Distributional Regression

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Eric Dong & Patrick Laub

Warning

This page is out of date for 2024, and will be updated shortly.

Lecture Outline

- **Uncertainty**
- Aleatoric Uncertainty

Quiz

Question: *If you decide to predict the claim amount of Bob using a deep learning model, which source(s) of uncertainty are you confronting?*

1. The inherent variability of the data-generating process.
2. Parameter error.
3. Model error.
4. Data uncertainty.
5. All of the above.

Answer

All of the above!

There are two major types of uncertainty in statistical or machine learning:

- Aleatoric uncertainty
- Epistemic uncertainty

Since there is no consensus on the definitions of aleatoric and epistemic uncertainty, we provide the most acknowledged definitions in the following slides.

Aleatoric Uncertainty

Qualitative Definition

Aleatoric uncertainty refers to the statistical variability and inherent noise with data distribution that modelling cannot explain.

Quantitative Definition

$$\mathrm{Ale}(Y|\boldsymbol{x}) = \mathbb{V}[Y|\boldsymbol{x}],$$

i.e., if $Y|\boldsymbol{x} \sim \mathcal{N}(\mu, \sigma^2)$, the aleatoric uncertainty would be σ^2. Simply, it is the conditional variance of the response variable Y given features/covariates $\boldsymbol{x}$.

UNSW SYDNEY

Epistemic Uncertainty

Qualitative Definition

Epistemic uncertainty refers to the lack of knowledge, limited data information, parameter errors and model errors.

Quantitative Definition

$$\mathrm{Epi}(Y|\boldsymbol{x}) = \mathrm{Uncertainty}(Y|\boldsymbol{x}) - \mathrm{Ale}(Y|\boldsymbol{x}),$$

i.e., the total uncertainty subtracting the aleatoric uncertainty $\mathbb{V}[Y|\boldsymbol{x}]$ would be the epistemic uncertainty.

UNSW SYDNEY

Uncertainty

Let's go back to the question at the beginning:

If you decide to predict the claim amount of an individual using a deep learning model, which source(s) of uncertainty are you dealing with?

1. The inherent variability of the data-generating process $\rightarrow$ aleatoric uncertainty.
2. Parameter error $\rightarrow$ epistemic uncertainty.
3. Model error $\rightarrow$ epistemic uncertainty.
4. Data uncertainty $\rightarrow$ epistemic uncertainty.

Code: Data

```python
import pandas as pd
sev_df = pd.read_csv('freMTPL2sev.csv')
freq_df = pd.read_csv('freMTPL2freq.csv')

# Create a copy of freq dataframe without 'claimfreq' column
freq_without_claimfreq = freq_df.drop(columns=['ClaimNb'])

# Merge severity dataframe with freq_without_claimfreq dataframe
new_sev_df = pd.merge(sev_df, freq_without_claimfreq, on='IDpol',
                                                      how='left')
new_sev_df = new_sev_df.dropna()
new_sev_df = new_sev_df.drop("IDpol", axis=1)
new_sev_df[:2]
```

	ClaimAmount	Exposure	VehPower	VehAge	DrivAge	Bo
0	995.20	0.59	11.0	0.0	39.0	56.0
1	1128.12	0.95	4.0	1.0	49.0	50.0

UNSW SYDNEY

Code: Preprocessing

```python
X_train, X_test, y_train, y_test = train_test_split(
  new_sev_df.drop("ClaimAmount", axis=1),
  new_sev_df["ClaimAmount"],
  random_state=2023)

# Reset each index to start at 0 again.
X_train = X_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
y_test = y_test.reset_index(drop=True)
```

Code: Preprocessing

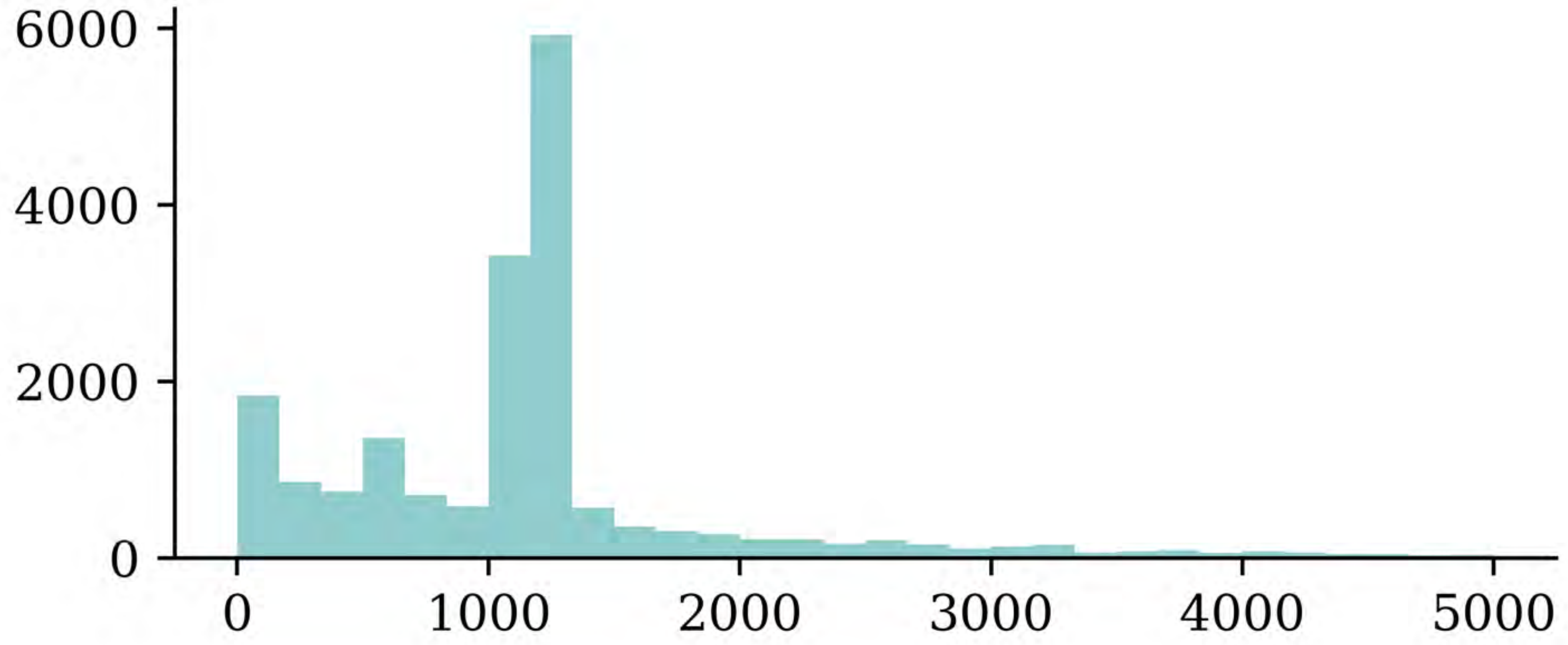
```python
# Transformation
ct = make_column_transformer(
  (OrdinalEncoder(), ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
   verbose_feature_names_out=False
)

# We don't apply entity embedding
X_train_ct = ct.fit_transform(X_train)
X_test_ct = ct.transform(X_test)
X_train = X_train_ct.drop(["VehBrand", "Region"], axis=1)
X_test = X_test_ct.drop(["VehBrand", "Region"], axis=1)
```

- `VehGas=1` if the car gas is regular.
- `Area=0` represents the rural area, and `Area=5` represents the urban center.

Histogram of the `ClaimAmount`

```python
plt.hist(y_train[y_train < 5000], bins=30);
```



Lecture Outline

- Uncertainty
- **Aleatoric Uncertainty**

GLM

The generalised linear model (GLM) is a statistical regression model that estimates the conditional mean of the response variable Y given an instance $\boldsymbol{x}$ via a link function g:

$$\mathbb{E}[Y|\boldsymbol{x}] = \mu(\boldsymbol{x}; \boldsymbol{\beta}_{\mathrm{GLM}}) = g^{-1}(\langle \boldsymbol{\beta}_{\mathrm{GLM}}, \boldsymbol{x} \rangle),$$

where

- $\boldsymbol{x} \in \mathbb{R}^{d_{\boldsymbol{x}}}$ is the vector of explanatory variables, with $d_{\boldsymbol{x}}$ denoting its dimension.
- $\boldsymbol{\beta}_{\mathrm{GLM}}$ represents the vector of regression coefficients.
- $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ represents the inner product of $\boldsymbol{a}$ and $\boldsymbol{b}$.

Gamma GLM

Suppose a fitted gamma GLM model has

- a log link function $g(x) = \log(x)$ and
- regression coefficients $\boldsymbol{\beta}_{\text{GLM}} = (\beta_0, \beta_1, \beta_2, \beta_3)$.

Then, it estimates the conditional mean of Y given a new instance $\boldsymbol{x} = (1, x_1, x_2, x_3)$ as follows:

$$\mathbb{E}[Y|\boldsymbol{x}] = g^{-1}(\langle \boldsymbol{\beta}_{\text{GLM}}, \boldsymbol{x} \rangle) = \exp\big(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3\big).$$

A GLM can model any other exponential family distribution using an appropriate link function g.

UNSW SYDNEY

“Loss Function” for a Gamma GLM

If $Y|\boldsymbol{x}$ is a gamma r.v., we can parameterise its density by its mean $\mu(\boldsymbol{x};\boldsymbol{\beta})$ and dispersion parameter ϕ:

$$f_{Y|\boldsymbol{X}}(y|\boldsymbol{x},\boldsymbol{\beta},\phi) = \frac{(\mu(\boldsymbol{x};\boldsymbol{\beta})\cdot\phi)^{-1/\phi}}{\Gamma(1/\phi)} \cdot y^{1/\phi-1} \cdot \mathrm{e}^{-y/(\mu(\boldsymbol{x};\boldsymbol{\beta})\cdot\phi)}.$$

The “loss function” for a gamma GLM is typically the negative log-likelihood (NLL):

$$\sum_{i=1}^{N} -\log f_{Y|\boldsymbol{X}}(y_i|\boldsymbol{x}_i,\boldsymbol{\beta},\phi) \propto \sum_{i=1}^{N} \log\mu(\boldsymbol{x}_i;\boldsymbol{\beta}) + \frac{y_i}{\mu(\boldsymbol{x}_i;\boldsymbol{\beta})} + \text{const},$$

i.e., we ignore the dispersion parameter ϕ while estimating the regression coefficients.

Fitting Steps

Step 1. Use the advanced second derivative iterative method to find the regression coefficients:

$$\boldsymbol{\beta}_{\text{GLM}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} \log \mu(\boldsymbol{x}_i; \boldsymbol{\beta}) + \frac{y_i}{\mu(\boldsymbol{x}_i; \boldsymbol{\beta})}$$

Step 2. Estimate the dispersion parameter:

$$\phi_{\text{GLM}} = \frac{1}{N - d_{\boldsymbol{x}}} \sum_{i=1}^{N} \frac{(y_i - \mu(\boldsymbol{x}_i; \boldsymbol{\beta}_{\text{GLM}}))^2}{\mu(\boldsymbol{x}_i; \boldsymbol{\beta}_{\text{GLM}})^2}$$

Code: Gamma GLM

In Python, we can fit a gamma GLM as follows:

```python
import statsmodels.api as sm

# Add a column of ones to include an intercept in the model
X_train_design = sm.add_constant(X_train)

# Create a Gamma GLM with a log link function
gamma_GLM = sm.GLM(y_train, X_train_design,
            family=sm.families.Gamma(sm.families.links.Log()))

# Fit the model
gamma_GLM = gamma_GLM.fit()

# Dispersion Parameter
mus = gamma_GLM.predict(X_train_design)
residuals = mus-y_train
variance = mus**2
dof = (len(y_train)-X_train.shape[1])
phi_GLM =  np.sum(residuals**2/variance)/dof
print(phi_GLM)
```

```
59.6306232357824
```

CANN

The Combined Actuarial Neural Network is a novel actuarial neural network architecture proposed by Schelldorfer and Wüthrich (2019). We summarise the CANN approach as follows:

- Find the coefficients $\boldsymbol{\beta}_{\text{GLM}}$ of the GLM with a link function $g(\cdot)$.
- Find the weights $\boldsymbol{w}_{\text{CANN}}$ of a neural network $\mathcal{M}_{\text{CANN}} : \mathbb{R}^{d_{\boldsymbol{x}}} \to \mathbb{R}$.
- Given a new instance $\boldsymbol{x}$, we have

$$\mathbb{E}[Y|\boldsymbol{x}] = g^{-1}\Big(\langle \boldsymbol{\beta}_{\text{GLM}}, \boldsymbol{x}\rangle + \mathcal{M}_{\text{CANN}}(\boldsymbol{x}; \boldsymbol{w}_{\text{CANN}})\Big).$$

Architecture

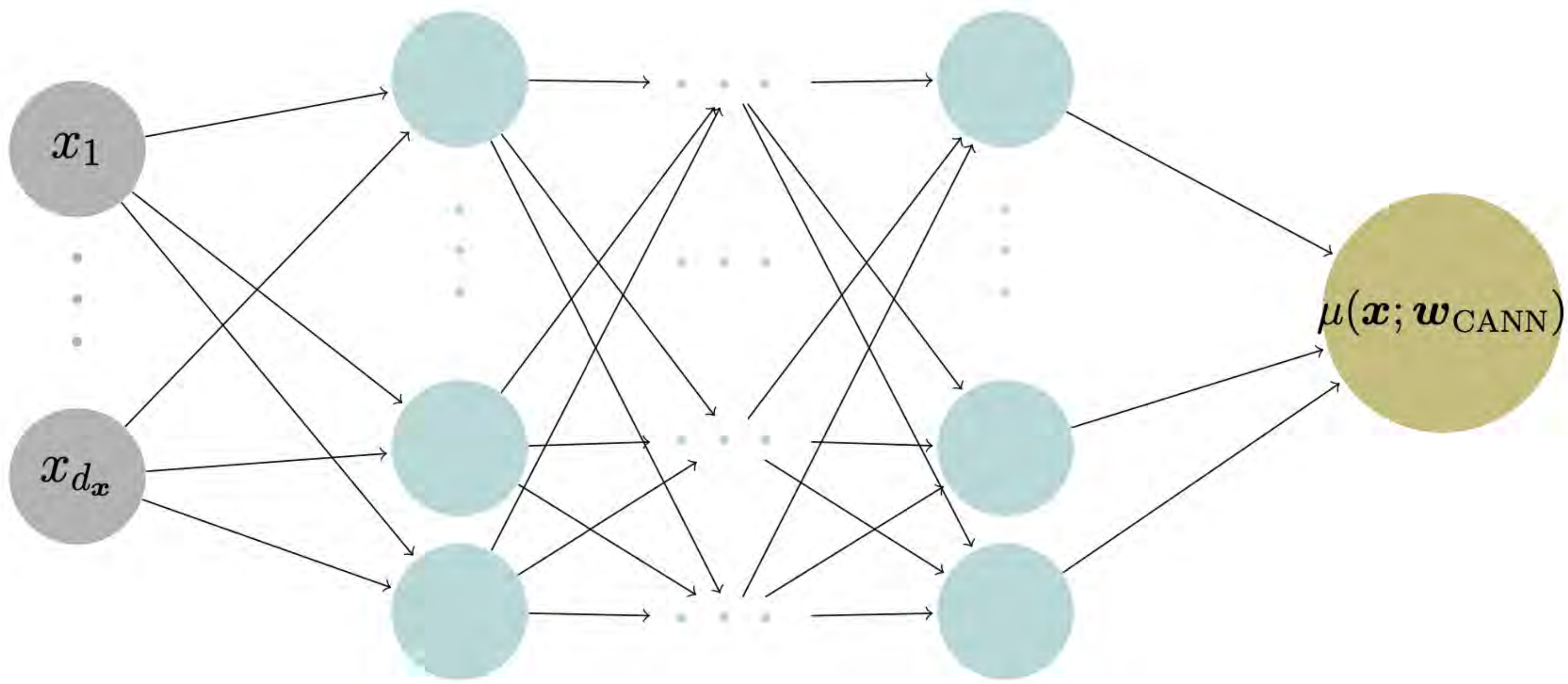


Figure: CANN approach.

Code: Architecture

```
gamma_GLM.params
```

```
const          7.786576
Area          -0.073226
VehGas         0.082292
                  ...
DrivAge       -0.022147
BonusMalus     0.157204
Density        0.010539
Length: 9, dtype: float64
```

```
# Ensure reproducibility
random.seed(1); tf.random.set_seed(1)

# Pre-defined constants
glm_weights = gamma_GLM.params.iloc[1:]
glm_bias = gamma_GLM.params.iloc[0]

# Define model inputs
inputs = Input(shape=X_train.shape[1:])

# Non-trainable GLM linear part
glm_logmu = Dense(1, activation='linear', trainable=False,
                     kernel_initializer=Constant(glm_weights),
                     bias_initializer=Constant(glm_bias))(inputs)

# Neural network layers
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
cann_logmu = Dense(1, activation='linear')(x)
```

Code: Loss Function

```python
# Combine GLM and CANN estimates
CANN = Model(inputs, Concatenate(axis=1)([cann_logmu, glm_logmu]))
```

We need to customise the loss function for CANN.

```python
def CANN_negative_log_likelihood(y_true, y_pred):
    #the new mean estimate
    CANN_logmu = y_pred[:, 0]
    GLM_logmu = y_pred[:, 1]
    mu = tf.math.exp(CANN_logmu + GLM_logmu)

    # Compute the negative log likelihood of the Gamma distribution
    nll = tf.reduce_mean(CANN_logmu + GLM_logmu + y_true/mu)

    return nll
```

Code: Model Training

```
CANN.compile(optimizer="adam", loss=CANN_negative_log_likelihood)
hist = CANN.fit(X_train, y_train,
    epochs=300,
    callbacks=[EarlyStopping(patience=30)],
    verbose=0,
    batch_size=64,
    validation_split=0.2)
```

Find the dispersion parameter.

```
mus = np.exp(np.sum(CANN.predict(X_train, verbose=0), axis = 1))
residuals = mus-y_train
variance = mus**2
dof = (len(y_train)-X_train.shape[1])
phi_CANN =  np.sum(residuals**2/variance) / dof
print(phi_CANN)
```

```
98.60976911896634
```

UNSW SYDNEY

Mixture Distribution

Given a finite set of resulting random variables $(Y_1, \ldots, Y_K)$, one can generate a multinomial random variable $Y \sim \text{Multinomial}(1, \boldsymbol{\pi})$. Meanwhile, Y can be regarded as a mixture of $Y_1, \ldots, Y_K$, i.e.,

$$Y = \begin{cases} Y_1 & \text{w.p. } \pi_1, \\ \vdots & \vdots \\ Y_K & \text{w.p. } \pi_K, \end{cases}$$

where we define a set of finite set of weights $\boldsymbol{\pi} = (\pi_1 \ldots, \pi_K)$ such that $\pi_k \geq 0$ for $k \in \{1, \ldots, K\}$ and $\sum_{k=1}^{K} \pi_k = 1$.

Mixture Distribution

Let $f_{Y_k|\boldsymbol{X}}$ and $F_{Y_k|\boldsymbol{X}}$ be the probability density function and the cumulative density function, respectively, of $Y_k|\boldsymbol{X}$ for all $k \in \{1, ..., K\}$. The random variable $Y|\boldsymbol{X}$, which mixes $Y_k|\boldsymbol{X}$'s with weights π_k's, has the density function

$$f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k(\boldsymbol{x}) f_k(y|\boldsymbol{x}),$$

and the cumulative density function

$$F_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k(\boldsymbol{x}) F_k(y|\boldsymbol{x}).$$

Mixture Density Network

A mixture density network (MDN) $\mathcal{M}_{\boldsymbol{w}^*}$ outputs each distribution component's mixing weights and parameters of Y given the input features $\boldsymbol{x}$, i.e.,

$$\mathcal{M}_{\boldsymbol{w}^*}(\boldsymbol{x}) = (\boldsymbol{\pi}(\boldsymbol{x}; \boldsymbol{w}^*), \boldsymbol{\theta}(\boldsymbol{x}; \boldsymbol{w}^*)),$$

where $\boldsymbol{w}^*$ is the networks' weights found by minimising the following negative log-likelihood loss function

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}) = -\sum_{i=1}^{N} \log f_{Y|\boldsymbol{x}}(y_i|\boldsymbol{x}, \boldsymbol{w}^*),$$

where $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ is the training dataset.

UNSW SYDNEY

Mixture Density Network

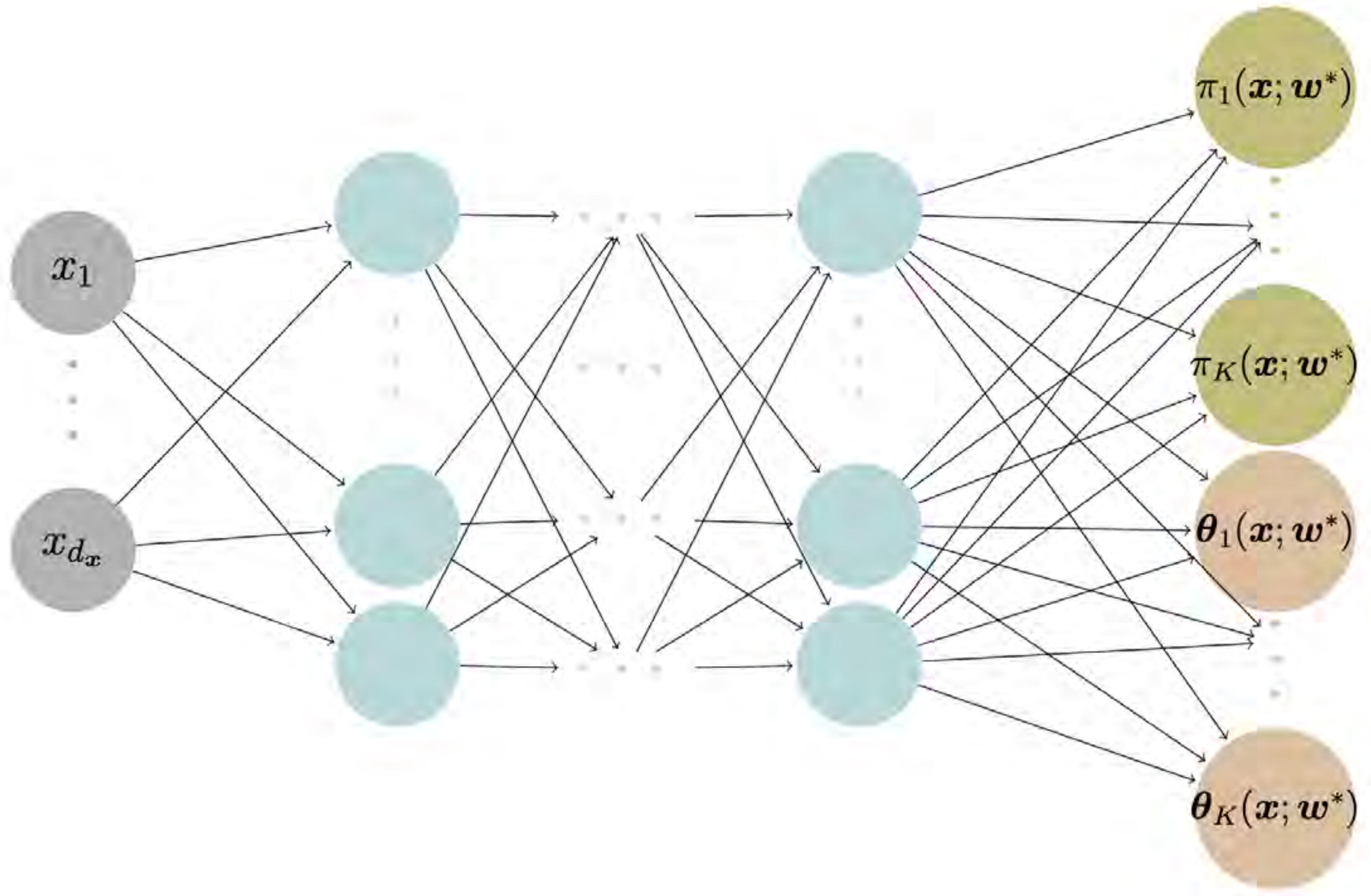


Figure: An MDN that outputs the parameters for a K component mixture distribution. $\boldsymbol{\theta}_k(\boldsymbol{x};\boldsymbol{w}^*) = (\theta_{k,1}(\boldsymbol{x};\boldsymbol{w}^*), \ldots, \theta_{k,|\boldsymbol{\theta}_k|}(\boldsymbol{x};\boldsymbol{w}^*))$ consists of the parameter estimates for the kth mixture component.

Model Specification

Suppose there are two types of claims:

- Type I: $Y_1|\boldsymbol{x} \sim \text{Gamma}(\alpha_1(\boldsymbol{x}), \beta_1(\boldsymbol{x}))$ and,
- Type II: $Y_2|\boldsymbol{x} \sim \text{Gamma}(\alpha_2(\boldsymbol{x}), \beta_2(\boldsymbol{x}))$.

The density of the actual claim amount $Y|\boldsymbol{x}$ follows

$$
\begin{aligned}
f_{Y|\boldsymbol{X}}(y|\boldsymbol{x}) = \pi_1(\boldsymbol{x}) \cdot \frac{\beta_1(\boldsymbol{x})^{\alpha_1(\boldsymbol{x})}}{\Gamma(\alpha_1(\boldsymbol{x}))} \mathrm{e}^{-\beta_1(\boldsymbol{x})y} y^{\alpha_1(\boldsymbol{x})-1} \\
+ (1 - \pi_1(\boldsymbol{x})) \cdot \frac{\beta_2(\boldsymbol{x})^{\alpha_2(\boldsymbol{x})}}{\Gamma(\alpha_2(\boldsymbol{x}))} \mathrm{e}^{-\beta_2(\boldsymbol{x})y} y^{\alpha_2(\boldsymbol{x})-1}.
\end{aligned}
$$

where $\pi_1(\boldsymbol{x})$ is the probability of a Type I claim given $\boldsymbol{x}$.

Output

The aim is to find the optimum weights

$$\boldsymbol{w}^* = \underset{w}{\arg\min}\, \mathcal{L}(\mathcal{D}, \boldsymbol{w})$$

for the Gamma mixture density network $\mathcal{M}_{\boldsymbol{w}^*}$ that outputs the mixing weights, shapes and scales of Y given the input features $\boldsymbol{x}$, i.e.,

$$\begin{aligned}\mathcal{M}_{\boldsymbol{w}^*}(\boldsymbol{x}) = (&\pi_1(\boldsymbol{x}; \boldsymbol{w}^*), \pi_2(\boldsymbol{x}; \boldsymbol{w}^*),\\ &\alpha_1(\boldsymbol{x}; \boldsymbol{w}^*), \alpha_2(\boldsymbol{x}; \boldsymbol{w}^*),\\ &\beta_1(\boldsymbol{x}; \boldsymbol{w}^*), \beta_2(\boldsymbol{x}; \boldsymbol{w}^*)).\end{aligned}$$

Architecture

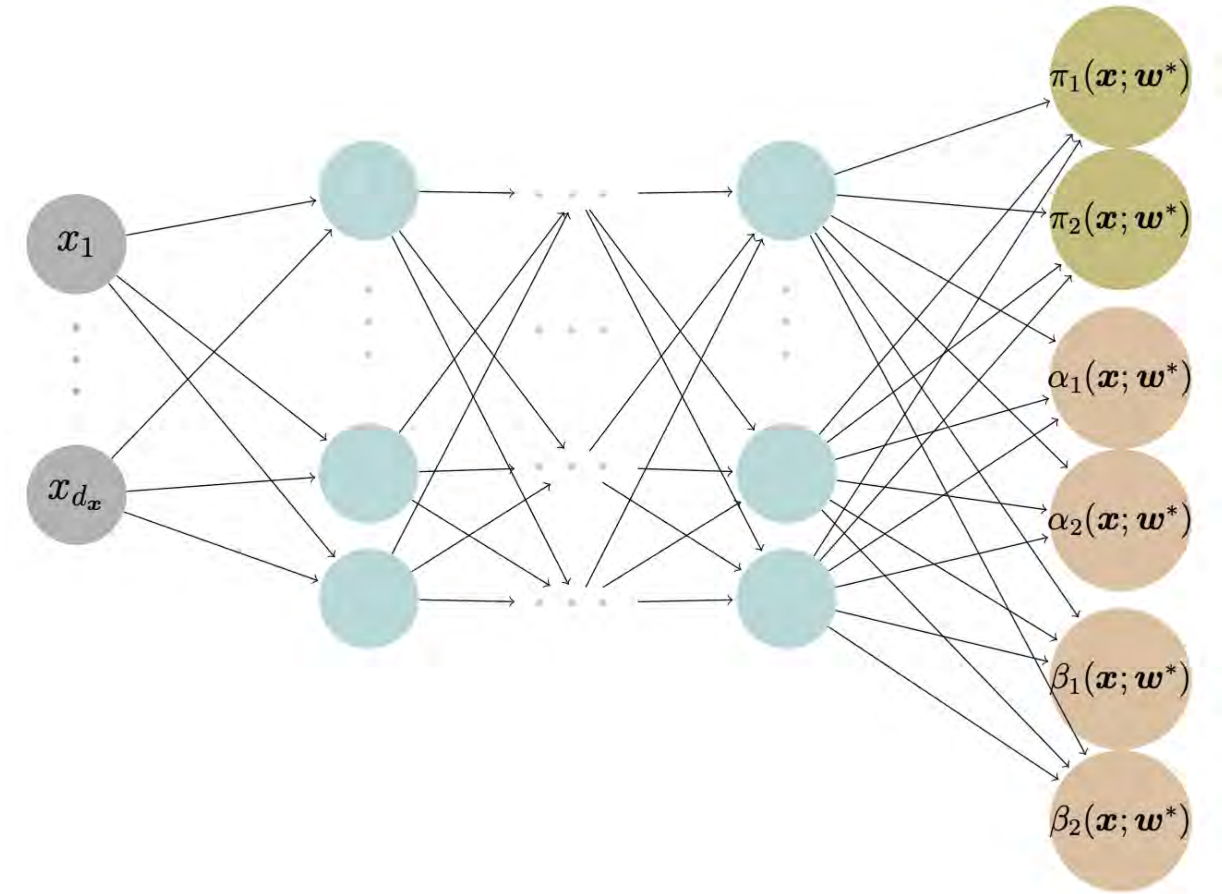


Figure: We demonstrate the structure of a gamma MDN that outputs the parameters for a gamma mixture with two components.

Code: Architecture

The following code resembles the architecture of the architecture of the gamma MDN from the previous slide.

```python
# Ensure reproducibility
random.seed(1); tf.random.set_seed(1)

inputs = Input(shape=X_train.shape[1:])

# Two hidden layers
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)

pis = Dense(2, activation='softmax')(x) #mixing weights
alphas = Dense(2, activation='exponential')(x) #shape parameters
betas = Dense(2, activation='exponential')(x) #scale parameters

# `y_pred` will now have 6 columns
gamma_mdn = Model(inputs, Concatenate(axis=1)([pis, alphas, betas]))
```

Loss Function

The negative log-likelihood loss function is given by

$$\mathcal{L}(\mathcal{D}, \boldsymbol{w}) = -\sum_{i=1}^{N} \log\ f_{Y|\boldsymbol{x}}(y_i|\boldsymbol{x}, \boldsymbol{w})$$

where the $f_{Y|\boldsymbol{x}}(y_i|\boldsymbol{x}, \boldsymbol{w})$ is defined by

$$\pi_1(\boldsymbol{x};\boldsymbol{w}) \cdot \frac{\beta_1(\boldsymbol{x};\boldsymbol{w})^{\alpha_1(\boldsymbol{x};\boldsymbol{w})}}{\Gamma(\alpha_1(\boldsymbol{x};\boldsymbol{w}))} \mathrm{e}^{-\beta_1(\boldsymbol{x};\boldsymbol{w})y} y^{\alpha_1(\boldsymbol{x};\boldsymbol{w})-1}$$
$$+ (1-\pi_1(\boldsymbol{x};\boldsymbol{w})) \cdot \frac{\beta_2(\boldsymbol{x};\boldsymbol{w})^{\alpha_2(\boldsymbol{x};\boldsymbol{w})}}{\Gamma(\alpha_2(\boldsymbol{x};\boldsymbol{w}))} \mathrm{e}^{-\beta_2(\boldsymbol{x};\boldsymbol{w})y} y^{\alpha_2(\boldsymbol{x};\boldsymbol{w})-1}$$

Code: Loss Function

We employ functions from `tensorflow_probability` to code the loss function for the gamma MDN. The `MixtureSameFamily` function facilitates defining a mixture distribution all components from the same distribution but have different parametrization.

```python
import tensorflow_probability as tfp
tfd = tfp.distributions
K = 2 # number of mixture components

def gamma_mixture_NLL(y_true, y_pred):
    K = y_pred.shape[1] // 3
    pis =  y_pred[:, :K]
    alphas = y_pred[:, K:2*K]
    betas = y_pred[:, 2*K:3*K]

    # The mixture distribution is a MixtureSameFamily distribution
    mixture_distribution = tfd.MixtureSameFamily(
        mixture_distribution=tfd.Categorical(probs=pis),
        components_distribution=tfd.Gamma(alphas, betas))

    # The loss is the negative log-likelihood of the data
    return -mixture_distribution.log_prob(y_true)
```

Code: Model Training

```
# Employ the loss function from previous slide
gamma_mdn.compile(optimizer="adam", loss=gamma_mixture_NLL)

hist = gamma_mdn.fit(X_train, y_train,
    epochs=300,
    callbacks=[EarlyStopping(patience=30)],
    verbose=0,
    batch_size=64,
    validation_split=0.2)
```

Proper Scoring Rules

Definition

The scoring rule $S : \mathcal{F} \times \mathbb{R} \to \bar{\mathbb{R}}$ is proper relative to the class $\mathcal{F}$ if

$$S(G, G) \leq S(F, G)$$

for all $F, G \in \mathcal{F}$. It is strictly proper if equality holds only if $F = G$.

Examples:

- Logarithmic Score (NLL)
- Continuous Ranked Probability Score (CRPS)

Proper Scoring Rules

Logarithmic Score (NLL)

The logarithmic score is defined as

$$\mathrm{LogS}(f, y) = -\log f(y),$$

where f is the predictive density.

Continuous Ranked Probability Score (CRPS)

The continuous ranked probability score is defined as

$$\mathrm{crps}(F, y) = \int_{-\infty}^{\infty} (F(t) - 1_{t \geq y})^2 \, \mathrm{d}t,$$

where F is the cumulative distribution function.

UNSW SYDNEY

Code: NLL

```python
from scipy.stats import gamma

def gamma_nll(mean, dispersion, y):
    # Calculate shape and scale parameters from mean and dispersion
    shape = 1 / dispersion; scale = mean * dispersion

    # Create a gamma distribution object
    gamma_dist = gamma(a=shape, scale=scale)

    return -np.mean(gamma_dist.logpdf(y))

# GLM
X_test_design = sm.add_constant(X_test)
mus = gamma_GLM.predict(X_test_design)
NLL_GLM = gamma_nll(mus, phi_GLM, y_test)

# CANN
mus = np.exp(np.sum(CANN.predict(X_test, verbose=0), axis = 1))
NLL_CANN = gamma_nll(mus, phi_CANN, y_test)

# MDN
NLL_MDN = gamma_mdn.evaluate(X_test, y_test, verbose=0)
```

Model Comparisons

```python
print(f'GLM: {round(NLL_GLM, 2)}')
print(f'CANN: {round(NLL_CANN, 2)}')
print(f'MDN: {round(NLL_MDN, 2)}')
```

```
GLM: 11.02
CANN: 11.5
MDN: 8.67
```

Package Versions

```
from watermark import watermark
print(watermark(python=True, packages="keras,matplotlib,numpy,pandas,seaborn,scipy,torch
```

```
Python implementation: CPython
Python version       : 3.11.9
IPython version      : 8.24.0

keras                 : 3.3.3
matplotlib            : 3.9.0
numpy                 : 1.26.4
pandas                : 2.2.2
seaborn               : 0.13.2
scipy                 : 1.11.0
torch                 : 2.3.1
tensorflow            : 2.16.1
tensorflow_probability: 0.24.0
tf_keras              : 2.16.0
```

Glossary

- aleatoric and epistemic uncertainty
- Bayesian neural network
- deep ensembles
- dropout
- CANN
- GLM
- MDN
- mixture distribution
- posterior sampling
- proper scoring rule
- uncertainty quantification
- variational approximation