

Uncertainty Quantification

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Eric Dong & Patrick Laub



Lecture Outline

- **Uncertainty**
- Aleatoric Uncertainty
- Epistemic 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 in data distribution that modelling cannot explain.

Quantitative Definition

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

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} .



Epistemic Uncertainty

Qualitative Definition

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

Quantitative Definition

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

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



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 → aleatoric uncertainty.
2. Parameter error → epistemic uncertainty.
3. Model error → epistemic uncertainty.
4. Data uncertainty → epistemic uncertainty.



Code: Data

```

1 import pandas as pd
2 sev_df = pd.read_csv('freMTPL2sev.csv')
3 freq_df = pd.read_csv('freMTPL2freq.csv')
4
5 # Create a copy of freq dataframe without 'claimfreq' column
6 freq_without_claimfreq = freq_df.drop(columns=['ClaimNb'])
7
8 # Merge severity dataframe with freq_without_claimfreq dataframe
9 new_sev_df = pd.merge(sev_df, freq_without_claimfreq, on='IDpol',
10                       how='left')
11 new_sev_df = new_sev_df.dropna()
12 new_sev_df = new_sev_df.drop("IDpol", axis=1)
13 new_sev_df[:2]

```

	ClaimAmount	Exposure	VehPower	VehAge	DrvAge	Bor
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



Code: Preprocessing

```
1 X_train, X_test, y_train, y_test = train_test_split(  
2     new_sev_df.drop("ClaimAmount", axis=1),  
3     new_sev_df["ClaimAmount"],  
4     random_state=2023)  
5  
6 # Reset each index to start at 0 again.  
7 X_train = X_train.reset_index(drop=True)  
8 X_test = X_test.reset_index(drop=True)  
9 y_train = y_train.reset_index(drop=True)  
10 y_test = y_test.reset_index(drop=True)
```



Code: Preprocessing

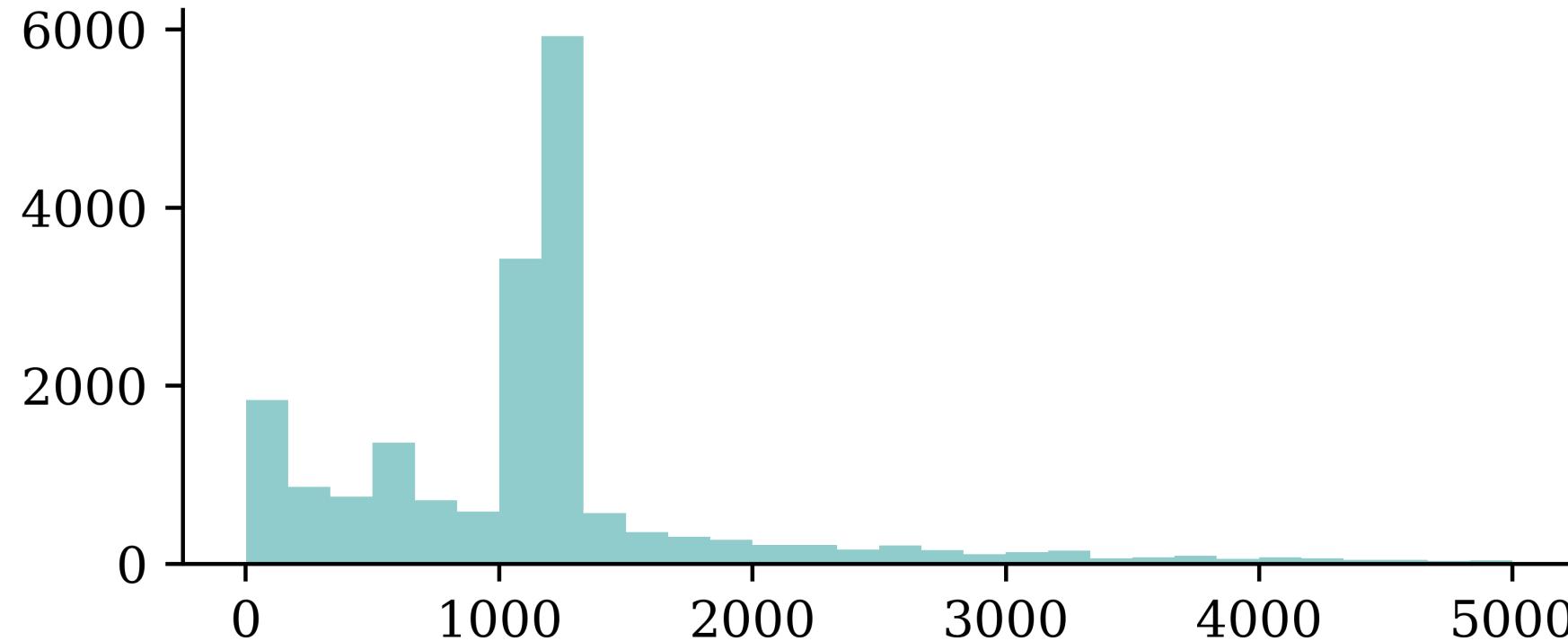
```
1 # Transformation
2 ct = make_column_transformer(
3     (OrdinalEncoder(), ["VehBrand", "Region", "Area", "VehGas"]),
4     remainder=StandardScaler(),
5     verbose_feature_names_out=False
6 )
7
8 # We don't apply entity embedding
9 X_train_ct = ct.fit_transform(X_train)
10 X_test_ct = ct.transform(X_test)
11 X_train = X_train_ct.drop(["VehBrand", "Region"], axis=1)
12 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

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



Lecture Outline

- Uncertainty
- **Aleatoric Uncertainty**
- Epistemic 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 \mathbf{x} via a link function g :

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

where

- $\mathbf{x} \in \mathbb{R}^{d_x}$ is the vector of explanatory variables, with d_x denoting its dimension.
- $\boldsymbol{\beta}_{\text{GLM}}$ represents the vector of regression coefficients.
- $\langle \mathbf{a}, \mathbf{b} \rangle$ represents the inner product of \mathbf{a} and \mathbf{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 $\mathbf{x} = (1, x_1, x_2, x_3)$ as follows:

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

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



“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 :

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

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:

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

Step 2. Estimate the dispersion parameter:

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



Code: Gamma GLM

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

```
1 import statsmodels.api as sm
2
3 # Add a column of ones to include an intercept in the model
4 X_train_design = sm.add_constant(X_train)
5
6 # Create a Gamma GLM with a log link function
7 gamma_GLM = sm.GLM(y_train, X_train_design,
8                      family=sm.families.Gamma(sm.families.links.Log()))
9
10 # Fit the model
11 gamma_GLM = gamma_GLM.fit()
12
13 # Dispersion Parameter
14 mus = gamma_GLM.predict(X_train_design)
15 residuals = mus-y_train
16 variance = mus**2
17 dof = (len(y_train)-X_train.shape[1])
18 phi_GLM = np.sum(residuals**2/variance)/dof
19 print(phi_GLM)
```

59.630623235781364



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 β_{GLM} of the GLM with a link function $g(\cdot)$.
- Find the weights w_{CANN} of a neural network $\mathcal{M}_{\text{CANN}} : \mathbb{R}^{d_x} \rightarrow \mathbb{R}$.
- Given a new instance x , we have

$$\mathbb{E}[Y|x] = g^{-1}\left(\langle \beta_{\text{GLM}}, x \rangle + \mathcal{M}_{\text{CANN}}(x; w_{\text{CANN}})\right).$$



Architecture

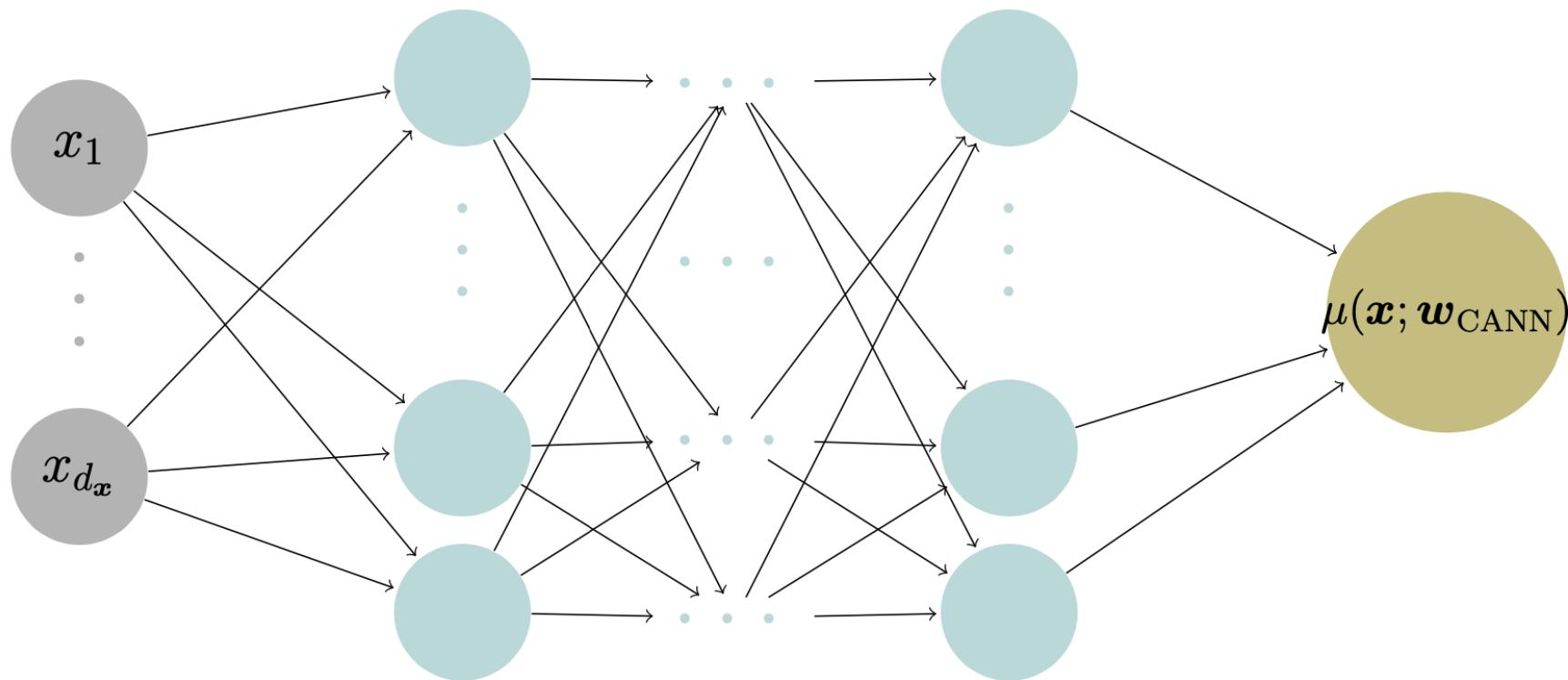


Figure: CANN approach.



Code: Architecture

```

1 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

1 # Ensure reproducibility
2 random.seed(1); tf.random.set_seed(1)
3
4 # Pre-defined constants
5 glm_weights = gamma_GLM.params.iloc[1:]
6 glm_bias = gamma_GLM.params.iloc[0]
7
8 # Define model inputs
9 inputs = Input(shape=X_train.shape[1:])
10
11 # Non-trainable GLM linear part
12 glm_logmu = Dense(1, activation='linear', trainable=False,
13                     kernel_initializer=Constant(glm_weights),
14                     bias_initializer=Constant(glm_bias))(inputs)
15
16 # Neural network layers
17 x = Dense(64, activation='relu')(inputs)
18 x = Dense(64, activation='relu')(x)
19 cann_logmu = Dense(1, activation='linear')(x)

```



Code: Loss Function

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

We need to customise the loss function for CANN.

```
1 def CANN_negative_log_likelihood(y_true, y_pred):
2     #the new mean estimate
3     CANN_logmu = y_pred[:, 0]
4     GLM_logmu = y_pred[:, 1]
5     mu = tf.math.exp(CANN_logmu + GLM_logmu)
6
7     # Compute the negative log likelihood of the Gamma distribution
8     nll = tf.reduce_mean(CANN_logmu + GLM_logmu + y_true/mu)
9
10    return nll
```



Code: Model Training

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

Find the dispersion parameter.

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

117.92906115555793



Mixture Distribution

Given a finite set of resulting random variables (Y_1, \dots, 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, \dots, 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 $\boldsymbol{\pi} = (\pi_1, \dots, \pi_K)$ such that $\pi_k \geq 0$ for $k \in \{1, \dots, K\}$ and $\sum_{k=1}^K \pi_k = 1$.



Mixture Distribution

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

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

and the cumulative density function

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



Mixture Density Network

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

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

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

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

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



Mixture Density Network

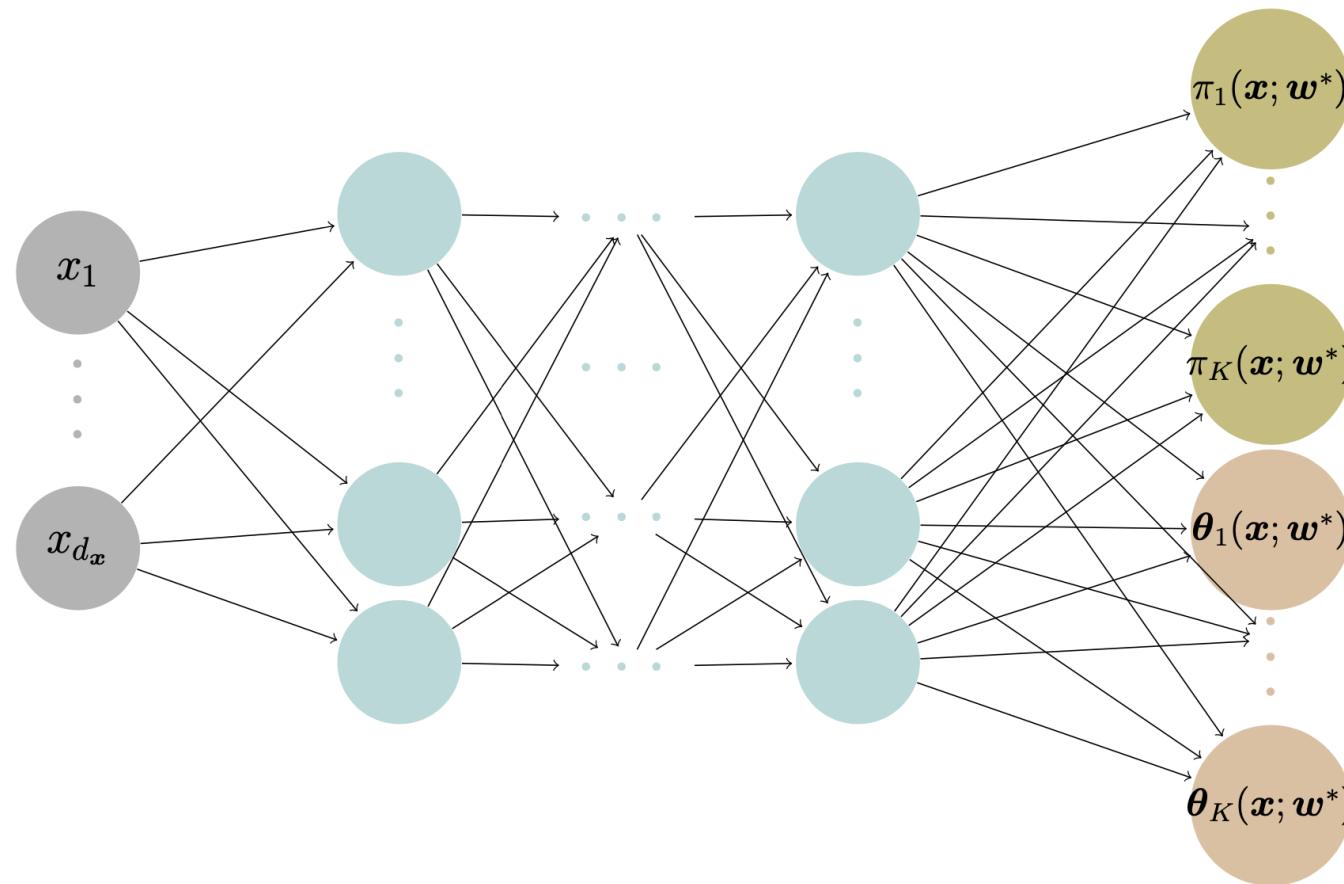


Figure: An MDN that outputs the parameters for a K component mixture distribution. $\boldsymbol{\theta}_k(\mathbf{x}; \mathbf{w}^*) = (\theta_{k,1}(\mathbf{x}; \mathbf{w}^*), \dots, \theta_{k,|\boldsymbol{\theta}_k|}(\mathbf{x}; \mathbf{w}^*))$ consists of the parameter estimates for the k th mixture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

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

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

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

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

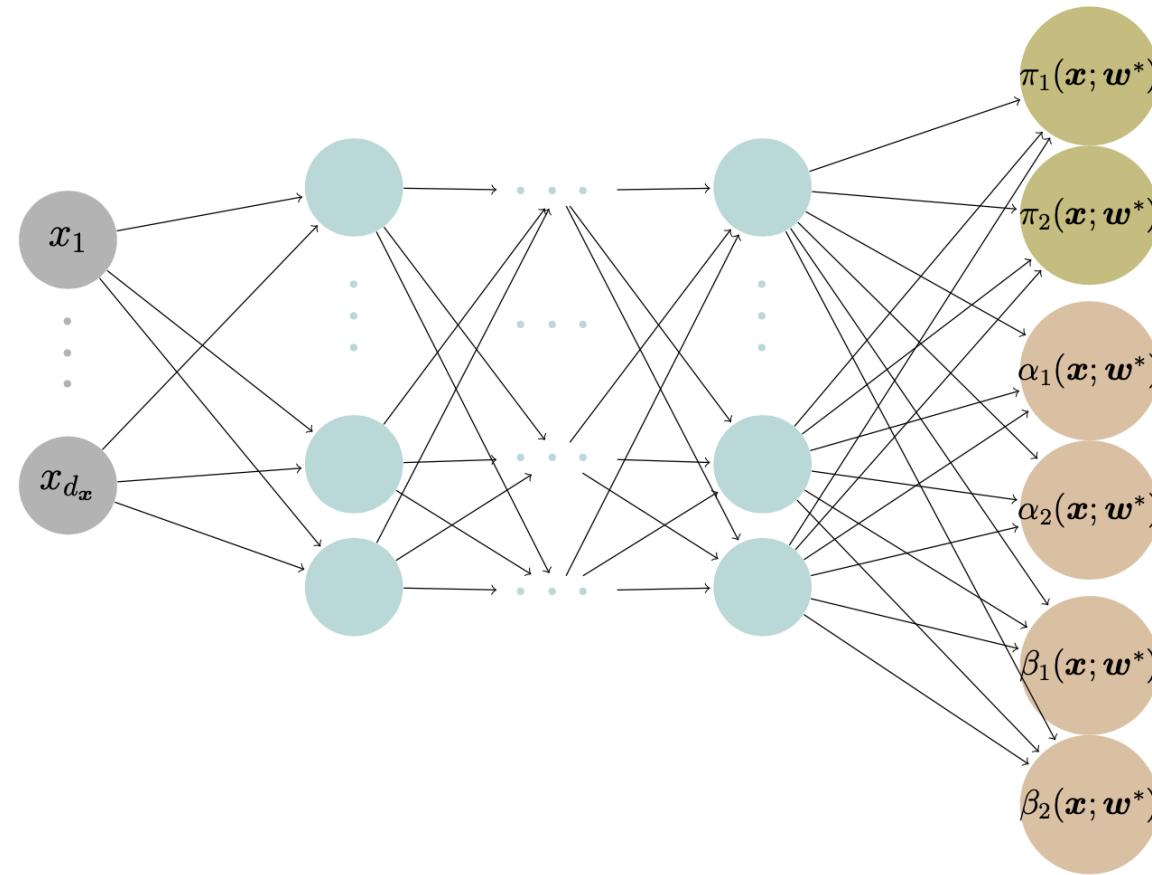


Figure: We demonstrate the structure of a gamma MDN that outputs the parameters for a gamma mixture with two components.



Code: Architecture

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

```
1 # Ensure reproducibility
2 random.seed(1); tf.random.set_seed(1)
3
4 inputs = Input(shape=X_train.shape[1:])
5
6 # Two hidden layers
7 x = Dense(64, activation='relu')(inputs)
8 x = Dense(64, activation='relu')(x)
9
10 pis = Dense(2, activation='softmax')(x) #mixing weights
11 alphas = Dense(2, activation='exponential')(x) #shape parameters
12 betas = Dense(2, activation='exponential')(x) #scale parameters
13
14 # `y_pred` will now have 6 columns
15 gamma_mdn = Model(inputs, Concatenate(axis=1)([pis, alphas, betas]))
```



LOSS Function

The negative log-likelihood loss function is given by

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

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

$$\begin{aligned} & \pi_1(\mathbf{x}; \mathbf{w}) \cdot \frac{\beta_1(\mathbf{x}; \mathbf{w})^{\alpha_1(\mathbf{x}; \mathbf{w})}}{\Gamma(\alpha_1(\mathbf{x}; \mathbf{w}))} e^{-\beta_1(\mathbf{x}; \mathbf{w})y} y^{\alpha_1(\mathbf{x}; \mathbf{w})-1} \\ & + (1 - \pi_1(\mathbf{x}; \mathbf{w})) \cdot \frac{\beta_2(\mathbf{x}; \mathbf{w})^{\alpha_2(\mathbf{x}; \mathbf{w})}}{\Gamma(\alpha_2(\mathbf{x}; \mathbf{w}))} e^{-\beta_2(\mathbf{x}; \mathbf{w})y} y^{\alpha_2(\mathbf{x}; \mathbf{w})-1} \end{aligned}$$



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.

```

1 import tensorflow_probability as tfp
2 tfd = tfp.distributions
3 K = 2 # number of mixture components
4
5 def gamma_mixture_NLL(y_true, y_pred):
6     K = y_pred.shape[1] // 3
7     pis = y_pred[:, :K]
8     alphas = y_pred[:, K:2*K]
9     betas = y_pred[:, 2*K:3*K]
10
11     # The mixture distribution is a MixtureSameFamily distribution
12     mixture_distribution = tfd.MixtureSameFamily(
13         mixture_distribution=tfd.Categorical(probs=pis),
14         components_distribution=tfd.Gamma(alphas, betas))
15
16     # The loss is the negative log-likelihood of the data
17     return -mixture_distribution.log_prob(y_true)

```



Code: Model Training

```
1 # Employ the loss function from previous slide
2 gamma_mdn.compile(optimizer="adam", loss=gamma_mixture_NLL)
3
4 hist = gamma_mdn.fit(X_train, y_train,
5     epochs=300,
6     callbacks=[EarlyStopping(patience=30)],
7     verbose=0,
8     batch_size=64,
9     validation_split=0.2)
```



Proper Scoring Rules

Definition

The scoring rule $S : \mathcal{F} \times \mathbb{R} \rightarrow \bar{\mathbb{R}}$ 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

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

where f is the predictive density.

Continuous Ranked Probability Score (CRPS)

The continuous ranked probability score is defined as

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

where F is the cumulative distribution function.



Code: NLL

```

1  from scipy.stats import gamma
2
3  def gamma_nll(mean, dispersion, y):
4      # Calculate shape and scale parameters from mean and dispersion
5      shape = 1 / dispersion; scale = mean * dispersion
6
7      # Create a gamma distribution object
8      gamma_dist = gamma(a=shape, scale=scale)
9
10     return -np.mean(gamma_dist.logpdf(y))
11
12 # GLM
13 X_test_design = sm.add_constant(X_test)
14 mus = gamma_GLM.predict(X_test_design)
15 NLL_GLM = gamma_nll(mus, phi_GLM, y_test)
16
17 # CANN
18 mus = np.exp(np.sum(CANN.predict(X_test, verbose=0), axis = 1))
19 NLL_CANN = gamma_nll(mus, phi_CANN, y_test)
20
21 # MDN
22 NLL_MDN = gamma_mdn.evaluate(X_test, y_test, verbose=0)

```



Model Comparisons

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

GLM: 11.02

CANN: 11.67

MDN: 8.88

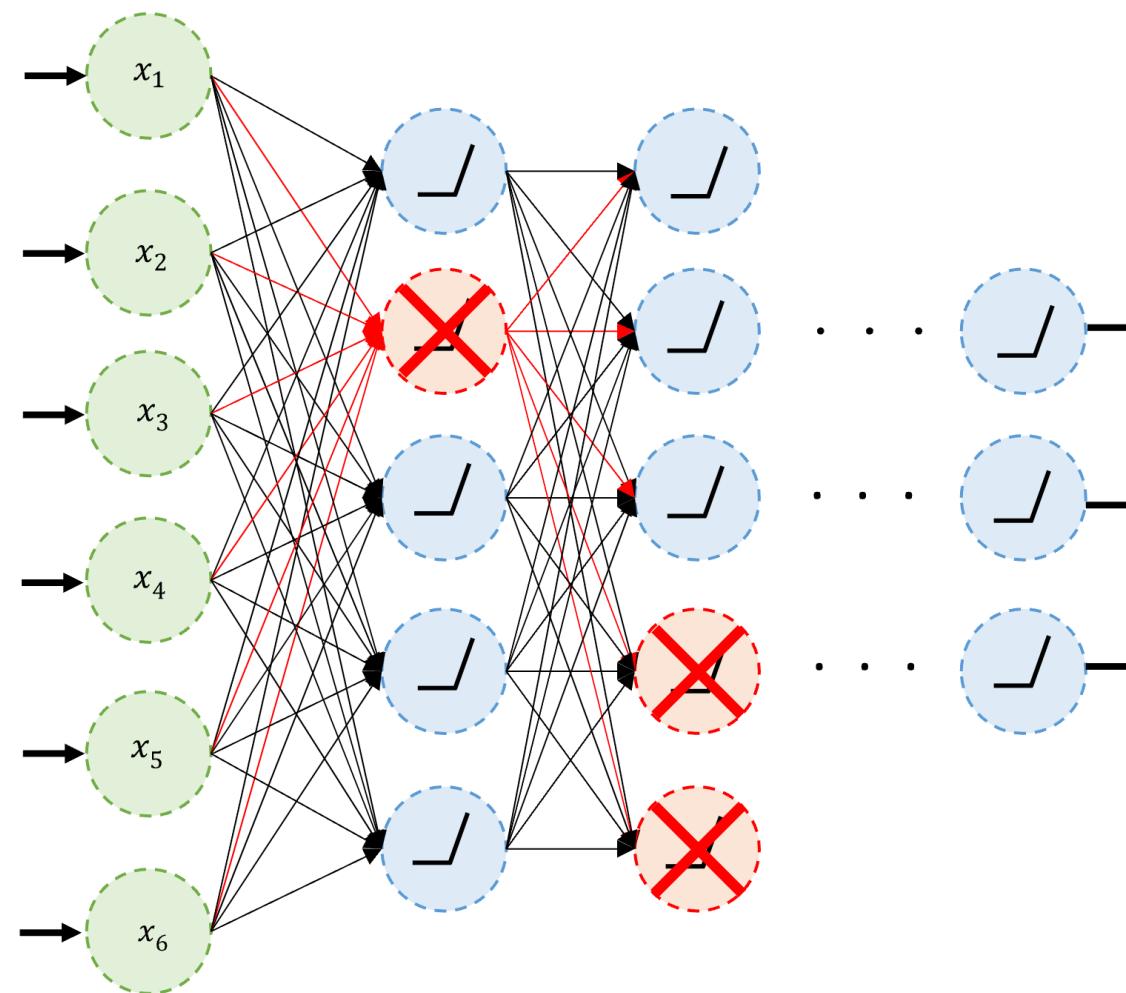


Lecture Outline

- Uncertainty
- Aleatoric Uncertainty
- **Epistemic Uncertainty**



Dropout



An example of neurons dropped during training.



Sources: Marcus Lautier (2022).



Dropout quote #1

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them.



Source: Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, p. 366

Dropout quote #2

The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.



Source: Aurélien Géron (2019), *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, p. 366

Code: Dropout

Dropout is just another layer in Keras.

```
1 from tf_keras.layers import Dropout
2 # Ensure reproducibility
3 random.seed(2); tf.random.set_seed(2)
4
5 model = Sequential([
6     Dense(30, activation="leaky_relu", name="hidden1"),
7     Dropout(0.2),
8     Dense(30, activation="leaky_relu", name="hidden2"),
9     Dropout(0.2),
10    Dense(1, activation="exponential", name="output")
11])
12
13 model.compile("adam", "mse")
14 model.fit(X_train, y_train, epochs=4, verbose=0);
```



Code: Dropout after training

Making predictions is the same as any other model:

```
1 model.predict(X_test.head(3),
                verbose=0)
```

```
array([[ 53.366356],
       [149.5078 ],
       [ 84.23206 ]], dtype=float32)
```

```
1 model.predict(X_test.head(3),
                verbose=0)
```

```
array([[ 53.366356],
       [149.5078 ],
       [ 84.23206 ]], dtype=float32)
```

We can make the model think it is still training:

```
1 model(X_test.head(3).to_numpy(),
        training=True).numpy()
```

```
array([[ 45.215103],
       [506.84497 ],
       [ 80.71608 ]], dtype=float32)
```

```
1 model(X_test.head(3).to_numpy(),
        training=True).numpy()
```

```
array([[170.88026],
       [140.37874],
       [231.01837]], dtype=float32)
```



Dropout Limitation

- Increased Training Time: Since dropout introduces noise into the training process, it can make the training process slower.
- Sensitivity to Dropout Rates: the performance of dropout is highly dependent on the chosen dropout rate.
- Uncertainty Quantification: the dropout can only provide a crude approximation to the theoretically justified Bayesian approach in terms of quantifying uncertainty.



Bayesian Neural Network

The weights \mathbf{w} of a Bayesian neural network (BNN) have their posterior distribution:

$$p(\mathbf{w}|\mathcal{D}) \propto \mathcal{L}(\mathcal{D}|\mathbf{w})p(\mathbf{w})$$

according to the Bayes' theorem.

- $\mathcal{L}(\mathcal{D}|\mathbf{w})$ represents the likelihood of data given the weights.
- $p(\mathbf{w})$ represents the density of the prior distribution of the weights.



Tractability of Posterior Distribution

Let $\boldsymbol{\theta}_0 = (\boldsymbol{\mu}_{\mathbf{w}_0}, \boldsymbol{\sigma}_{\mathbf{w}_0})$ be the parameters of the prior distribution of weights:

$$\mathbf{w} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{w}_0}, \boldsymbol{\sigma}_{\mathbf{w}_0}).$$

The derivation of the true posterior

$$p(\mathbf{w}|\mathcal{D}) \propto \mathcal{L}(\mathcal{D}|\mathbf{w})p(\mathbf{w})$$

is non-trivial due to the complexity of the model. We cannot compute the true posterior distribution efficiently.



Variational Approximation

The variational approximation is a potential solution. Intuitively, we approximate the true posterior distribution with a variational distribution that is more tractable:

$$\underbrace{p(\mathbf{w}|\mathcal{D})}_{\text{True Posterior Distribution}} \approx \underbrace{q(\mathbf{w}|\boldsymbol{\theta})}_{\text{Variational Distribution}} \sim \mathcal{N}(\boldsymbol{\mu}_w, \boldsymbol{\sigma}_w),$$

i.e., a normal distribution with parameters $\boldsymbol{\theta} = (\boldsymbol{\mu}_w, \boldsymbol{\sigma}_w)$ is used to approximate the true posterior distribution of $\mathbf{w}|\mathcal{D}$.



Demonstration

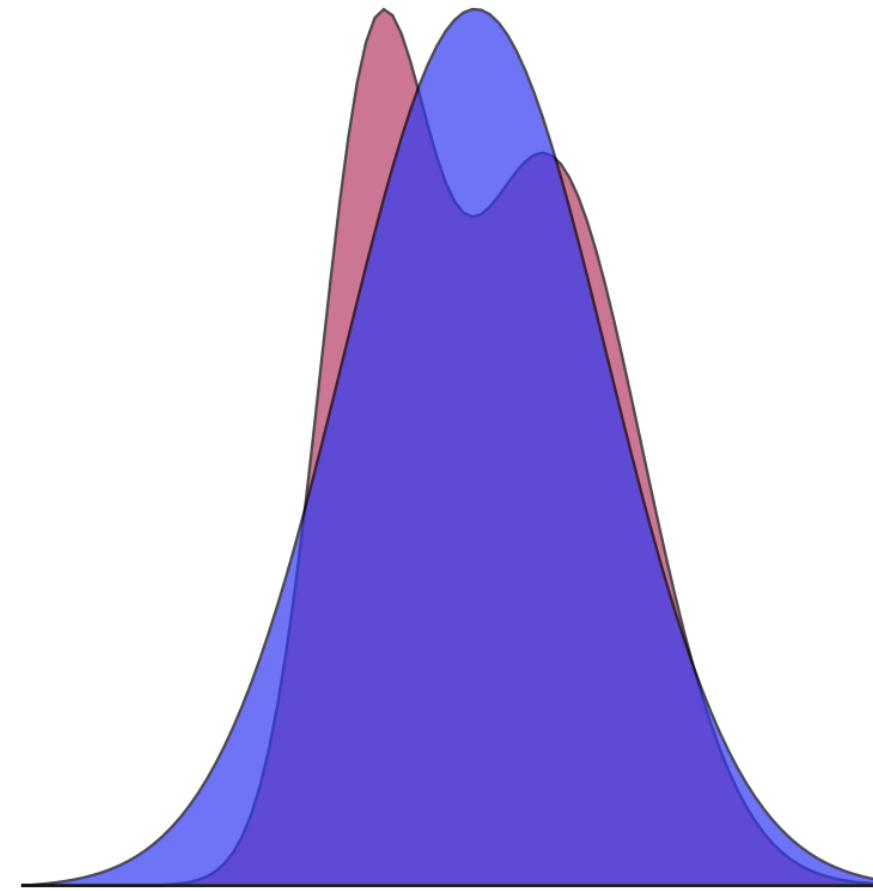


Figure: The idea is to use the **blue** curve (variational distribution) to approximate the **purple** curve (true posterior).



Code: Variational Layers

```
1 import tensorflow_probability as tfp
2 tfd = tfp.distributions
3
4 def prior(kernel_size, bias_size, dtype=None):
5     n = kernel_size + bias_size
6     return lambda t: tfd.Independent(
7         tfd.Normal(loc=tf.zeros(n, dtype=dtype),
8                    scale=1),
9         reinterpreted_batch_ndims=1)
10
11 def posterior(kernel_size, bias_size, dtype=None):
12     n = kernel_size + bias_size
13     return Sequential([
14         tfp.layers.VariableLayer(2 * n, dtype=dtype),
15         tfp.layers.DistributionLambda(lambda t: tfd.Independent(
16             tfd.Normal(loc=t[... , :n],
17                        scale=1e-5 + tf.nn.softplus(0.01 * t[... , n:])),
18             reinterpreted_batch_ndims=1)),
19     ])
```



Architecture

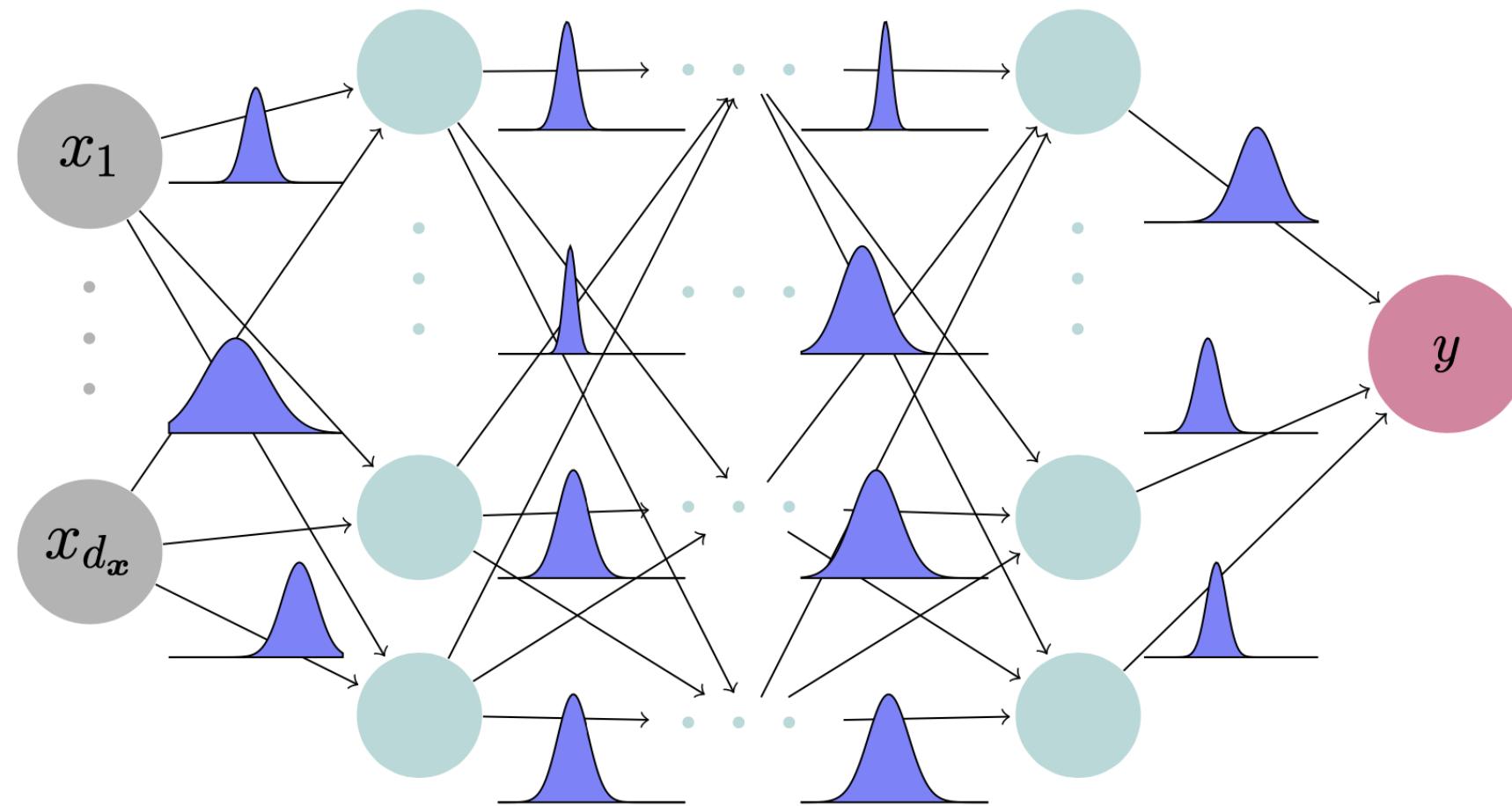


Figure: We demonstrate the typical structure of a Bayesian neural network (BNN).

Source: Blundell et al. (2015), Weight Uncertainty in Neural Networks.



LOSS Function

The KL divergence between the true posterior and variational distribution is given by:

$$D_{\text{KL}} [q(\mathbf{w}|\boldsymbol{\theta})||p(\mathbf{w}|\mathcal{D})] = \mathbb{E}_{\mathbf{w} \sim q(\mathbf{w}|\boldsymbol{\theta})} \left[\log \left(\frac{q(\mathbf{w}|\boldsymbol{\theta})}{p(\mathbf{w}|\mathcal{D})} \right) \right]$$

After some algebra, we acknowledge the final representation:

$$D_{\text{KL}} [q(\mathbf{w}|\boldsymbol{\theta})||p(\mathbf{w}|\mathcal{D})] = \underbrace{D_{\text{KL}} [q(\mathbf{w}|\boldsymbol{\theta})||p(\mathbf{w})]}_{\text{Complexity Loss}} - \underbrace{\mathbb{E}_{\mathbf{w} \sim q(\mathbf{w}|\boldsymbol{\theta})} [\log p(\mathcal{D}|\mathbf{w})]}_{\text{Error Loss}} + \text{const.}$$



Evaluation of Loss

In practice, we estimate loss function

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}) = \underbrace{D_{\text{KL}} [q(\mathbf{w}|\boldsymbol{\theta}) || p(\mathbf{w})]}_{\text{Complexity Loss}} - \underbrace{\mathbb{E}_{\mathbf{w} \sim q(\mathbf{w}|\boldsymbol{\theta})} [\log p(\mathcal{D}|\mathbf{w})]}_{\text{Error Loss}}$$

through Monte Carlo estimates

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}) \approx \frac{1}{M} \sum_{m=1}^M \underbrace{\log \left(\frac{q \left(\mathbf{w}^{(m)} | \boldsymbol{\theta}^{(m)} \right)}{p \left(\mathbf{w}^{(m)} \right)} \right)}_{\text{Complexity Loss}} - \underbrace{\log p \left(\mathcal{D} | \mathbf{w}^{(m)} \right)}_{\text{Error Loss}}$$

where $\{\mathbf{w}^{(m)}\}_{m=1}^M$ are random samples of $\mathbf{w}|\boldsymbol{\theta}$.



“Bayesian-Gamma” Loss

If the output consists of the shape and scale parameter of a gamma distribution, the loss function would be

$$\mathcal{L}(\mathcal{D}, \boldsymbol{\theta}) \approx \frac{1}{M} \sum_{m=1}^M \underbrace{\log \left(\frac{q(\mathbf{w}^{(m)} | \boldsymbol{\theta}^{(m)})}{p(\mathbf{w}^{(m)})} \right)}_{\text{Complexity Loss}} - \underbrace{\sum_{i=1}^N \log f(y_i | \mathbf{x}_i, \mathbf{w}^{(m)})}_{\text{Error Loss}},$$

where $f(y_i | \mathbf{x}_i, \mathbf{w}^{(m)})$ denotes the density value of y_i given \mathbf{x}_i , under the m th Monte Carlo sample $\mathbf{w}^{(m)}$, i.e.,

$$f(y_i | \mathbf{x}_i, \mathbf{w}^{(m)}) = \frac{\beta(\mathbf{x}; \mathbf{w}^{(m)})^{\alpha(\mathbf{x}; \mathbf{w}^{(m)})}}{\Gamma(\alpha(\mathbf{x}^{(m)}; \mathbf{w}^{(m)}))} e^{-\beta(\mathbf{x}; \mathbf{w}^{(m)})y_i} y_i^{\alpha(\mathbf{x}; \mathbf{w}^{(m)})-1}.$$



Architecture

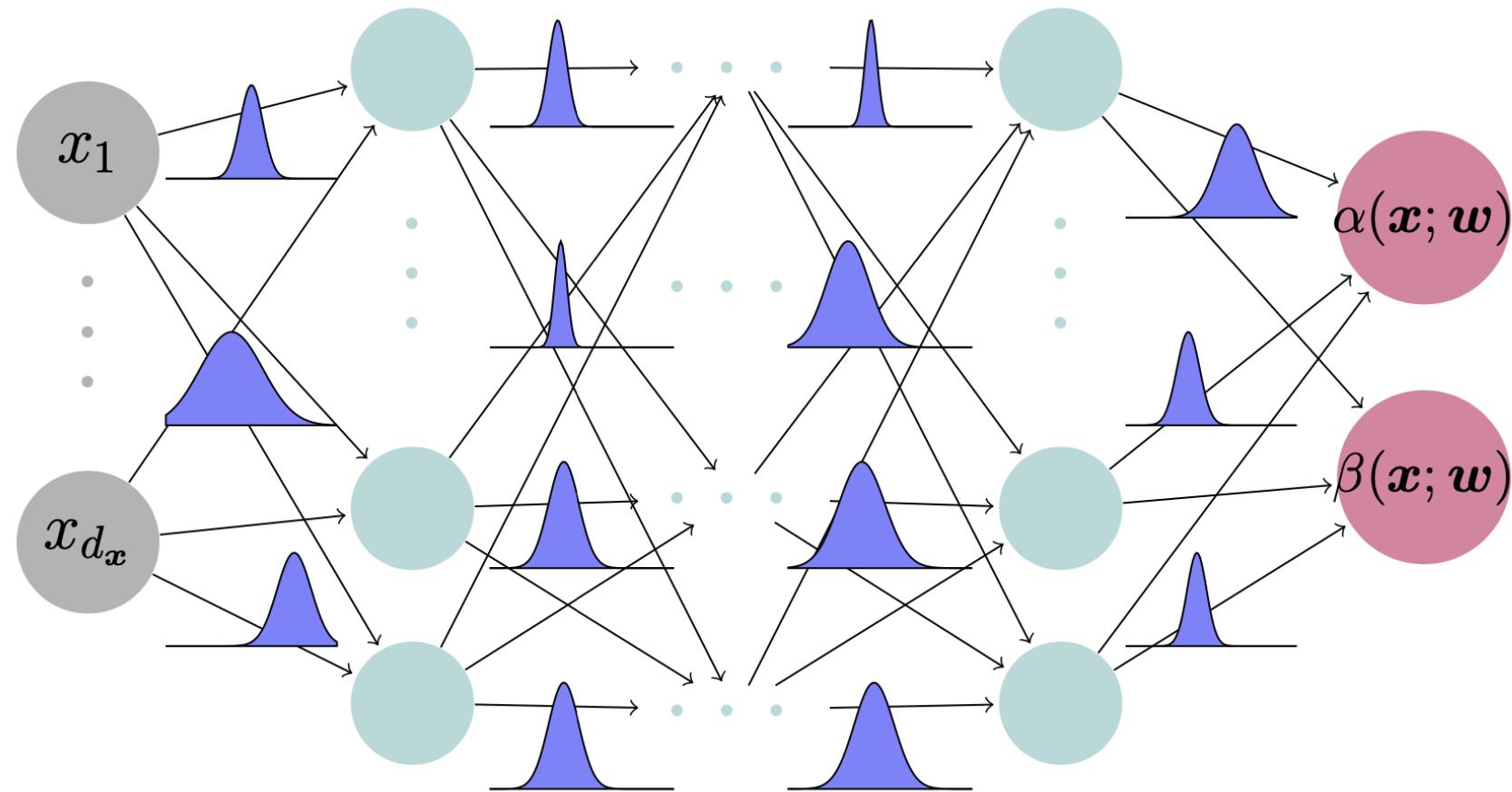


Figure: The output of our Bayesian neural network now consists of the shape parameter $\alpha(\mathbf{x}; \mathbf{w})$ and the scale parameter $\beta(\mathbf{x}; \mathbf{w})$.

Code: Architecture

The `tfp.layers` allows us to extract the parameters from the output, which is a gamma distribution object.

```
1 # Ensure reproducibility
2 random.seed(1); tf.random.set_seed(1)
3
4 inputs = Input(shape=X_train.shape[1:])
5
6 # DenseVariational layer
7 x = tfp.layers.DenseVariational(64, posterior, prior,
8                               kl_weight=1/X_train.shape[0])(inputs)
9 outputs = Dense(2, activation = 'softplus')(x)
10
11 # Construct the Gamma distribution on the last layer
12 distributions = tfp.layers.DistributionLambda(
13     lambda t: tfd.Gamma(concentration=t[ ..., 0:1],
14                           rate=t[ ..., 1:2]))(outputs)
15
16 # Define the model
17 gamma_bnn = Model(inputs, distributions)
```



Code: Loss Function and Training

```
1 def gamma_loss(y_true, y_pred):
2     return -y_pred.log_prob(y_true)
3
4 # Then use the loss function when compiling the model
5 gamma_bnn.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
6                     loss=gamma_loss)
7
8 hist = gamma_bnn.fit(X_train, y_train,
9                       epochs=300,
10                      callbacks=[EarlyStopping(patience=30)],
11                      verbose=0,
12                      batch_size=64,
13                      validation_split=0.2)
```



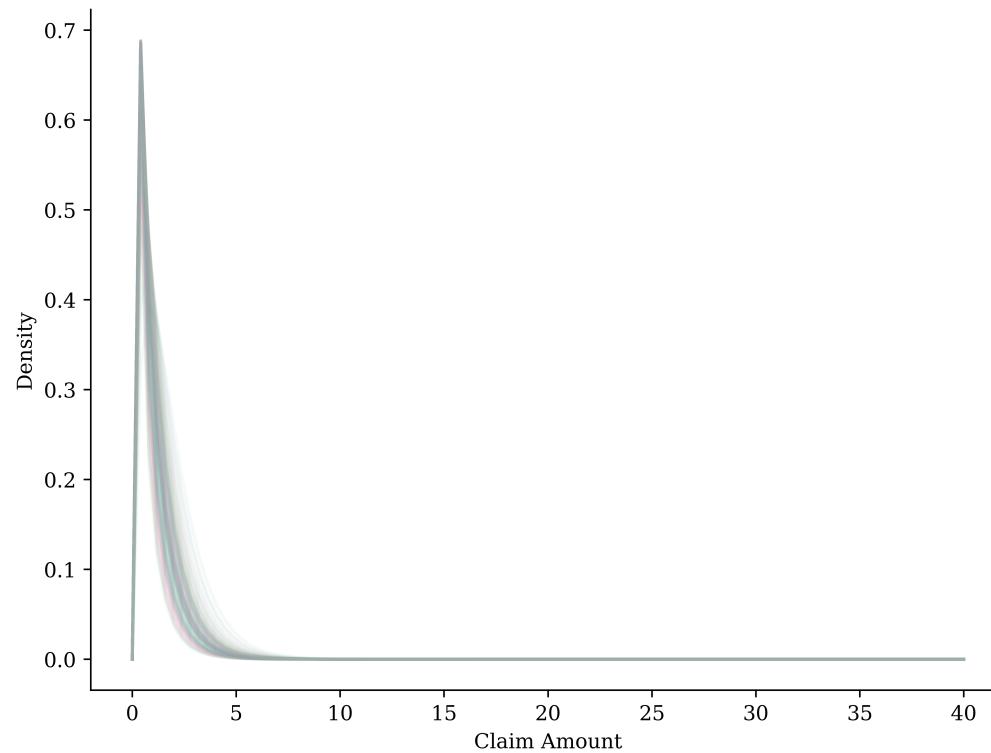
Code: Output Sampling

In practice, we can further increase the number of samples.

```
1 # Define the number of samples
2 n_samples = 1000
3
4 # Store all predictions in a list
5 alphas = []; betas = []
6
7 # Run the model `n_samples` times and store the predicted parameters
8 for i in range(n_samples):
9     # Predict the distributions
10    predicted_distributions = gamma_bnn(X_test[9:10].values)
11    # Get the parameters
12    alphas.append(predicted_distributions.concentration.numpy())
13    betas.append(predicted_distributions.rate.numpy())
```



Sampled Density Functions



We plot some of the sampled posterior density functions. The variability of the sampled density functions is one critical consideration for epistemic uncertainty.



Uncertainty Quantification (UQ)

We analyse the total variance formula:

$$\begin{aligned}
 \mathbb{V}[Y] &= \mathbb{E}[\mathbb{V}[Y|\boldsymbol{x}]] + \mathbb{V}[\mathbb{E}[Y|\boldsymbol{x}]] \\
 &\approx \underbrace{\frac{1}{M} \sum_{m=1}^M \mathbb{V}[Y|\boldsymbol{x}, \boldsymbol{w}^{(m)}]}_{\text{Aleatoric}} \\
 &\quad + \underbrace{\frac{1}{M} \sum_{m=1}^M \left(\mathbb{E}[Y|\boldsymbol{x}, \boldsymbol{w}^{(m)}] - \frac{1}{M} \sum_{m=1}^M \mathbb{E}[Y|\boldsymbol{x}, \boldsymbol{w}^{(m)}] \right)^2}_{\text{Epistemic}},
 \end{aligned}$$

where M is the number of posterior samples generated.



Code: Applying UQ

```
1 # Convert to numpy array for easier manipulation
2 alphas = np.array(alphas); betas = np.array(betas)
3
4 # Aleatoric uncertainty: Mean of the variances of the predicted Gamma distributions
5 aleatoric_uncertainty = np.mean(alphas/betas**2)
6
7 # Epistemic uncertainty: Variance of the means of the model's predictions
8 epistemic_uncertainty = np.var(alphas/betas)
9
10 print(f"Aleatoric uncertainty: {aleatoric_uncertainty}")
11 print(f"Epistemic uncertainty: {epistemic_uncertainty}")
```

Aleatoric uncertainty: 12227508.0
Epistemic uncertainty: 1425105.75



Deep Ensembles

Lakshminarayanan et al. (2017) proposed deep ensembles as another prominent approach to obtaining epistemic uncertainty. Such a technique can be an alternative to BNNs. It's simple to implement and requires very little hyperparameter tuning.

We summarise the deep ensemble approach for uncertainty quantification as follows:

1. Train D neural networks with different random weights initialisations independently in parallel. The trained weights are $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(D)}$.



Code: Deep Ensembles I

```

1 K = 1 # number of mixtures
2
3 def MDN_DE(num_ensembles):
4     models = []
5     for k in range(num_ensembles):
6         # Ensure reproducibility
7         random.seed(k); tf.random.set_seed(k)
8         inputs = Input(shape=X_train.shape[1:])
9
10        # Two hidden layers
11        x = Dense(64, activation='relu')(inputs)
12        x = Dense(64, activation='relu')(x)
13
14        pis = Dense(1, activation='softmax')(x) # mixing weights
15        alphas = Dense(1, activation='softplus')(x) # shape parameters
16        betas = Dense(1, activation='softplus')(x) # scale parameters
17
18        # Concatenate by columns: `y_pred` will now have 6 columns
19        gamma_mdn_new = Model(inputs, Concatenate(axis=1)([pis, alphas, betas]))
20        gamma_mdn_new.compile(optimizer="adam",
21                               loss=gamma_mixture_NLL)
22        gamma_mdn_new.fit(X_train, y_train,
23                            epochs=100, callbacks=[EarlyStopping(patience=10)],
24                            verbose=0, batch_size=64, validation_split=0.2)
25        models.append(gamma_mdn_new)
26
27 return(models)

```



Code: Deep Ensembles II

2. For a new instance \mathbf{x} , obtain

$$\left\{ \left(\mathbb{E}[Y|\mathbf{x}, \mathbf{w}^{(d)}], \mathbb{V}[Y|\mathbf{x}, \mathbf{w}^{(d)}] \right) \right\}_{d=1}^D,$$

```

1 D = 10 # number of MDNs
2 MDN_models = MDN_DE(D)
3
4 # Store all predictions in a list
5 weights = [0]*D; alphas = [0]*D; betas = [0]*D
6
7 # Store the parameters
8 for i in range(D):
9     weights[i], alphas[i], betas[i] = MDN_models[i].predict(X_test[9:10], verbose=0)[0]
10
11 # Predict the means and variances
12 means = np.array(alphas)/np.array(betas)
13 variances = np.array(alphas)/np.array(betas)**2

```



Code: Deep Ensembles III

3. Apply the variance decomposition

$$\mathbb{V}[Y] = \mathbb{E}[\mathbb{V}[Y|\mathbf{x}]] + \mathbb{V}[\mathbb{E}[Y|\mathbf{x}]]$$

```
1 aleatoric_uncertainty = np.mean(variances)
2 epistemic_uncertainty = np.var(means)
3
4 print(f"Aleatoric uncertainty: {aleatoric_uncertainty}")
5 print(f"Epistemic uncertainty: {epistemic_uncertainty}")
```

Aleatoric uncertainty: 71000552.0
Epistemic uncertainty: 16579237.0



Package Versions

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

```
Python implementation: CPython  
Python version      : 3.11.8  
IPython version     : 8.23.0  
  
keras                : 3.2.0  
matplotlib           : 3.8.4  
numpy                : 1.26.4  
pandas               : 2.2.1  
seaborn               : 0.13.2  
scipy                : 1.11.0  
torch                : 2.2.2  
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

