal patience value $\mathcal{P} = 1, 2, 3, 4, 5$ when fixing $\mathcal{E} = 400$ and optimizer `AdaDelta` due to its efficiency in handling complex optimization problems. Then it experiments with different values of $\mathcal{B}$, ζ, and $\mathfrak{p}$, and the suitable values are $\mathcal{B} = 100, 110$, $\zeta = 0.0097, 0.01$, and $\mathfrak{p} = 0.05, 0.10, 0.20$. Each architectural variation may demand different parameter configurations to achieve optimal performance. The choices of $\mathcal{B}$ are not exactly the powers of 2 but are close to 128. In summary, the choices are

$$\begin{aligned} &\textit{Manual}: \mathfrak{L} = 5, \quad \boldsymbol{m} = (45, 30, 15, 10, 5), \quad \mathfrak{p} = 0.05, 0.10, 0.20, \quad f_{1:4} = f_{\texttt{Linear}}, f_5 = f_{\texttt{ELU}}, \\ &\mathcal{E} = 400, \mathcal{B} = 100, 110, \zeta = 0.0097, 0.01, \mathcal{P} = 1, ..., 5, \texttt{optimizer} = \texttt{AdaDelta}. \end{aligned} \tag{26}$$

The choice of $\mathfrak{p}$ also depends on the convergence behavior of the models, and the choice of $\boldsymbol{m}$ (Table A1) is to ensure that the total number of model parameters M will be less than the training data size $N_T = \mathfrak{p}N$. This rule prevents overfitting and enhances the network's capability to capture meaningful patterns.

2.6.2.2. Bayesian Optimization Bayesian optimization (BO) search (Shahriari et al. 2016; Klein et al. 2017; Kandasamy et al. 2018; Masum et al. 2021) uses a probability function based on hyperparameters. It is a technique for tuning the hyperparameter set $\mathfrak{9}$ consisting of $\mathfrak{L}$, $\boldsymbol{m}$, $\mathcal{E}$, $\mathcal{B}$, ζ and the optimizer to optimize the loss function $\mathcal{J}(\boldsymbol{\theta}|\mathfrak{9})$ given the architecture $\mathfrak{9}$. This loss function can be chosen to be NLL or MSE. The strategy is to treat $\mathcal{J}(\boldsymbol{\theta})$ as a random function and place a prior over it. It uses surrogate models like Gaussian processes (GP) to define a prior distribution. Starting with $\mathfrak{n}$ sets of hyperparameters $\mathfrak{9}_{1:\mathfrak{n}}$ and function evaluations $\mathcal{J}(\boldsymbol{\theta}|\mathfrak{9}_{1:\mathfrak{n}})$, which are treated as data, the GP prior of $\mathcal{J}(\boldsymbol{\theta})$ is updated to form a GP of the posterior distribution. The posterior distribution, in turn, is used to construct an acquisition function (or selection function), such as expected improvement (EI),

$$\mathrm{EI}_{\mathfrak{n}}(\vartheta) = \int_{-\infty}^{\infty} \max(\mathcal{J}(\boldsymbol{\theta}|\vartheta_{\mathfrak{n}}^{*}) - \mathcal{J}(\boldsymbol{\theta}|\vartheta), 0) \mathrm{f}_{\mathrm{GP}}(\mathcal{J}(\boldsymbol{\theta}|\vartheta_{1:\mathfrak{n}})) \mathrm{d}\mathcal{J}(\boldsymbol{\theta}|\vartheta)$$

where $\vartheta_{\mathfrak{n}}^{*} := \mathrm{argmax}_{\mathfrak{i}\in 1:\mathfrak{n}}\ \mathrm{EI}(\vartheta_{\mathfrak{i}})$ and $f_{\mathrm{GP}}(\mathcal{J}(\vartheta|\vartheta_{1:\mathfrak{n}}))$ is the density of the posterior GP. Then the next ϑ to select is $\vartheta_{\mathfrak{n}+1} = \mathrm{argmax}\ \mathrm{EI}_{\mathfrak{n}}(\vartheta)$ based on the updated posterior. The search strategy iteratively chooses the next set of hyperparameters $\vartheta_{\mathfrak{n}+1}$ according to the acquisition function EI (ϑ) that balances exploration (sampling areas where the surrogate model is uncertain) and exploitation (sampling areas where the surrogate model predicts high performance). The process is repeated for a predefined number of iterations or until a stopping criterion is met. The advantage of BO is that it efficiently explores the hyperparameter space and converges to the optimal ϑ with a relatively smaller number of evaluations compared to traditional grid search or random search methods.

Guided by the manual exploration, the BO search is conducted for the hyperparameter spaces,

$$BO : \mathcal{E} \in [250, 450], \mathcal{B} \in [75, 120], \zeta \in [0.0097, 0.01], \mathfrak{p} = 0.05, 0.10, 0.20, \tag{27}$$
$$\texttt{optimizer} \in \{\texttt{SGD}, \texttt{Adam}, \texttt{AdaDelta}, \texttt{AdaGrad}\}.$$

where *[a, b]* indicates the inclusive range of values from a to b. This BO search is executed in each of the fivefold cross-validations (CV), each repeated 3 times so that 15 distinct combinations are extracted. The best set of hyperparameters is selected with a minimum score of custom PM MSE from the initial random sampling list of 15 with iterations 10 (see Table 1 for BO). The `Python` code for implementing BO can be found in code list 3 in Appendix D.

Apart from hyperparameters, we also apply all regularization techniques in Section 2.5. For early stop, we consider both NLL and MSE metrics.

3. EMPIRICAL STUDIES

3.1. Telematics Data

The data set originated from cars driven in the United States where special UBI sensors were installed. The University of Haifa Actuarial Research Center provided the data, where UBI modeling is analyzed (Chan et al. 2022; Usman et al. 2024). The data set contains $J_0 = 65$ DVs constructed based on information collected from telematics and GPS for $N = 14157$ drivers. See the detailed exploratory data analysis in Section 3.3 of Usman et al. (2024) for the properties of these DVs and Appendix A for their description. These DVs are aggregated over time to obtain certain incidence rates and scaled to normalize their ranges for better interpretability of their coefficients in predictive models. These procedures transform the multidimensional longitudinal DVs into a single row for each driver, which is the unit of analysis.

The $J_0 = 65$ DVs are presented as column vectors $\boldsymbol{x}_{\bullet j}, j = 1, ..., J_0$. The data also contain two column vectors of claim counts $\boldsymbol{y} = y_{1:N}$ and policy duration or exposure $\boldsymbol{n} = n_{1:N}$ in years. Ninety-two percent of $\boldsymbol{y}$ are zero. Figure 2 displays three histograms for $\boldsymbol{y}$, $\boldsymbol{n}$, and annual number of claims $\boldsymbol{a} = \{a_i = y_i/n_i, i = 1, ..., N\}$, respectively. Their averages are $\bar{y} = 0.083, \bar{n} = 1.146$, and $\bar{a} = 0.075$, respectively. The variance of $y_{1:N}$ is 0.089, showing an equidispersion, possibly due to the large proportion of zeros. Drivers with 0, 1, and at least 2 claims form three classes $C_b = \{i : y_i = b\}$ with proportions $p_b = N_b/N$ being 0.92, 0.07, 0.005, averaged exposure $\bar{n}_b = \sum_{i\in C_b} n_i/N_b$ being 1.13, 1.38, 1.64, and averaged annual claim $\bar{a}_b = \sum_{i\in C_b} a_i/N_b$ being 0, 0.92, 1.71. Also, $\bar{y}_{2^+} = \sum_{i\in C_{2^+}} y_i/N_{2^+} = 2.11$. The number of claims is not simply linearly related to

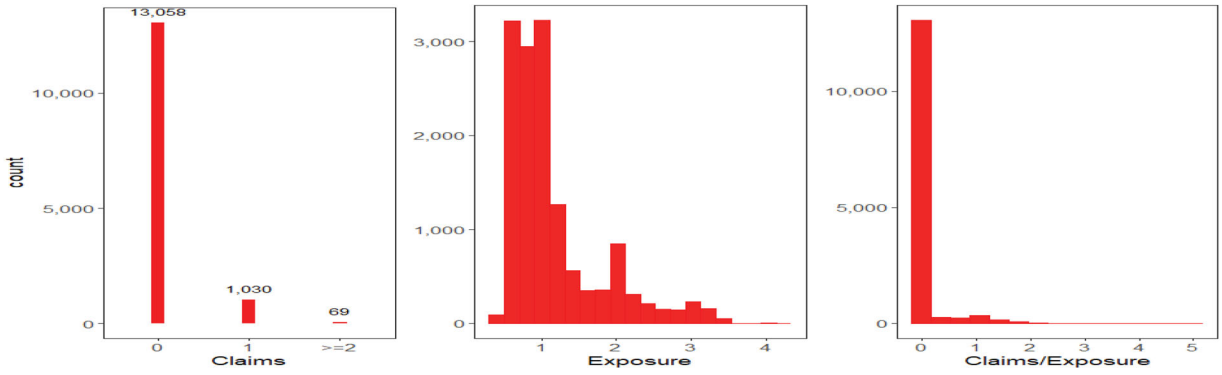


FIGURE 2. (a) Histogram of Number of Claims; (b) Exposure; and (c) Number of Claims per Exposure. *Source:* Usman et al. (2024).

exposure (R-squared is 0.014 regressing y_i on n_i), showing some possible impact of DVs $\boldsymbol{x}_{\bullet j}$ on y_i. Figures A.2 and A.3 of Usman et al. (2024) present the values, correlation matrix, and hierarchical clustering of the 65 DVs. Table A1 of Usman et al. (2024) further evaluates the information content of these DVs and selects $J = 45$ informative DVs in some analyses. This article adopts these $J = 45$ DVs and performs the analyses using PM DLNNs.

3.2. Constraints for Safe and Risky Groups

To implement the PM model, the NLL loss function defined in Equation (21) involves annual number of claims $\hat{a}_{i,1:2}(\boldsymbol{\theta})$ from both safe and risky groups. To ensure that the two groups are well defined, the predicted annual number of claims from the safe group should be restricted to be less than or at least mostly less than the those claims from the risky group, that is,

$$\hat{a}_{i1}(\boldsymbol{\theta}) \leq \hat{a}_{i2}(\boldsymbol{\theta}),$$

mostly. To secure such constraint, transformation functions $f_{cg}(\cdot)$ should be applied to the two output neurons, that is,

$$\hat{a}_{ig}(\boldsymbol{\theta}) = f_{cg}(a_{ig}^{(\mathfrak{Q}+1)}), \;\; g = 1, 2 \tag{28}$$

as in Equation (11). This section proposes the two transformation functions given by

$$\begin{aligned} \hat{a}_{i1} &= f_{c1}(a_{i1}) = 0.5\bar{a} \cdot f_{\texttt{sigmoid}}(\xi_1 a_{i1}) \in (0, 0.0375), \\ \hat{a}_{i2} &= f_{c2}(a_{i2}) = \min\big(0.5\bar{a} + 0.5\bar{a} \cdot f_{\texttt{softplus}}(\xi_2 a_{i2}), 5\big) \in (0.0375, 5) \end{aligned} \tag{29}$$

where the output neurons $a_{ig} := a_{ig}(\boldsymbol{\theta}) = a_{ig}^{(\mathfrak{Q}+1)}$, $f_{\texttt{signmoid}}(\cdot)$ is defined in Equation (19), $f_{\texttt{softplus}}(\cdot)$ is defined in Equation (18), $\bar{a} = 0.075$ as given in Section 3.1 is the average of the observed annual number of claims, and the hyperparameters ξ_1, ξ_2 are estimated to be 0.08 and 1, respectively, together with the network parameters $\boldsymbol{\theta}$. The ideas behind these specifications are given next.

For the first constraint, the function $f_{\texttt{sigmoid}}(\cdot)$ ensures that the transformed annual claim stays within the range (0, 1), and the scaling factor $0.5\bar{a}$ maps the outputs to the desired range (0,0.0375). The choice of taking $0.5\bar{a}$ as the threshold between safe and risky groups is arbitrary, but it is reasonable to set some higher threshold annual claim to define the risky group.

For the second constraint, the `softplus` function $f_{\texttt{softplus}}$ with a range $(0, \infty)$ ensures a positive predicted number of claims. Then the scaling of $0.5\bar{a}$ is applied, and lastly, shifting is conducted using $0.4\bar{a}$ or $0.5\bar{a}$ to achieve the desired output range (0.03, ∞) or (0.0375, ∞), respectively. The first range for the risky group overlaps with (0, 0.0375) for the safe group, whereas the second range does not overlap with (0, 0.0375). These two choices are assessed using MSE, and (0.0375, ∞) with $0.5\bar{a}$ shifting is chosen for $\hat{a}_{i2}$ to achieve equilibrium between $\hat{a}_{i,1:2}$ and maintain balance between conservatism and flexibility. Lastly, the range for the risky group is capped by 5 using `min(·, 5)` to avoid an excessively high predicted annual number of claims. The two hyperparameters ξ_1, ξ_2 tune the rate of increase for the sigmoid and soft plus functions to achieve a better approximation of the nonlinearities.

3.3. Estimation of Prior Probability π

Estimation of the previous probability π in Equation (2) is challenging in the DLNN, as it is not part of the network parameters $\boldsymbol{\theta}$; hence, its optimal value needs to be estimated separately from $\boldsymbol{\theta}$. There are three ways to estimate the optimal value of π.

3.3.1. Grid Search

A simple and intuitive method is to run the PM DLNN for a set of fixed $\pi \in \{0.88, 0.89, 0.90, \; 0.91, 0.92\}$ and an optimal π is chosen according to some objective functions such as NLL and MSE. The choice of π is based on the observed 0.92 proportion of zero claims for the telematics data and the belief that it is more likely to have risky zero-claim drivers than safe nonzero-claim drivers.

To ensure robust and reliable learning and balance the uncertainty due to fixing π, the proposed model undergoes refinement through 10 repeat runs, with each repeat comprising 2–3 trials to ensure that a stable model is obtained for each repeat. Each trial of each repeat uses a different seed. Instead of averaging the models across repeats, this strategy selects a model that optimizes an objective function such as MSE in Equation (21). The repeats facilitate a thorough exploration of the networks' intricacies, contribute to identifying optimal settings and configurations, and provide valuable insights to understand networks'

capabilities so that they ultimately lead to fine-tuning for optimal outcomes. The chosen best submodels should showcase the convergence path in the epoch history plot with a consistent and progressively descending trajectory to reflect a robust and reliable learning process. Moreover, this should have a low MSE so that the predicted annual number of claims in the scatter plot against observed align along the 45° diagonal line. The `Python` code list 4 in Appendix E is provided to guide the implementation of the DLNN with the PM model. Conditional on $\pi = 0.88$, say, the models using architectures from Manual and BO search are called MAN-0.88 and BO-0.88, respectively.

3.3.2. Dynamic Prior Probability

The idea of fixing the prior probability π in Section 3.3.1 presents some challenges in estimation because the current choice of $\pi \in \{0.88, 0.89, 0.90,\ 0.91, 0.92\}$ is not enough to provide a more accurate estimation, but increasing the points of the grid makes it more computationally demanding. One consideration is to allow a dynamic prior probability (DPP) by setting π_i to vary across drivers i in the network and assigning the *third neuron* in the output layer to estimate π_i. To ensure $\pi_i \in (0.88, 0.93)$, the third constraint for π_i is

$$0.88 + 0.05 \cdot f_{\texttt{sigmoid}}(\pi_i)$$

where $f_{\texttt{sigmoid}}(\cdot) \in (0, 1)$. This model is different from the models assuming a fixed π, but as π_i is restricted to a small range of (0.88,0.93), one would expect the results to be similar to those with π fixed within this range. Note also that this dynamic prior probability π_i is still different from the posterior probability $\eth_i$ in Equation (5), and $\eth_i$ can be calculated using Equation (5) by replacing π with π_i. The models are based on architectures from MAN search,

$$DPP - MAN : \mathfrak{Q} = 5, \quad \boldsymbol{m} = (45, 30, 15, 10, 5), \quad \mathfrak{p} = 0.20, \quad f_{1:4} = f_{\texttt{Linear}}, f_5 = f_{\texttt{ELU}}, \quad \mathcal{E} = 300, \tag{30}$$
$$\mathcal{B} = 120, \quad \zeta = 0.0097, \quad \mathcal{P} = 1, \quad \texttt{optimizer} = \texttt{AdaDelta},$$

and BO search with hyperparameter space

$$DPP - BO : \mathcal{E} \in [200, 350], \quad \mathcal{B} \in [90, 150], \quad \zeta \in [0.0095, 0.01], \quad \mathfrak{p} = 0.05, 0.10, 0.20, \tag{31}$$
$$\texttt{optimizer} \in \{\texttt{SGD}, \texttt{Adam}, \texttt{AdaDelta}, \texttt{AdaGrad}\}$$

and they are called DPP-MAN and DPP-BO, respectively. Since the dynamic π_i provides some flexibility in the network, no repeats or reruns are taken.

3.3.3. Iterative Conditional Optimization

Without compromising the nonregressive nature of π and rigorously estimating π and the DLNN parameter $\boldsymbol{\theta}$, we propose the iterative conditional optimization (ICO) estimation process, which can be conducted in two iterative steps: one for updating the DLNN parameter $\boldsymbol{\theta}$ given the current π, and another for updating π given the current DLNN parameters $\boldsymbol{\theta}$. In the training of π, the gradient of π with respect to the overall loss function in Equation (21) suggests adjusting an update value of π to minimize the overall loss, considering the interplay between model predictions and the custom loss function. This process is repeated until the algorithm converges. The whole procedures require an inner epoch of size 50 and an outer epoch of size 100. As the estimation of π changes along the outer epoch, allowing some flexibility, no repeats or reruns are performed. The choice of architectures is based on the BO search.

4. EXPERIMENTAL RESULTS

Proposed PM DLNNs are applied to fit the telematics data with a train–test random split. The training data contain $\mathsf{p} = 80\%$ of $N = 14,157$ observations. The architectures of the manual search are provided in Equation (26), whereas those from the BO search are obtained in Equation (27). The parameter π of these DLNNs is estimated using the methodologies in Section 3.3.3. To compare the performance of different models, we calculated several metrics. Specifically, we evaluated MSE training and validation in Equation (20), and NLL in Equation (21). The results of these evaluations are summarized in Table 1.

4.1. Grid Search

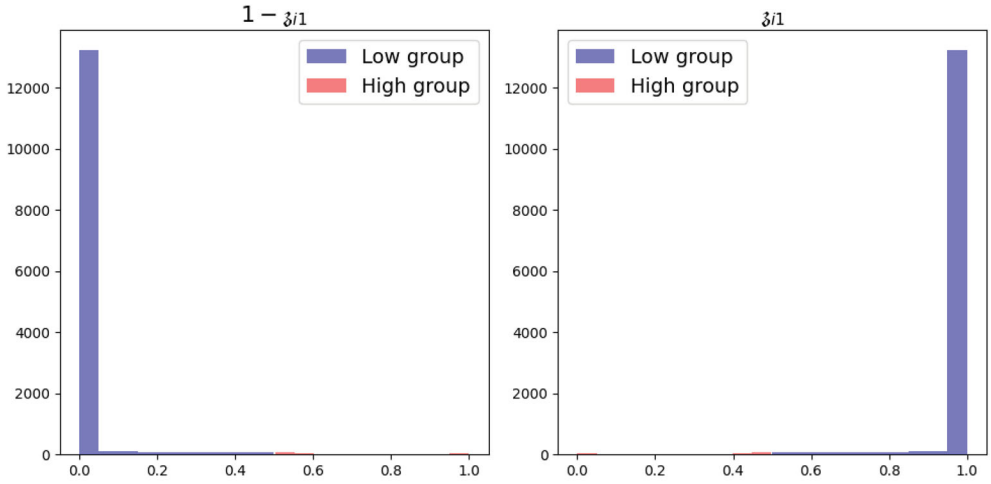
Results of the optimal submodel from 10 repeats of each π using the MAN and BO search are reported in Table 1. The best models for each architecture search method, namely, MAN-0.89 and BO-0.88, are selected based on prediction accuracy using PM MSE in Equation (20). This MSE compares model accuracy in terms of the observed and predicted number of claims using the PM model. The MSE, as well as NLL, is cross-classified by training and validation sets. The convergence and stability of the two best models are checked from the epoch history of NLL, showing slow downward trends consistent for both training and validation data.

To access the classification performance, the posterior probabilities $\mathfrak{z}_{i1}$ in Equation (5) and $1 - \mathfrak{z}_{i1}$ are plotted in Figure 3 for the three best models. The distributions of $\mathfrak{z}_{i1}$ and $1 - \mathfrak{z}_{i1}$ indicate a clear classification into the safe group for the majority of drivers, as most of the $\mathfrak{z}_{i1} \simeq 1$ are expected since 92% of drivers have zero claims. However, a small portion of drivers with $\mathfrak{z}_{i1} \simeq 0.5$ show some uncertainties in classifying safe drivers. Driver classification can be obtained by classifying driver i to the safe group $\mathcal{G}_1$ if $\mathfrak{z}_{i1} > 0.5$; otherwise, he/she is classified to the risky group $\mathcal{G}_2$. Results show that there is a small portion of drivers with $\mathfrak{z}_{i1} > 0.5$ (3.40% for model MAN-0.89 and 2.67% for model BO-0.88), and they are classified as risky drivers.

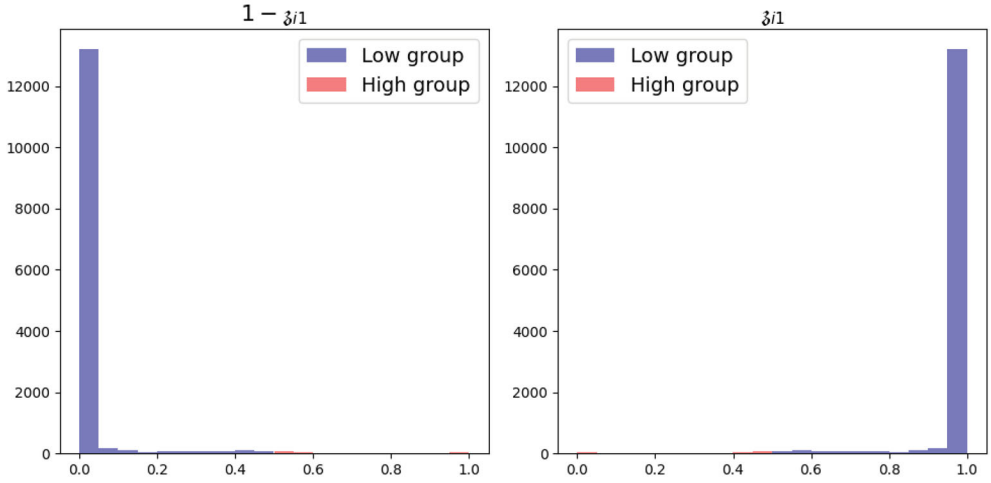
Lastly, the prediction accuracy overall and by the driver groups can be visualized in the scatter plots of predicted marginal posterior annual number of claims $\hat{a}_i(\boldsymbol{\theta})$ (weighted by $\mathfrak{z}_{ig}$ between $\hat{a}_{i1}$ and $\hat{a}_{i2}$ in Equation (7)) against the observed annual number of claims graphed in Figure 4. Both best-selected models exhibit desirable agreement of the predicted annual number of claims, as the majority of blue points with one claim align closely to the diagonal line without any discernible patterns. Moreover, the small red dot indicates that the majority of zero-claim drivers are predicted to have close to zero annual number of claims. These drivers are classified mostly to the safe group. Moreover, the right plots indicate that nearly all drivers with at least two claims are risky drivers, as expected, and they are mostly predicted to have higher than observed annual number of claims (above the diagonal line). In summary, the amount of deviation of the points from the diagonal line is reflected in the PM MSE measure.

TABLE 1.
Model Performance for DLNN and REG Models with Architectures Using MAN Search in Equation (26) and BO Search in Equation (27). Each Grid Search Model is Repeated 10 Times. The Best Model is Selected Based on the MSE in Equation (20)

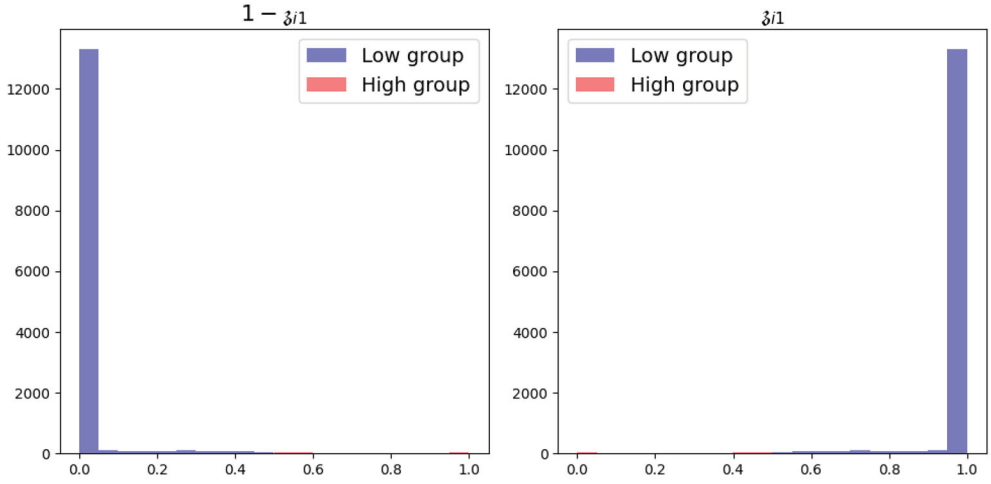
						NLL		MSE	
π	$\mathcal{B}$	$\mathcal{E}$	ζ	Optimizer	$\mathfrak{p}$	Training	Validation	Training	Validation
Grid search method using architecture from Manual (MAN) search									
0.88	110	400	0.0100	`AdaDelta`	0.20	0.3883	0.4124	0.1144	0.1379
0.89	110	400	0.0097	`AdaDelta`	0.20	0.4001	0.3828	0.1212	**0.1087**
0.90	110	400	0.0097	`AdaDelta`	0.20	0.4091	0.3914	0.1222	0.1097
0.91	110	400	0.0097	`AdaDelta`	0.20	0.4114	0.4066	0.1191	0.1203
0.92	110	400	0.0097	`AdaDelta`	0.20	0.4189	0.4139	0.1194	0.1207
0.93	110	400	0.0100	`AdaDelta`	0.20	0.4256	0.4206	0.1195	0.1208
Grid search method using architecture from Bayesian optimization (BO) search									
0.88	93	267	0.0097	`SGD`	0.10	0.3915	0.3999	0.1190	**0.1183**
0.89	93	267	0.0097	`SGD`	0.20	0.3972	0.4057	0.1193	0.1186
0.90	107	337	0.0098	`SGD`	0.10	0.3979	0.4214	0.1141	0.1352
0.91	120	396	0.0099	`SGD`	0.20	0.4067	0.4315	0.1153	0.1382
0.92	99	291	0.0098	`SGD`	0.20	0.4132	0.4386	0.1154	0.1368
0.93	107	337	0.0098	`SGD`	0.20	0.4209	0.4465	0.1158	0.1371
Dynamic prior probability (DPP) using architecture from MAN (row 1) and BO (row 2) search									
0.93	120	300	0.0097	`AdaDelta`	0.20	0.4070	0.4156	0.1186	**0.1174**
0.93	97	232	0.0096	`SGD`	0.20	0.4206	0.3798	0.1192	0.1242
Iterative conditional optimization using architecture from BO search with estimation of π									
0.80	93	171	0.0097	`SGD`	0.20	0.2920	0.3195	0.0879	**0.0872**



(a) **Model MAN-0.88** $\mathcal{E} = 400$; $\mathcal{B} = 110$; $\zeta = 0.0100$

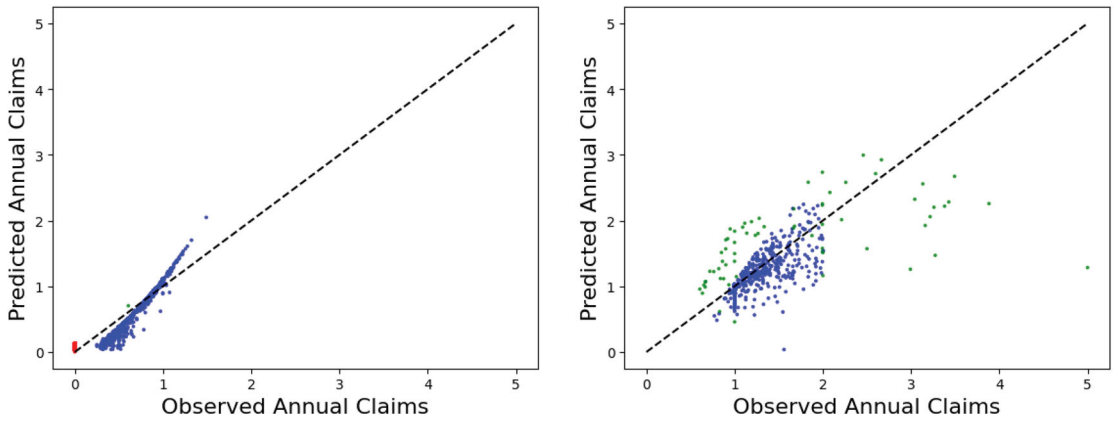


(b) **Model BO-0.89** $\mathcal{E} = 267$; $\mathcal{B} = 93$; $\zeta = 0.0097$



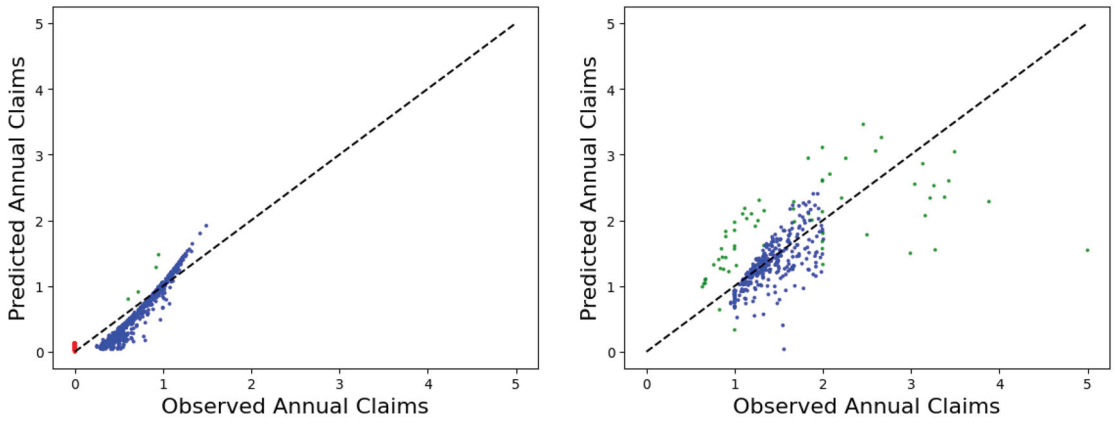
(c) **Model DP-MAN** $\mathcal{E} = 300$; $\mathcal{B} = 120$; $\zeta = 0.0097$

FIGURE 3. Distribution of Posterior Probabilities $\hat{z}_{i1}$ and $1 - \hat{z}_{i1}$ for Model MAN-0.89 (Row 1), Model BO-0.88 (Row 2), and Model DPP-MAN (Row 3).



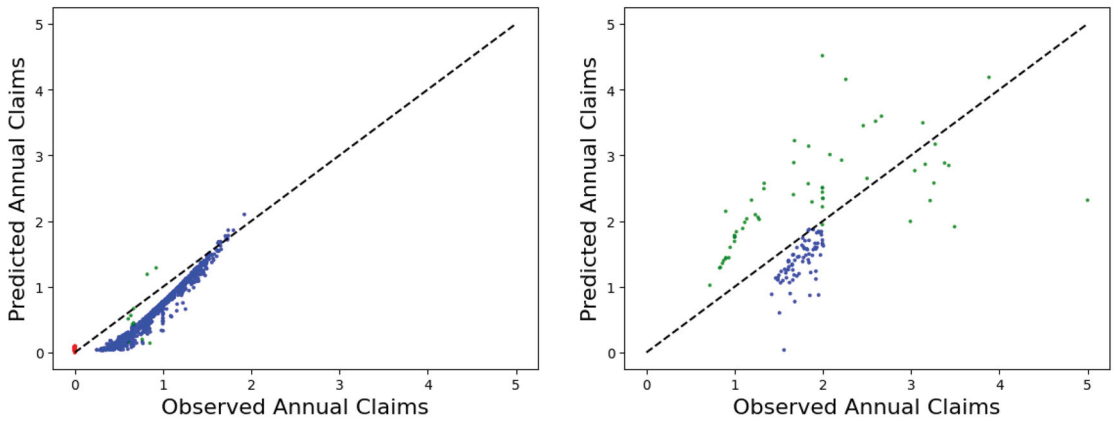
(a) Low Group (b) High Group

Model MAN-0.88 $\mathcal{E} = 400$; $\mathcal{B} = 110$; $\zeta = 0.01$; PM MSE $= 0.0129$



(c) Low Group (d) High Group

Model BO-0.89 $\mathcal{E} = 267$; $\mathcal{B} = 93$; $\zeta = 0.0097$; PM MSE $= 0.0123$



(e) Low Group (f) High Group

Model DPP-MAN $\mathcal{E} = 300$; $\mathcal{B} = 120$; $\zeta = 0.0097$; PM MSE $= 0.0140$

FIGURE 4. Predicted Marginal Posterior Annual Number of Claims against Observed Annual Number of Claims for the Best Models, MAN-0.89 (Row 1), BO-0.88 (Row 2), and DPP-MAN (Row 3) for Low (Left), and High (Right) Claim Groups. Red, Blue, and Green Dots Denote 0, 1, and $\geq$2 Observed Number of Claims, Respectively.

4.2. Dynamic Prior Probability

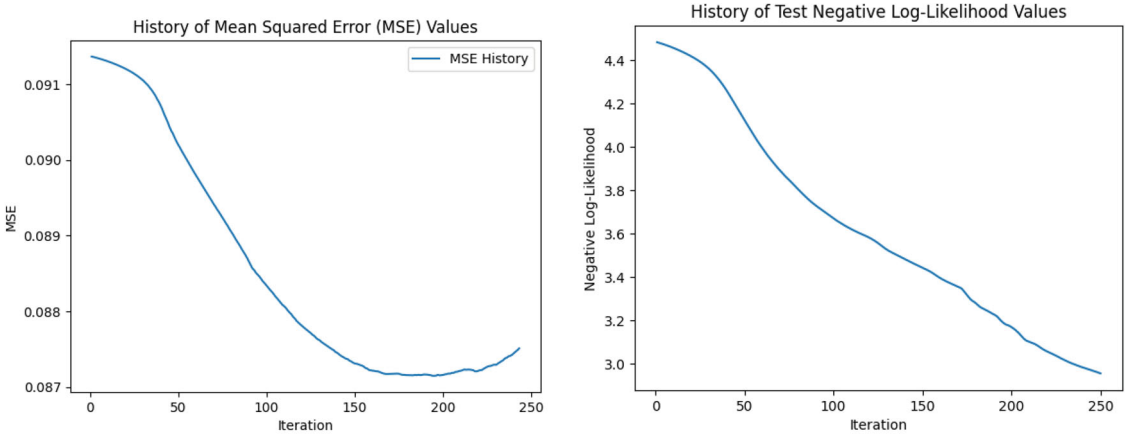
Similarly, Table 1 and Figures 3 and 4 also report results using the DPP method. The proportion of drivers classified into risk groups is 0.93% for the DPP-MAN model and 1.08% for the DPP-BO model. Between using MAN and BO architecture search, the DPP-MAN model is chosen to be the best model according to PM MSE. For this model, the average of π_i overall drivers is $\bar{\pi} = 0.93$, and it has a slightly higher PM MSE than the MAN-0.89 model. Figure 4 shows that DPP-MAN slightly underestimates the annual number of claims for the safe group and slightly overestimates the annual number of claims for the risky group. Results using the DPP method as reported in Table 1 and Figures 3 and 4 are similar to those of the grid search method.

4.3. Iterative Conditional Optimization

This method allows the prior probability to be estimated as a nonregression parameter through iterations, which is different from the dynamic prior probability method. However, experience from setting up early stop through searching for $\mathcal{P}$ shows that prediction performance is sensitive to epochs. Hence, we choose MSE, which measures prediction accuracy, as a criterion for early stopping to avoid overfitting. Figure 5(a) and (b) present the MSE and NLL values across epochs. The validation MSE of 0.0872 at epoch 170 is the lowest MSE across epochs and is also the lowest MSE across all models in Table 1, indicating that it is the best model. We note that NLL in Figure 5(a) does not converge, a common occurrence in neural network training that highlights an alternative notion of convergence often linked to overfitting. This selected ICO model has the lowest validation MSE and NLL.

Results from Table 1 show that the prior probability π is estimated to be 0.8, which is lower than other π estimates. When comparing Figures 6 and 7 with Figure 4, the driver classification based on the posterior probability $\hat{\gamma}_{i1}$ defined in Equation (5) is approximately similar. The 45-degree dashed line indicates agreement between the observed and predicted annual number of claims defined in Equation (7). We remark that agreement (measured by MSE) may not always indicate good model performance, as we also account for the effect of driving behavior, which may impact the predicted annual number of claims as expected.

In Figure 6, the majority of the annual numbers of claims are estimated to be lower than the observed, but there are small groups of high-risk drivers whose annual number of claims are estimated to be much higher. The distribution between observed and predicted annual numbers of claims is similar between the training and test sets. Figure 7 allows us to identify a few distinct driver groups. The first group contains zero-claim drivers who are estimated to have near zero annual number of claims. Hence, they are safe drivers. This group is represented by small red dots in the left-hand plots of Figure 7. The agreement shows that they do not exhibit risky driving, which may increase the predicted annual claim frequency. The second group is again safe drivers, who are estimated to have near-zero annual claim frequency, despite having some nonzero claims (horizontal blue points in Figure 7). This group shows the ability of the ICO-BO model to identify a group of safe drivers using DVs from the remaining risky drivers. Then the insurance company is able to allow them lower premiums because of their safe driving despite having nonzero claims. The third group consists mostly of risky drivers, whose predicted annual number



(a) Validation MSE History (b) Negative Log-Likelihood History

FIGURE 5. (a) MSE and (b) NLL Values of the Selected ICO Model for the Validation Sample (without Resampling) across Epochs.

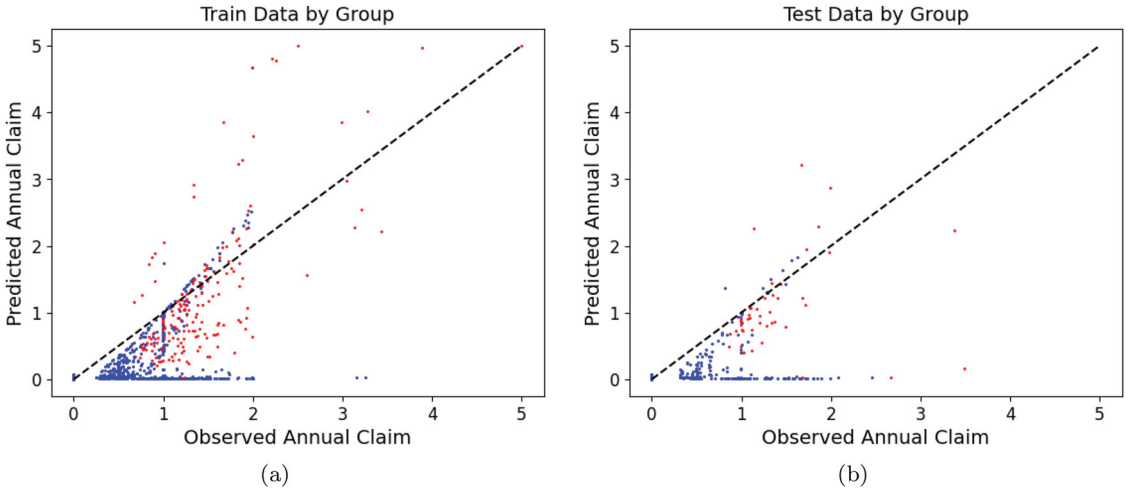


FIGURE 6. Predicted Marginal Posterior Annual Number of Claims against Observed Annual Number of Claims by Risk Group for the Best ICO-BO Model for (a) the Train Data (Left) and (b) the Test Data (Right). Blue Indicates the Low-Risk Group, Whereas Red Denotes the High-Risk Group.

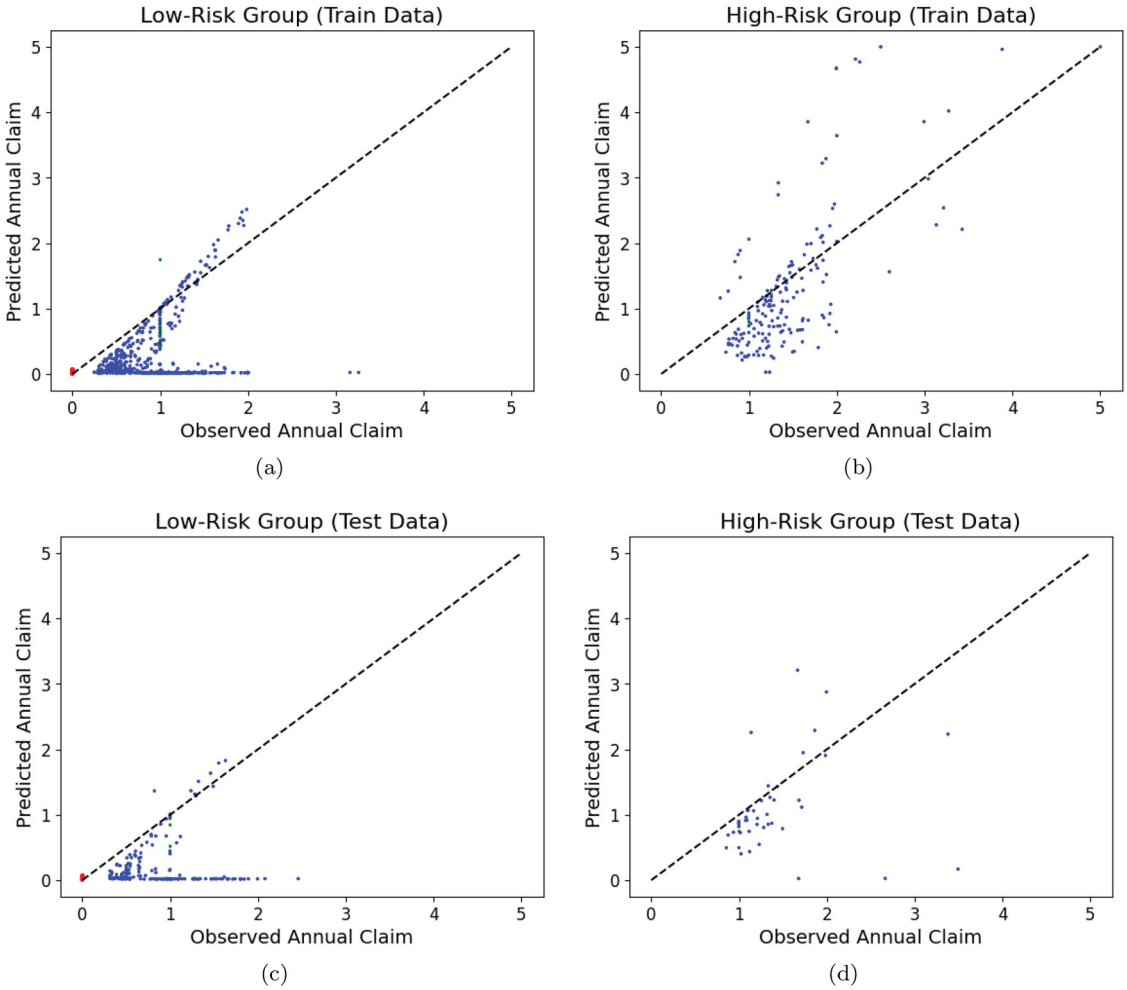


FIGURE 7. Predicted Marginal Posterior Annual Number of Claims against Observed Annual Number of Claims by a Number of Claims for the Best ICO-BO Model by Train Data (Row 1) and Test Data (Row 2) and by Low (Left) and High (Right) Claim Groups. Red, Blue, and Green Dots Denote 0, 1, and $\geq$2 Observed Number of Claims, Respectively.

of claims cluster around the 45-degree line. Consequently, the observed and predicted annual numbers of claims are linearly related, and those lying higher above the 45-degree line are more likely to be predicted as risky drivers.

Results from the predicted number of claims and the driver group present opportunities to provide targeted alerts or relevant driver education to mitigate future risks. Results are also useful for calculating the UBI experience-rating premium, as discussed in Usman et al. (2024). Python codes are given in Appendix section E.

5. CONCLUSION

This article proposes a PM DLNN to improve the accuracy of predicting drivers' number of claims and group classifications. The contribution of this research is multifaceted, addressing fundamental challenges, and pushing the boundaries of modeling precision. Several factors significantly impact the performance of DLNN models, including the network structures that influence the ability to uncover complex data patterns, the learning rates that affect processing time, and the estimators of nonregression prior parameters and optimizers of network parameters, which impact training efficiency and modeling stability. We discuss in detail how we develop, compile, and train the models, addressing these issues. We also highlight the innovative use of the PM NLL loss function, PM MSE prediction accuracy metric, and custom constraints on output neurons to differentiate between driver groups. One of the key advantages of our proposed model is its ability to identify and appropriately price different risk profiles. The aims and contributions of this article are discussed in detail next.

First, the research begins with a careful search of architectures based on experience, guidelines from literature, and efficient BO search techniques. Architecture search involves fine-tuning parameters such as epochs, batch size, learning rates, and optimizers, as well as regularization techniques like batch normalization, early stopping, and dropout, to enhance convergence and balance the trade-off between underfitting and overfitting. The deliberate and strategic use of manual and BO searches further enhances the model's effectiveness, balancing model efficiency with computational resources. The manual search is an initial exploration, identifying optimal hyperparameter combinations and providing insights into the data's intricacies and model complexity. The manual search also aids in deciphering these nuances, enabling researchers to establish appropriate ranges of hyperparameters for subsequent more advanced and efficient BO.

Second, introducing the designated PM NLL with two mean parameters presents difficulties in model implementation. Identifiability issues are common in mixture models. Since the labeling of components in a mixture model is arbitrary, swapping two components produces the same likelihood, making the interpretation of labels difficult (Stephens 2000; Yao 2012). Moreover, the two components should be adequately separated; otherwise, the effectiveness of the classification will be reduced. To ensure that the two driver groups are well separated, a strategic effort is to incorporate constraints for the ranges of predicted annual number of claims with $a_{i1} \in (0, 0.0375)$ and $a_{i2} \in (0.0375, 5)$. Figures 4 and 7 show a reasonable separation of the two driver groups.

Lastly, the simultaneous estimation of network parameters $\boldsymbol{\theta}$ and nonnetwork prior probability parameter π causes challenges. Three methods, grid search, DPP, and ICO, are proposed to ensure that the networks align with the underlying dynamics of the data, emphasizing precision in reflecting real-world scenarios. The common grid search strategy is first employed using six π values, and a PM DLNN is run with 10 repeats for each π. The 10 repeats provide refinement to the optimal PM DLNN, enhancing robustness and adaptability. Two PM DLNNs are selected according to PM MSE for the architectures using manual and BO search, respectively. The second DPP method compromises the fixed prior probability in Equation (2) with a DPP that treats π as a regression/network parameter, assigning it to the third output neuron and allowing it to vary within a narrow range of (0.88, 0.93). Despite the flexibility in estimation, it shares the same restriction as the grid research method, namely, that the range of π is constrained in its implementation.

Drawing from the insights of the search method, we design the third ICO method such that π is updated based on the gradients computed from the loss function condition on the current state of network parameters. In other words, this novel method demonstrates an intertwined optimization process where both network parameters $\boldsymbol{\theta}$ and prior probability π are iteratively updated, estimating one condition on the others and vice versa, iteratively until convergence or early stop criteria are met. The predicted number of claims and driver groups can be applied to various scenarios, including personalized insurance pricing, such as the UBI experience-rating method, and risk classification.

Furthermore, insurance portfolios are dynamic and continuously evolve due to the introduction of new policy types, adjustments in coverage options, and changes in customer demographics. To address this complexity, the proposed method enables personalized pricing by using real-time data on driving behavior from individual customers. This allows insurers to move beyond traditional risk pools and adopt a more data-driven, individualized pricing strategy. In addition, the model can be regularly and more frequently updated to reflect structural shifts due to external and environmental changes, leading to changes in driving behavior. Moreover, the threshold modeling framework introduced by Dong and Chan (2013) offers a flexible approach to incorporate new legislative changes and policy rules as they arise. This threshold model framework captures policy changes by allowing the model to reestimate parameters after a time-based threshold, enhancing adaptability and extending the applicability of the current model. Overall, NNs bring enhanced accuracy, efficiency, and automation to the insurance claim prediction processes. By leveraging their capabilities, insurers can make more informed decisions, minimize risks, detect fraud, and improve customer experiences. The threshold model also casts a promising future research direction.

Apart from threshold models, another intriguing avenue for exploration involves extending the number of driver groups G for the PM model beyond two, capturing other distinct driving behaviors. Previous studies in clinical and criminology (Chan et al. 1998; Brame et al. 2006) demonstrated the applicability and superior performance of mixture models with three or more components. This extension to telematics data could closely examine different driving styles using both LASSO regressions and DLNNs. For the case with $G = 3$, the model can potentially identify a group with excessive or structural zero claims, another group with a low number of claims, and lastly, a group with a high number of claims, combining the zero-inflated Poisson model with PM model to capture the heterogeneity in the data and enhance prediction accuracy. Hence, the extended multicomponent mixture models remain a promising avenue, although the identification and interpretation of these groups pose additional challenges. These challenges stem from setting up proper constraints to differentiate driver groups, thereby avoiding label switching (Stephens 2000; Yao 2012) in some iterative estimation procedures such as MCMC and SGD. The lack of prior exploration in this direction makes it promising to derive unprecedented insights into the dynamics of driving behavior from telematics data.

ACKNOWLEDGEMENT

The authors thank the editor and anonymous reviewers for their careful review and insightful comments that strengthened the article.

DISCLOSURE STATEMENT

No potential conflict of interest was reported by the author(s).

ORCID

Farha Usman http://orcid.org/0000-0002-6540-3172
Jennifer S. K. Chan http://orcid.org/0000-0002-3167-6586
Alice X. D. Dong http://orcid.org/0009-0001-7774-0440

REFERENCES

Amini A, Schwarting W, Soleimany A, Rus D. 2020. Deep evidential regression. In: Advances in Neural Information Processing Systems, Vol. 33. p. 14927–14937.

Bengio Y. 2012. Practical recommendations for gradient-based training of deep architectures. In: Neural networks: tricks of the trade. 2nd ed. p. 437–478. arXiv. https://arxiv.org/abs/1206.5533

Berry MJ, Linoff GS. 2004. Data mining techniques: for marketing, sales, and customer relationship management. John Wiley & Sons.

Boger Z, Guterman H. 1997. Knowledge extraction from artificial neural network models. In: 1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Vol. 4. p. 3030–3035.

Brame R, Nagin DS, Wasserman L. 2006. Exploring some analytical characteristics of finite mixture models. J Quant Criminol. 22(1):31–59. https://doi.org/10.1007/s10940-005-9001-8

Buyrukoğlu G, Buyrukoğlu S, Topalcengiz Z. 2021. Comparing regression models with count data to artificial neural network and ensemble models for prediction of generic *Escherichia coli* population in agricultural ponds based on weather station measurements. Microb Risk Anal. 19:100171. https://doi.org/10.1016/j.mran.2021.100171

Chan JS, Boris Choy ST, Makov U, Shamir A, Shapovalov V. 2022. Variable selection algorithm for a mixture of Poisson regression for handling overdispersion in claims frequency modeling using telematics car driving data. Risks. 10(4):83. https://doi.org/10.3390/risks10040083

Chan JSK, Kuk AYC, Bell J, Mc Gilchrist C. 1998. The analysis of methadone clinic data using marginal and conditional logistic models with mixture or random effects. Aus NZ J Stat. 40(1):1–10. https://api.semanticscholar.org/CorpusID:35954752. https://doi.org/10.1111/1467-842X.00001

Dong AXD, Chan JSK. 2013. Bayesian analysis of loss reserving using dynamic models with generalized beta distribution. Insur Math Econ. 53(2):355–365. https://doi.org/10.1016/j.insmatheco.2013.07.001

Dong W et al. 2016. Characterizing driving styles with deep learning. arXiv preprint arXiv:1607.03611.

Drori I. 2022. The science of deep learning. Cambridge University Press.

Fallah N et al. 2009. Nonlinear Poisson regression using neural networks: a simulation study. Neural Comput Appl. 18(8):939–943. https://doi.org/10.1007/s00521-009-0277-8

Gao G, Wüthrich MV. 2018. Feature extraction from telematics car driving heatmaps. Eur Actuar J. 8(2):383–406. https://doi.org/10.1007/s13385-018-0181-7

Gao G, Wüthrich MV, Yang H. 2019. Evaluation of driving risk at different speeds. Insur Math Econ. 88:108–119. https://doi.org/10.1016/j.insmatheco.2019.06.004

Garbin C, Zhu X, Marques O. 2020. Dropout vs. batch normalization: an empirical study of their impact to deep learning. Multimed Tools Appl. 79(19-20):12777–12815. https://doi.org/10.1007/s11042-019-08453-9

Goodfellow I, Bengio Y, Courville A. 2016. Deep learning. MIT Press.

Guo J, Liu Y, Zhang L, Wang Y. 2018. Driving behaviour style study with a hybrid deep learning framework based on GPS data. Sustainability. 10(7):2351. https://doi.org/10.3390/su10072351

Hanafy M, Ming R. 2021. Machine learning approaches for auto insurance big data. Risks. 9(2):42. https://doi.org/10.3390/risks9020042

Ioffe S, Szegedy C. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning. PMLR. p. 448–456.

Kandasamy K, Neiswanger W, Schneider J, Poczos B, Xing EP. 2018. Neural architecture search with bayesian optimisation and optimal transport. In: Advances in Neural Information Processing Systems. p. 31.

Kang SY. 1991. An investigation of the use of feedforward neural networks for forecasting [PhD dissertation]. http://ezproxy.library.usyd.edu.au/login?url=https://www.proquest.com/dissertations-theses/investigation-use-feedforward-neural-networks/docview/303933938/se-2

Karsoliya S. 2012. Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture. International Journal of Engineering Trends and Technology. 3:714–717.

Kim J-M, Kim J-M, Ha I. 2022. Application of deep learning and neural network to speeding ticket and insurance claim count data. Axioms. 11(6):280. https://doi.org/10.3390/axioms11060280

Klein A, Falkner S, Bartels S, Hennig P, Hutter F. 2017. Fast bayesian optimization of machine learning hyperparameters on large datasets. In: Artificial intelligence and statistics. PMLR. p. 528–536.

LeCun Y, Bengio Y, Hinton G. 2015. Deep learning. Nature. 521(7553):436–444. https://doi.org/10.1038/nature14539

Li M, Soltanolkotabi M, Oymak S. 2020. Gradient descent with early stopping is provably robust to label noise for overparameterized neural networks. In: International Conference on Artificial Intelligence and Statistics. PMLR. p. 4313–4324.

Liu Y, Starzyk JA, Zhu Z. 2007. Optimizing number of hidden neurons in neural networks. EeC. 1(1):6.

Makov U, Weiss J. 2016. Predictive modeling for usage-based auto insurance. In: Frees EW, Meyers G, Derrig RA, editors. Predictive modeling applications in actuarial science. Vol. 2. Cambridge University Press. p. 290–308.

Masters D, Luschi C. 2018. Revisiting small batch training for deep neural networks. arXiv Preprint. arXiv:1804.07612.

Masum M et al. 2021. Bayesian hyperparameter optimization for deep neural network-based network intrusion detection. In: 2021 IEEE International Conference on Big Data (Big Data). IEEE. p. 5413–5419.

Mishra S, Yamasaki T, Imaizumi H. 2019. Improving image classifiers for small datasets by learning rate adaptations. In: 2019 16th International Conference on Machine Vision Applications (MVA). IEEE. p. 1–6.

Moon TK. 1996. The expectation-maximization algorithm. IEEE Signal Process Mag. 13(6):47–60. https://doi.org/10.1109/79.543975

Paefgen J, Staake T, Thiesse F. 2013. Evaluation and aggregation of pay-as-you-drive insurance rate factors: a classification analysis approach. Decis Support Syst. 56:192–201. https://doi.org/10.1016/j.dss.2013.06.001

Prechelt L. 2002. Early stopping-but when?. In: Neural networks: tricks of the trade. Springer. p. 55–69.

Qu Z, Yuan S, Chi R, Chang L, Zhao L. 2019. Genetic optimization method of pantograph and catenary comprehensive monitor status prediction model based on Adadelta deep neural network. IEEE Access. 7:23210–23221. https://doi.org/10.1109/ACCESS.2019.2899074

Sakthivel KM, Rajitha CS. 2017. A comparative study of zero-inflated, hurdle models with artificial neural network in claim count modeling. Int J Stat Syst. 12(2):265–276.

Santurkar S, Tsipras D, Ilyas A, Madry A. 2019. How does batch normalization help optimization?. In: Advances in Neural Information Processing Systems. arXiv. p. 31. https://arxiv.org/abs/1805.11604

Shahriari B et al. 2016. Taking the human out of the loop: a review of Bayesian optimization. Proc IEEE. 104(1):148–175. https://doi.org/10.1109/JPROC.2015.2494218

Sharma S, Sharma S, Athaiya A. 2017. Activation functions in neural networks. Towards Data Sci. 6(12):310–316.

Simoncini M et al. 2018. Vehicle classification from low-frequency GPS data with recurrent neural networks. Transp Res Part C Emerg Technol. 91:176–191. https://doi.org/10.1016/j.trc.2018.03.024

Smith KA, Willis RJ, Brooks M. 2000. An analysis of customer retention and insurance claim patterns using data mining: a case study. J Oper Res Soc. 51(5):532–541. https://doi.org/10.1057/palgrave.jors.2600941
Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. 2014. Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res. 15(1):1929–1958.
Stephens M. 2000. Dealing with label switching in mixture models. J R Stat Soc Ser B (Stat Methodol). 62(4):795–809. https://doi.org/10.1111/1467-9868.00265
Sze V, Chen Y-H, Yang T-J, Emer JS. 2017. Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE. 105(12):2295–2329. https://doi.org/10.1109/JPROC.2017.2761740
Tang Z, Fishwick P. 1993. Feedforward neural nets as models for time series forecasting. INFORMS J Comput. 5(4):374–385. https://doi.org/10.1287/ijoc.5.4.374
Usman F, Chan J, Makov U, Wang Y, Dong AX. 2024. Claim prediction and premium pricing for telematics auto insurance data using Poisson regression with Lasso regularisation. Risks. 12(9):137. https://doi.org/10.3390/risks12090137
Weerasinghe K, Wijegunasekara MC. 2016. A comparative study of data mining algorithms in the prediction of auto insurance claims. Eur Int J Sci Technol. 5(1):47–54.
Williams AR, Jin Y, Duer A, Alhani T, Ghassemi M. 2022. Nightly automobile claims prediction from telematics-derived features: a multilevel approach. Risks. 10(6):118. https://doi.org/10.3390/risks10060118
Wong SY, Chan JS, Azizi L. 2025. Quantifying neural network uncertainty under volatility clustering. Neurocomputing. 614:128816. https://doi.org/10.1016/j.neucom.2024.128816
Wüthrich MV, Merz M. 2019. EDITORIAL: YES, WE CANN!. ASTIN Bull. 49(1):1–3. https://doi.org/10.1017/asb.2018.42
Yao W. 2012. Model based labeling for mixture models. Stat Comput. 22(2):337–347. https://doi.org/10.1007/s11222-010-9226-8
Ying X. 2019. An overview of overfitting and its solutions. Int J Phys Conf Ser. 1168:022022. https://doi.org/10.1088/1742-6596/1168/2/022022
You K, Long M, Wang J, Jordan MI. 2019. How does learning rate decay help modern neural networks?. https://arxiv.org/abs/1908.01878
Yunos Z, A. Ali Siti Mariyam Shamsuddin Ismail Noriszura, Roselina Sallehuddin 2016. Predictive modelling for motor insurance claims using artificial neural networks. Int J Adv Soft Comput Appl. 8.
Zeiler MD. 2012. Adadelta: an adaptive learning rate method. arXiv:1212.5701.
Zhang Z. 2018. Improved Adam optimizer for deep neural networks. In: 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS). IEEE. p. 1–2.

APPENDIX A. COMPUTATION OF THE NUMBER OF PARAMETERS

TABLE A1.
Computation of the Number of Parameters M for PM-DLNN with $m_0 = 45$ Features, $\Omega = 5$ Hidden Layers, $\boldsymbol{m} = (45, 30, 15, 10, 5)$ Neurons in the Hidden Layers, and $m_f = 2$ Neuron in the Output Layer

Layer l	No. of neuron m_l	No. of parameters	Total
Input	$m_0 = 45$	0	0
1	$m_1 = m_0 = 45$	$45 \times 45 + 45$	2070
`BatchN`	45	4×45	180
2	$m_2 = 2m_1/3 = 30$	$45 \times 30 + 30$	1380
3	$m_3 = m_2/2 = 15$	$30 \times 15 + 15$	465
`DropOut` (0.20)	0	0	0
4	$m_4 = 2m_3/3 = 10$	$15 \times 10 + 10$	160
5	$m_5 = m_4/2 = 5$	$10 \times 5 + 5$	55
Output	$m_f = 2$	$5 \times 2 + 2$	12
Overall total M			4322

APPENDIX B. DEFINE PM NLL FUNCTION WITH FIXED π

To implement the PM DLNNs, the NLL function in Equation (21) with $\mu_{ig} = n_i \hat{a}_{ig}(\boldsymbol{\theta}) = n_i a_{ig}^{\varrho+1}$ called `mixture_poisson_loss` is defined in `Python`. The breakdown of each code is explained here:

- The true values of `y` are `y_true` and the predicted values are `y_pred`.
- a_{i1} (`a1 = a_pred[:, 0]`) and a_{i2} (`a2 = a_pred[:, 1]`) extract the predicted outputs for the low and high groups. These values are used in the calculation of NLL.
- `n=y_true[:, -1]` extracts the exposure column values (n_i) from the y array and `y=y_true [:, :-1]` extracts the claim (y_i) values by first removing the last columns (n_i) from the y array, leaving only the claim counts.
- Constraints are imposed on two predicted outputs a_{i1} and a_{i2} as in Equation (29).

LISTING 1: Customise PM NLL in `Python`

```python
# Define the custom loss function
def mixture_poisson_loss_with_exposure(y_true, y_pred):
    a_pred1 = a_pred[:, 0]  # Extracting the predicted output a1
    a_pred2 = a_pred[:, 1]  # Extracting the predicted output a2

    pi = 0.92  # Define the pi values
    a_bar = 0.075  # annual claim 0.0751375
    n = y_true[:, -1]
    y = y_true[:, :-1]
    # Small epsilon value to avoid zero or negative values in logarithm
    epsilon = 1e-10

    # Apply constraint: a1<a2, a1(0,0.0375), a2(0.0375,5), xi1,xi2 to be est
    a1 = 0.0375 * tf.math.exp(a_pred1*xi1) / (1 + tf.math.exp(a_pred1*xi1))
    a2 = tf.minimum(0.0375 + 0.0375 * tf.math.softplus(a_pred2*xi2), 5)

    mu1 = tf.multiply(a1, n)
    mu2 = tf.multiply(a2, n)

    p1 = tf.divide(
        tf.multiply(tf.math.pow(mu1, y), tf.exp(-mu1)),
        tf.exp(tf.math.lgamma(y + 1))
    )
    p2 = tf.divide(
        tf.multiply(tf.math.pow(mu2, y), tf.exp(-mu2)),
        tf.exp(tf.math.lgamma(y + 1))
    )

    loss0 = -tf.math.log(tf.multiply(pi, p1) + tf.multiply((1 - pi), p2))
    loss = tf.reduce_mean(loss0)

    return loss
```

APPENDIX C. DEFINE PM MSE FUNCTION WITH FIXED π

LISTING 2: Customise PM MSE metrics in `Python`

```python
# Define the custom MSE metrics function
def pmmse(y_true, y_pred):
    a_pred1 = a_pred[:, 0]  # Extracting the predicted output a1
    a_pred2 = a_pred[:, 1]  # Extracting the predicted output a2

    pi= 0.92  # Define the pi values
    a_bar = 0.075 # claim 0.0751375

    n = y_true[:, -1]
    y = y_true[:, :-1]

    epsilon = 1e-10  # Small epsilon value to avoid zero or negative values in logarithm

    # Apply constraint: a1<a2, a1(0,0.0375), a2(0.0375,5), xi1,xi2 to be est
    a1 = 0.0375 * tf.math.exp(a_pred1*xi1) / (1 + tf.math.exp(a_pred1*xi1))
    a2 = tf.minimum(0.0375 + 0.0375 * tf.math.softplus(a_pred2*xi2), 5)

    m1 = tf.multiply(a1, n)
    m2 = tf.multiply(a2, n)

    p1 = tf.divide(
        tf.multiply(tf.math.pow(m1, y), tf.exp(-m1)),
        tf.exp(tf.math.lgamma(y + 1))
    )
    p2 = tf.divide(
        tf.multiply(tf.math.pow(m2, y), tf.exp(-m2)),
        tf.exp(tf.math.lgamma(y + 1))
    )

    #posterior prob of group 1
    z1= tf.divide(
    (tf.multiply(pi, p1)),
    (tf.multiply(pi, p1) + tf.multiply((1 - pi), p2))
    )

    # ahat using posterior prob
    ahat=tf.multiply(z1, a1) + tf.multiply((1 - z1), a2)

    pmmse = tf.reduce_mean(tf.square((y/n) - ahat))

    return pmmse
```

APPENDIX D. DEFINE BAYESIAN OPTIMIZATION

The package `:bayes_opt` in `Python` is implemented for BO search. The breakdown of each code is explained here:

- `init_points` $= 15$ specifies 15 initial random samples.
- `n_iter` $= 10$ specifies 10 iterations.
- `nn_cl_bo()` defines a DLNN model with hyperparameters and returns a PM MSE score.
- `optimizerL` contains the names of optimizers and `optimizerD` is dictionary maps of the optimizer names with the specified learning rate (`lr`).
- `nn_poi_fun` is an inner function that defines the structure of the DLNN with specified neurons and `DropOut` rate. The model is compiled with PM NLL loss and the specified optimizer with the learning rate.
- `KerasRegressor` is created with the defined model function `nn_poi_fun` with a specified number of epochs and batch size.

- `KFold` conducts K-fold cross-validation is used with five splits, shuffling the data, and a fixed random state for reproducibility.
- `cross_val_score` defines the score for cross-validation within the `KerasRegressor` model, and a customized PM MSE metric in Equation (7) is used as the scoring metric.

LISTING 3: Bayesian optimization search in `Python`

```
# Create a custom scoring function using the customized PMMSE
def custom_scoring(y_true, y_pred):
    # Convert numpy arrays to TensorFlow tensors
    y_true = tf.constant(y_true, dtype=tf.float32)
    y_pred = tf.constant(y_pred, dtype=tf.float32)

    # Calculate the score using the pmmse
    score = pmmse(y_true, y_pred)

    # Convert TensorFlow tensor back to numpy array and return the negative value (for
        minimization)
    return score.numpy()

# Create function
def nn_cl_bo(optimizer, learning_rate, batch_size, epochs):
    optimizerL = ['SGD', 'Adam', 'Adadelta', 'Adagrad']
    optimizerD= {'Adam':Adam(lr=learning_rate), 'SGD':SGD(lr=learning_rate),
    'Adadelta':Adadelta(lr=learning_rate),
                 'Adagrad':Adagrad(lr=learning_rate)}
    optimizer = optimizerD[optimizerL[round(optimizer)]]
    batch_size = round(batch_size)
    epochs = round(epochs)
    def nn_cl_fun():
        nn = Sequential()
        nn.add(Dense(45, input_dim=45, activation='linear'))
        nn.add(BatchNormalization())
        nn.add(Dense(30, activation='linear'))
        nn.add(Dense(15, activation='linear'))
        nn.add(Dropout(0.05))
        nn.add(Dense(10, activation='linear'))
        nn.add(Dense(5, activation='elu'))
        nn.add(Dense(2))
        nn.compile(loss=mixture_poisson_loss_with_exposure, optimizer=optimizer)
        return nn
    es = EarlyStopping(monitor="val_loss",verbose=0, patience=1)
    nn = KerasRegressor(build_fn=nn_cl_fun, epochs=epochs, batch_size=batch_size,
    verbose=0)
    kfold = KFold(n_splits=5, shuffle=True, random_state=123)
    score = cross_val_score(nn, x_train, y_train, scoring=make_scorer(custom_scoring,
    greater_is_better=False), cv=kfold, fit_params={'callbacks':[es]}).mean()
    return score

params_nn ={
    'optimizer':(0,3),
    'learning_rate':(0.0097, 0.01),
    'batch_size':(75,120),
    'epochs':(250, 450),
}

# Run Bayesian Optimization
nn_bo = BayesianOptimization(nn_cl_bo, params_nn,random_state=121)
nn_bo.maximize(init_points=15, n_iter=10)
```

APPENDIX E. DEFINE CONDITIONAL ESTIMATION FOR π

- The command `optimizer.build()` is used to explicitly prepare the optimizer to manage and optimize modeling parameters and additional trainable variables including π.

LISTING 4: PM DLNN Architecture in `Python`

```
1
2# Initialization of pi with a value around 0.9
3pi_initial = tf.constant(0.99, dtype=tf.float32, name="pi")
4pi = tf.Variable(pi_initial, trainable=True, dtype=tf.float32, name="pi")
5
6
7# Optimization loop with history tracking
8@tf.function
9def optimization_loop():
10    apred1_history = []
11    apred2_history = []
12    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
13    initial_learning_rate = 0.001
14    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
15        initial_learning_rate=initial_learning_rate,
16        decay_steps=10000,
17        decay_rate=0.9,  # Slight reduction at each step
18        staircase=True
19    )
20    optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
21
22
23    # Build the optimizer's state with trainable variables
24    optimizer.build(model.trainable_variables + [pi] + [b, c]) # Include b and c
25    # Lists to store the history of pi values
26    iteration_history = []
27    pi_history = []
28
29    # Reset the history lists at the beginning of the loop
30    train_log_likelihood_history = []
31    test_log_likelihood_history = []
32    mse_train_history = []
33    mse_test_history = []
34    mse_apred_train_history = []
35    mse_apred_test_history = []
36    weighted_mse_train_history = []
37    weighted_mse_test_history = []
38    mse_history = []  # Define mse_history here
39
40    # Early stopping parameters
41    patience = 5
42    best_val_log_likelihood = float('inf')
43    wait = 0  # Counter for how long we've waited
44
45    for k in range(1,251):
46        with tf.GradientTape(persistent=True) as tape:
47            # Compute the forward pass
48            y_pred = model(x2_train, training=True)
49
50            # Split y_pred into two parts, one for each distribution
51            y_pred1 = y_pred[:, 0]
52            y_pred2 = y_pred[:, 1]
53
54            # Compute the loss and MSE using the custom loss function
55            loss, mse = custom_loss(y2_train, y_pred1, y_pred2, pi)
56
57        # Compute the gradients of the loss with respect to model weights
58        model_gradients = tape.gradient(loss, model.trainable_variables)
59        optimizer.apply_gradients(zip(model_gradients, model.trainable_variables))
60
61        # Compute the gradient for pi
62        pi_gradients = tape.gradient(loss, pi)
63        optimizer.apply_gradients([(pi_gradients, pi)])
64        pi.assign(tf.clip_by_value(pi, 0.5, 1.0))
65
66        # Append the current value of pi to the history
67        iteration_history.append(k)
68        pi_history.append(pi.numpy())
69
70        mse_history.append(mse.numpy())
71
72        # Append the current value of pi to the history
```

```
        train_log_likelihood_history.append(loss.numpy())

        # Compute validation loss and MSE
        y_val_pred = model(x2_test, training=False)
        y_val_pred1 = y_val_pred[:, 0]
        y_val_pred2 = y_val_pred[:, 1]

        val_loss, val_mse = custom_loss(y2_test, y_val_pred1, y_val_pred2, pi)
        test_log_likelihood_history.append(val_loss.numpy())
        mse_test_history.append(val_mse.numpy())

        # Early stopping logic
        if val_loss.numpy() < best_val_log_likelihood:  # If current loss is better (
            smaller)
          best_val_log_likelihood = val_loss.numpy()  # Update best loss
          wait = 0  # Reset wait counter
        else:
          wait += 1  # Increment wait counter

        # If we have waited too long, stop the training
        if wait >= patience:
          print(f"Early stopping at iteration {k}")
          break
        tf.print(f"Iteration {k}, pi = {pi}, loss = {loss}, val_loss = {val_loss},
            mse_train = {mse}, mse_test = {val_mse}")
        tf.print(f"Iteration {k}, Val Loss: {val_loss.numpy()}, Best Loss: {
            best_val_log_likelihood}, Wait: {wait}")


        # Append apred1 and apred2 to the history
        apred1_history.append(y_pred1.numpy())
        apred2_history.append(y_pred2.numpy())

    # Save MSE history to a CSV file
    mse_df = pd.DataFrame({'Iteration': iteration_history, 'MSE': mse_history})
    mse_df.to_csv('mse_history.csv', index=False)

    # Save apred1 history to a CSV file
    apred1_df = pd.DataFrame(np.array(apred1_history).squeeze(), columns=[f"apred1_{i}"
        for i in range(len(apred1_history[0]))])

    # Transpose the DataFrame
    apred1_df_transposed = apred1_df.T

    # Save the transposed DataFrame to a CSV file
    apred1_df_transposed.to_csv('apred1_history_transposed.csv', index=False)

    # Save apred2 history to a CSV file
    apred2_df = pd.DataFrame(np.array(apred2_history).squeeze(), columns=[f"apred2_{i}"
        for i in range(len(apred2_history[0]))])
    # Transpose the DataFrame
    apred2_df_transposed = apred2_df.T
    apred2_df_transposed.to_csv('apred2_history_transposed.csv', index=False)



    # After the optimization loop, you'll have the updated pi value
    final_pi = pi.numpy()
    tf.print(f"Final pi: {final_pi}")
    tf.print(f"Final estimated values: b = {b.numpy()}, c = {c.numpy()}")

    # Create a pandas DataFrame with the pi history
    df = pd.DataFrame({'Iteration': iteration_history, 'Pi Estimate': pi_history})

    # Save the DataFrame to an Excel file
    df.to_excel('pi_history.xlsx', index=False)

    # Plot the history of pi values in a single line
    plt.plot(iteration_history, pi_history, label='pi History')
    plt.title('History of pi Estimates')
    plt.xlabel('Iteration')
    plt.ylabel('pi Value')
    plt.legend()
```

```
    plt.show()

    # Save log-likelihood history to a CSV file
    log_likelihood_df = pd.DataFrame({'Iteration': iteration_history, 'Negative Log-Likelihood': test_log_likelihood_history })
    log_likelihood_df.to_csv('test_log_likelihood_history .csv', index=False)

    # Plot the history of log-likelihood values
    plt.plot(iteration_history, test_log_likelihood_history , label='Negative Log-Likelihood History')
    plt.title('History of Test Negative Log-Likelihood Values')
    plt.xlabel('Iteration')
    plt.ylabel('Negative Log-Likelihood')
    plt.show()

    # Save log-likelihood history to a CSV file
    log_likelihood_df = pd.DataFrame({'Iteration': iteration_history, 'Negative Log-Likelihood': train_log_likelihood_history })
    log_likelihood_df.to_csv('train_log_likelihood_history .csv', index=False)

    # Plot the history of log-likelihood values
    plt.plot(iteration_history, train_log_likelihood_history , label='Negative Log-Likelihood History')
    plt.title('History of Train Negative Log-Likelihood Values')
    plt.xlabel('Iteration')
    plt.ylabel('Negative Log-Likelihood')
    plt.show()

    # Plot the history of MSE values
    plt.plot(iteration_history, mse_history, label='MSE History')
    plt.title('History of Mean Squared Error (MSE) Values')
    plt.xlabel('Iteration')
    plt.ylabel('MSE')
    plt.show()

optimization_loop()
```