Uncertainty Quantification

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Eric Dong & Patrick Laub

Lecture Outline

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

UNSW SYDNEY

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

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

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 $\boldsymbol{x}$.

UNSW SYDNEY

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.

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 → 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	DrivAge	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

UNSW SYDNEY

Code: Preprocessing

```
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)
```

UNSW SYDNEY

Code: Preprocessing

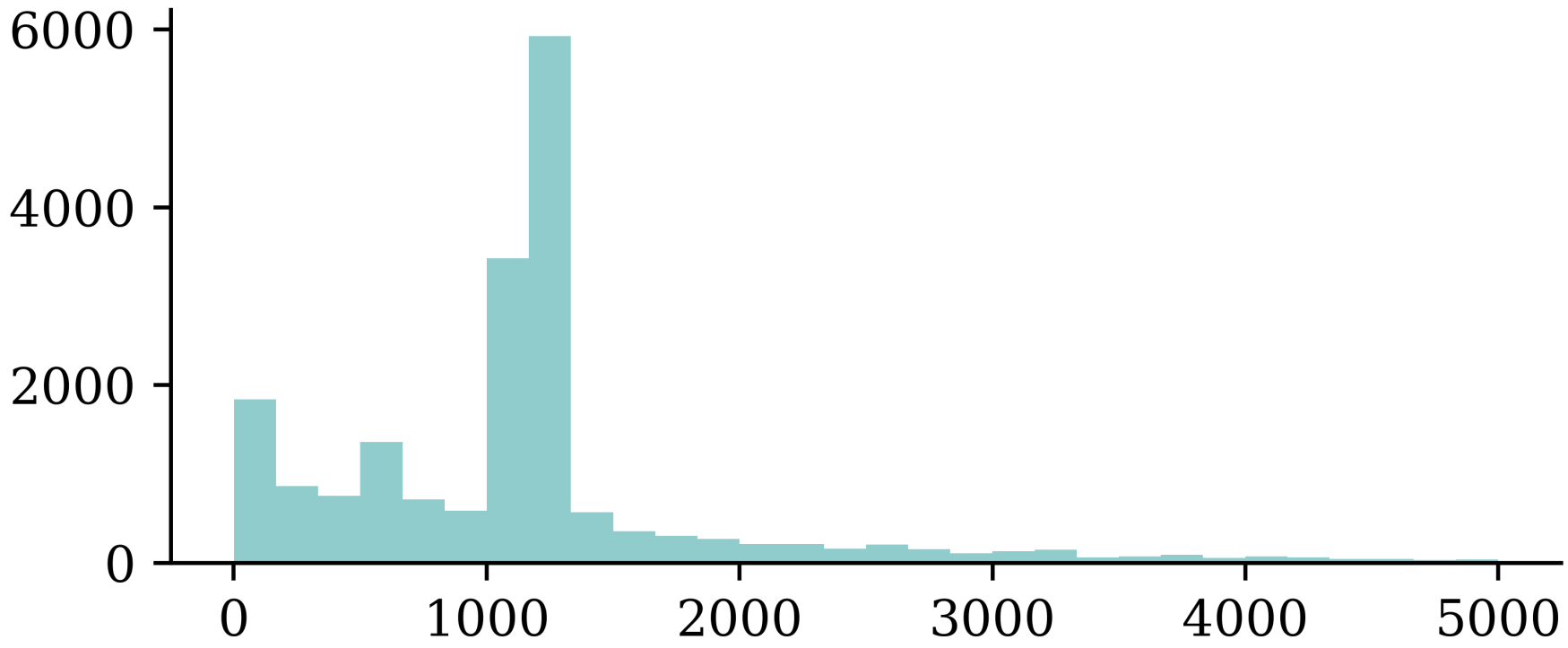
```
# Transformation
ct = make_column_transformer(
  (OrdinalEncoder(), ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
   verbose_feature_names_out=False
)

# 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`

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



Lecture Outline

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

UNSW SYDNEY

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:

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

where

- $\boldsymbol{x} \in \mathbb{R}^{d_{\boldsymbol{x}}}$ is the vector of explanatory variables, with $d_{\boldsymbol{x}}$ denoting its dimension.
- $\boldsymbol{\beta}_{\text{GLM}}$ 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:

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

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:

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

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.

UNSW SYDNEY

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

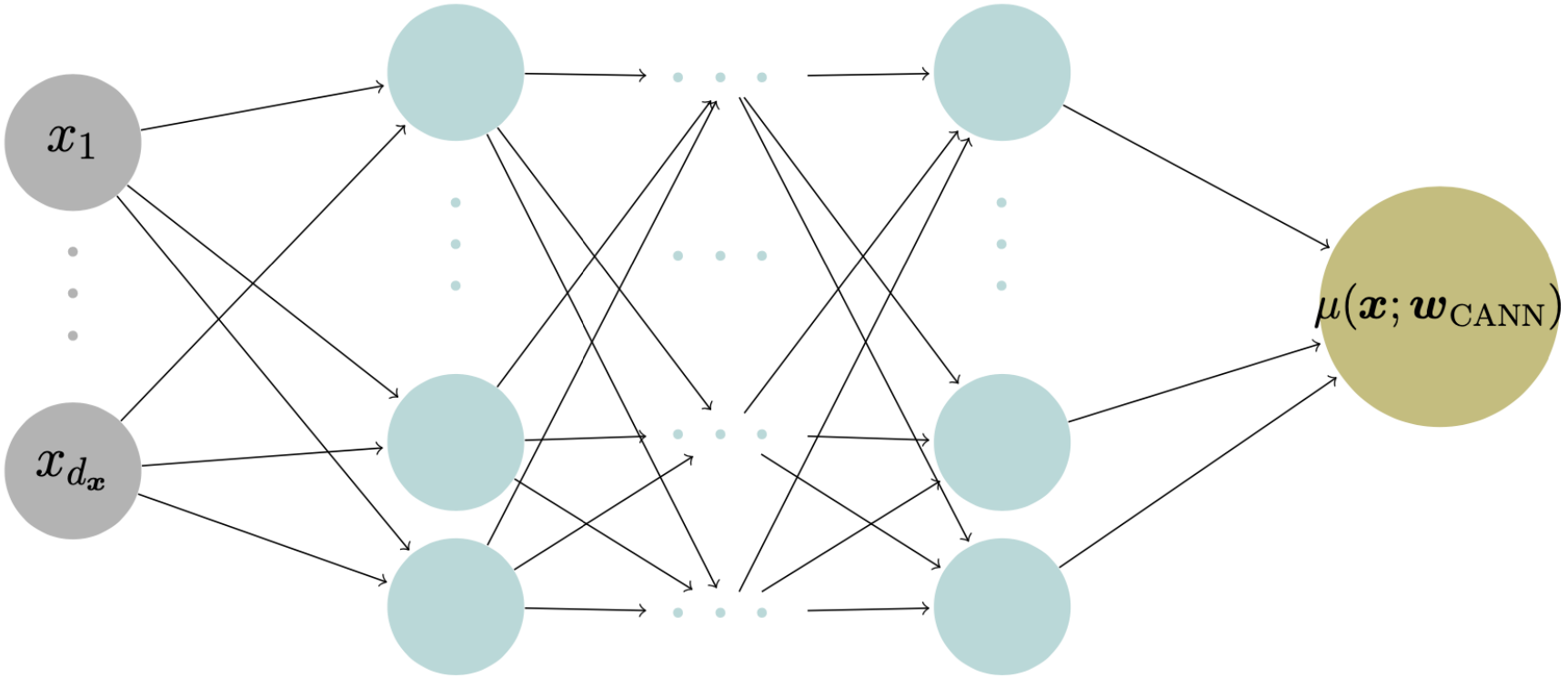


Figure: CANN approach.

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)
```

UNSW SYDNEY

Code: Loss Function

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

We need to customise the loss function for CANN.

```
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
```

UNSW SYDNEY

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
```

Mixture Distribution

Given a finite set of resulting random variables $(Y_1, ..., Y_K)$, one can generate a multinomial random variable $Y \sim \text{Multinomial}(1, \boldsymbol{\pi})$. Meanwhile, Y can be regarded as a mixture of $Y_1, ..., Y_K$, 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 $\boldsymbol{\pi} = (\pi_1..., \pi_K)$ such that $\pi_k \geq 0$ for $k \in \{1, ..., K\}$ 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

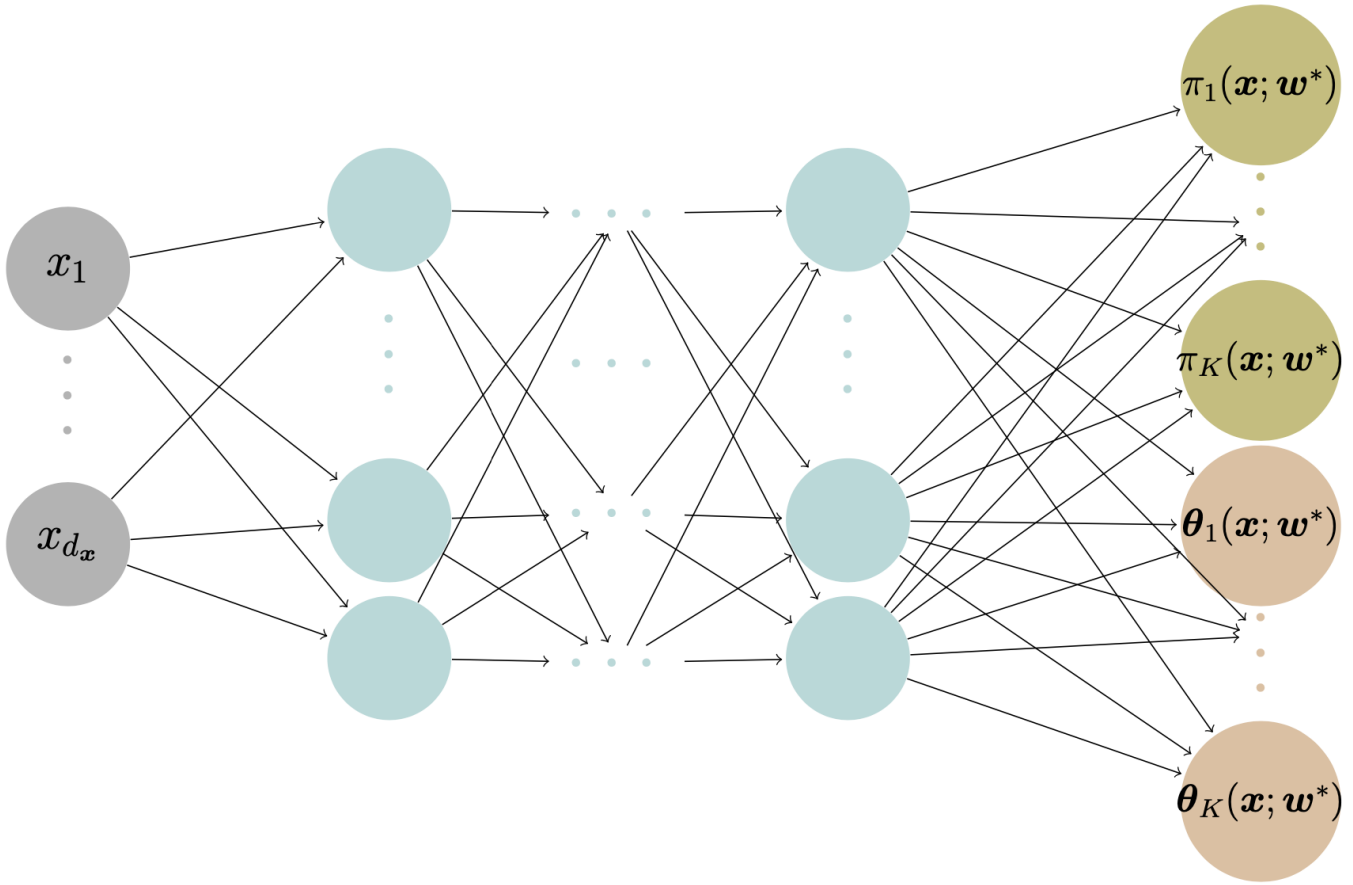


Figure: An MDN that outputs the parameters for a K component mixture distribution. $\boldsymbol{\theta}_k(\boldsymbol{x}; \boldsymbol{w}^*) = (\theta_{k,1}(\boldsymbol{x}; \boldsymbol{w}^*), ..., \theta_{k,|\boldsymbol{\theta}_k|}(\boldsymbol{x}; \boldsymbol{w}^*))$ consists of the parameter estimates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

UNSW SYDNEY

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

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

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

UNSW SYDNEY

Output

The aim is to find the optimum weights

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

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

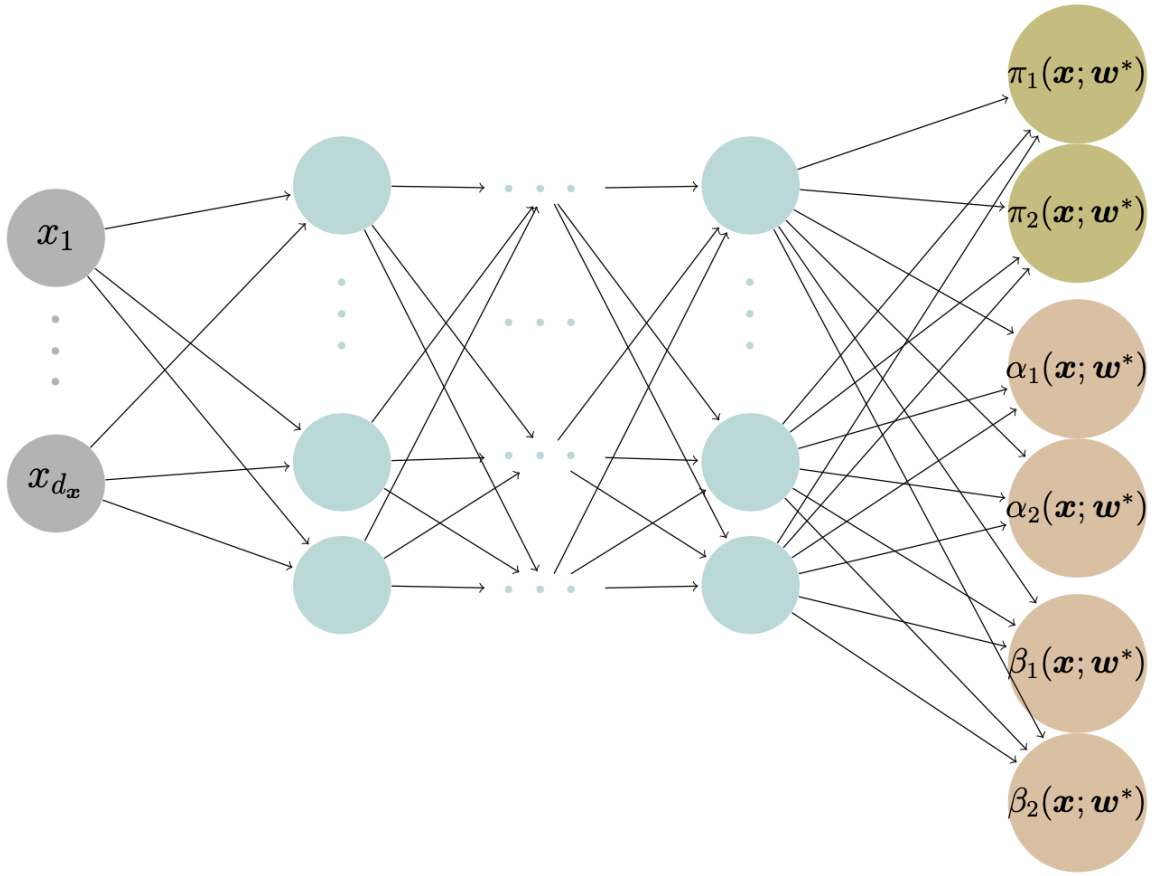


Figure: We demonstrate the structure of a gamma MDN that outputs the parameters for a gamma mixture with two components.

UNSW SYDNEY

Code: Architecture

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

```
# 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

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

UNSW SYDNEY

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)
```

UNSW SYDNEY

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)

UNSW SYDNEY

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

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

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

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

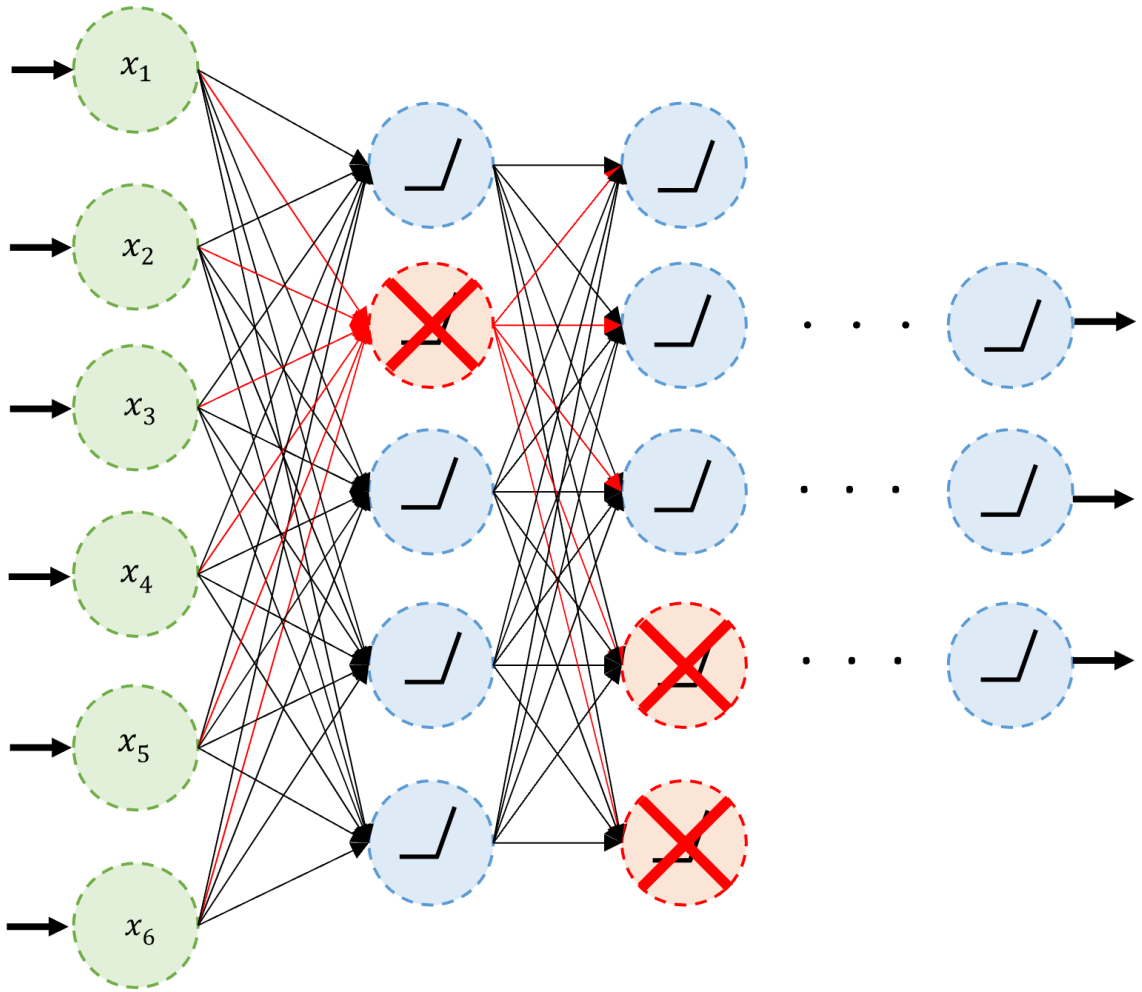
```
GLM: 11.02
CANN: 11.5
MDN: 8.67
```

Lecture Outline

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

UNSW SYDNEY

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.

```
from tf_keras.layers import Dropout
# Ensure reproducibility
random.seed(2); tf.random.set_seed(2)

model = Sequential([
    Dense(30, activation="leaky_relu", name="hidden1"),
    Dropout(0.2),
    Dense(30, activation="leaky_relu", name="hidden2"),
    Dropout(0.2),
    Dense(1, activation="exponential", name="output")
])

model.compile("adam", "mse")
model.fit(X_train, y_train, epochs=4, verbose=0);
```

Code: Dropout after training

Making predictions is the same as any other model:

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

```
array([[ 53.365997],
       [149.5073  ],
       [ 84.2315  ]], dtype=float32)
```

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

```
array([[ 53.365997],
       [149.5073  ],
       [ 84.2315  ]], dtype=float32)
```

We can make the model think it is still training:

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

```
array([[ 45.215286],
       [506.83798 ],
       [ 80.71608 ]], dtype=float32)
```

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

```
array([[170.87773],
       [140.37846],
       [231.01816]], dtype=float32)
```

UNSW SYDNEY

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 $\boldsymbol{w}$ of a Bayesian neural network (BNN) have their posterior distribution:

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

according to the Bayes' theorem.

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

Tractability of Posterior Distribution

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

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

The derivation of the true posterior

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

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:

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

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

Demonstration

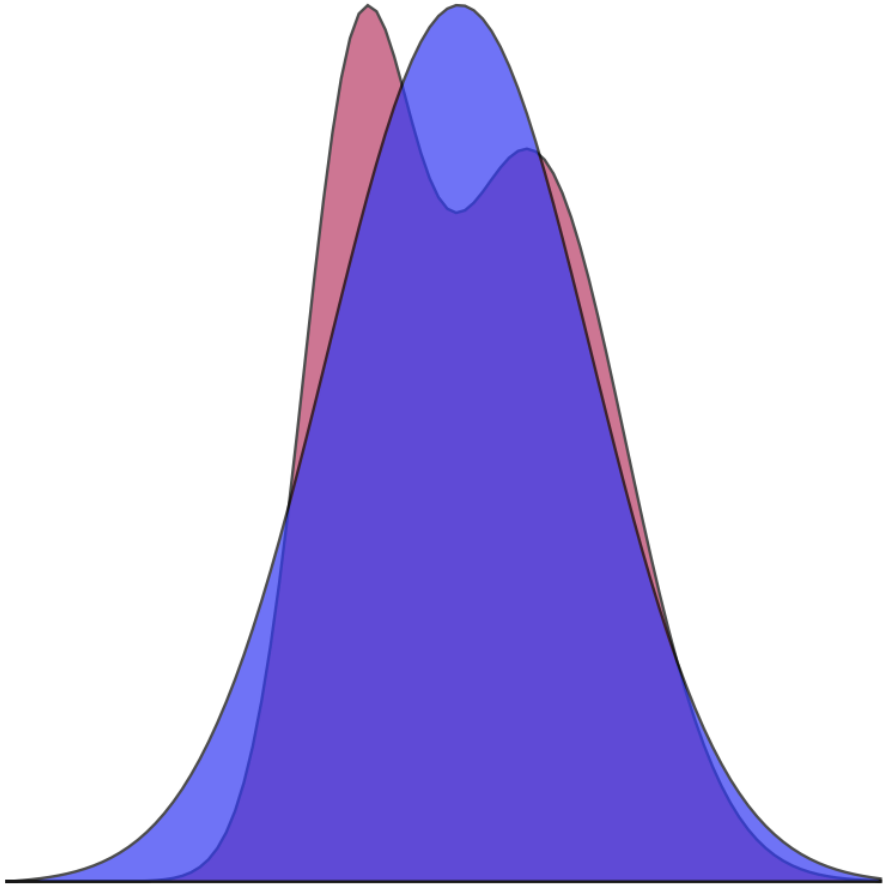


Figure: The idea is to use the blue curve (variational distribution) to approximate the purple curve (true posterior).

Code: Variational Layers

```python
import tensorflow_probability as tfp
tfd = tfp.distributions

def prior(kernel_size, bias_size, dtype=None):
    n = kernel_size + bias_size
    return lambda t: tfd.Independent(
        tfd.Normal(loc=tf.zeros(n, dtype=dtype),
                   scale=1),
                   reinterpreted_batch_ndims=1)

def posterior(kernel_size, bias_size, dtype=None):
    n = kernel_size + bias_size
    return Sequential([
      tfp.layers.VariableLayer(2 * n, dtype=dtype),
      tfp.layers.DistributionLambda(lambda t: tfd.Independent(
          tfd.Normal(loc=t[..., :n],
                     scale=1e-5 + tf.nn.softplus(0.01 * t[..., n:])),
          reinterpreted_batch_ndims=1)),
    ])
```

UNSW SYDNEY

Architecture

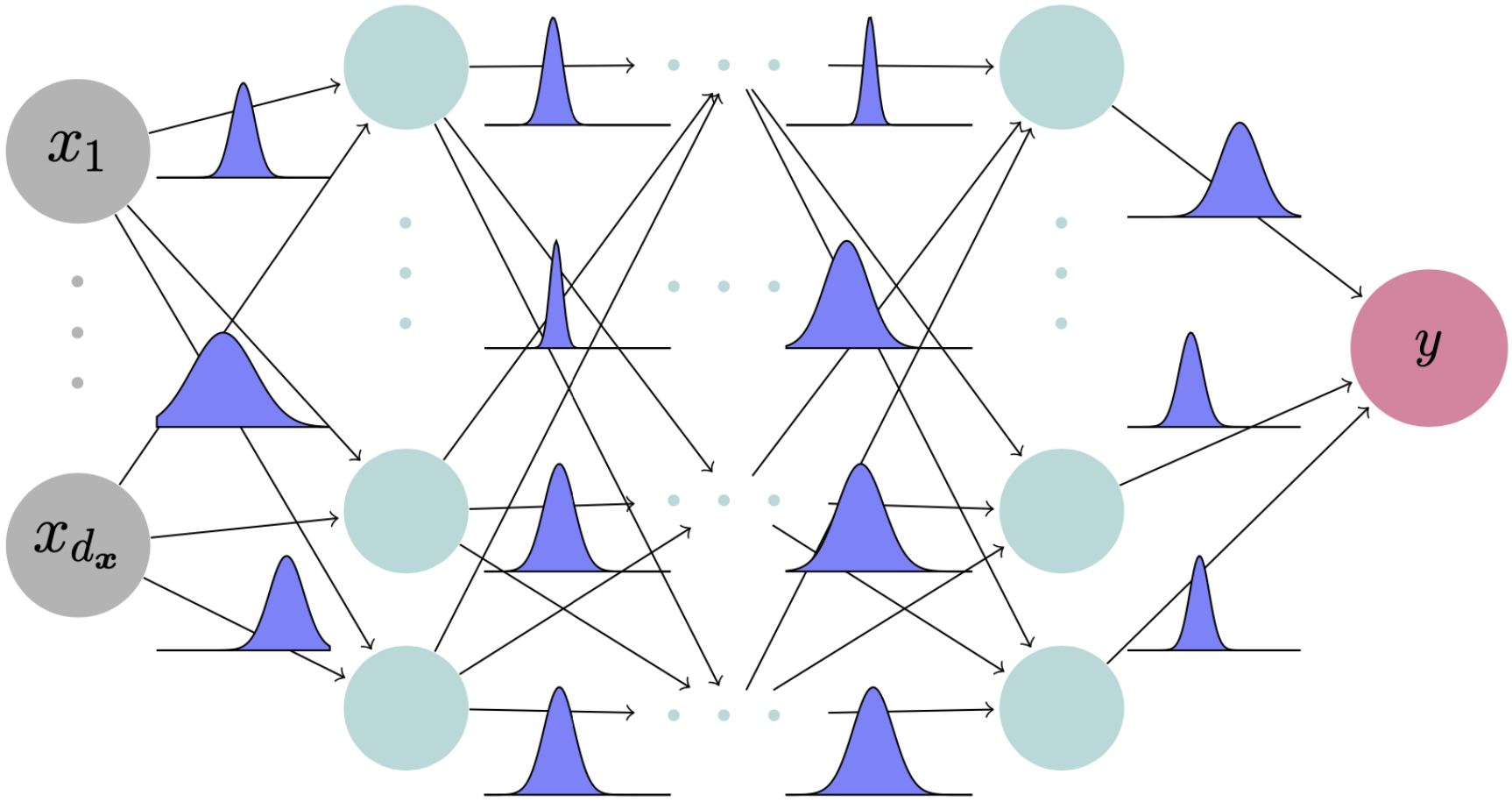


Figure: We demonstrate the typical structure of a Bayesian neural network (BNN).

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

UNSW SYDNEY

Loss Function

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

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

After some algebra, we acknowledge the final representation:

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

Evaluation of Loss

In practice, we estimate loss function

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

through Monte Carlo estimates

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

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

UNSW SYDNEY

"Bayesian-Gamma" Loss

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

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

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

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

UNSW SYDNEY

Architecture

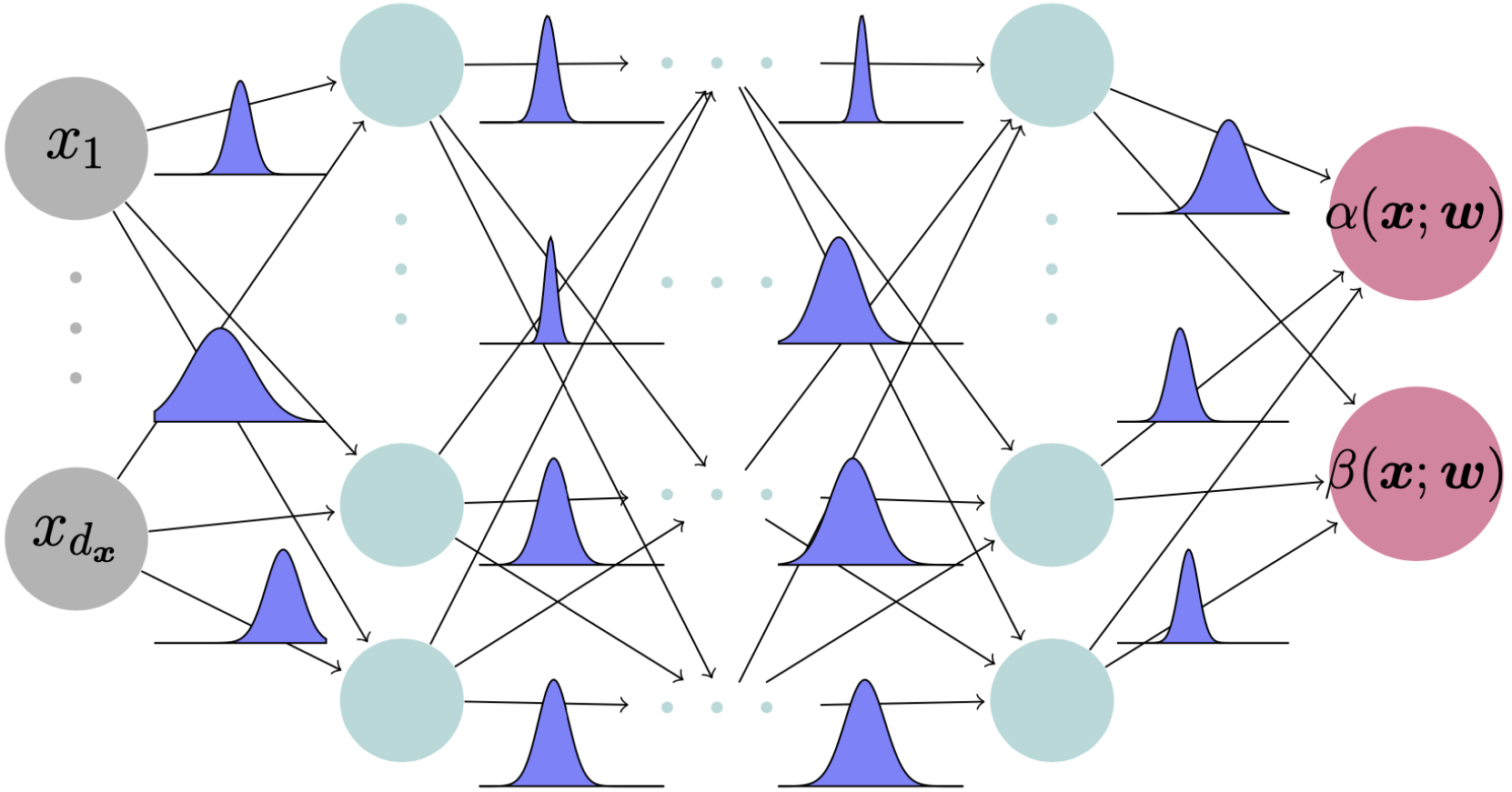


Figure: The output of our Bayesian neural network now consists of the shape parameter $\alpha(\boldsymbol{x}; \boldsymbol{w})$ and the scale parameter $\beta(\boldsymbol{x}; \boldsymbol{w})$.

UNSW SYDNEY

Code: Architecture

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

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

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

# DenseVariational layer
x = tfp.layers.DenseVariational(64, posterior, prior,
                kl_weight=1/X_train.shape[0])(inputs)
outputs = Dense(2, activation = 'softplus')(x)

# Construct the Gamma distribution on the last layer
distributions = tfp.layers.DistributionLambda(
      lambda t: tfd.Gamma(concentration=t[ ... , 0:1],
                          rate=t[ ... , 1:2]))(outputs)
# Define the model
gamma_bnn = Model(inputs, distributions)
```

UNSW SYDNEY

Code: Loss Function and Training

```python
def gamma_loss(y_true, y_pred):
    return -y_pred.log_prob(y_true)

# Then use the loss function when compiling the model
gamma_bnn.compile(optimizer=tf_keras.optimizers.Adam(learning_rate=0.001),
                  loss=gamma_loss)

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

Code: Output Sampling

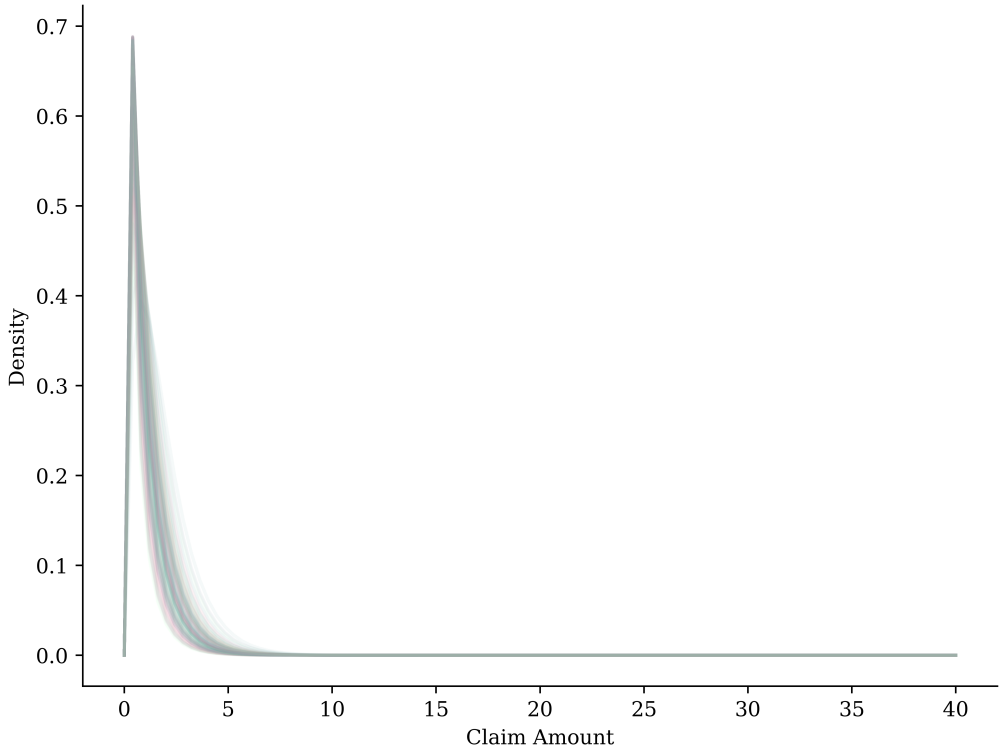
In practice, we can further increase the number of samples.

```
# Define the number of samples
n_samples = 1000

# Store all predictions in a list
alphas = []; betas = []

# Run the model `n_samples` times and store the predicted parameters
for i in range(n_samples):
  # Predict the distributions
  predicted_distributions = gamma_bnn(X_test[9:10].values)
  # Get the parameters
  alphas.append(predicted_distributions.concentration.numpy())
  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:

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

where M is the number of posterior samples generated.

UNSW SYDNEY

Code: Applying UQ

```python
# Convert to numpy array for easier manipulation
alphas = np.array(alphas); betas = np.array(betas)

# Aleatoric uncertainty: Mean of the variances of the predicted Gamma distributions
aleatoric_uncertainty = np.mean(alphas/betas**2)

# Epistemic uncertainty: Variance of the means of the model's predictions
epistemic_uncertainty = np.var(alphas/betas)

print(f"Aleatoric uncertainty: {aleatoric_uncertainty}")
print(f"Epistemic uncertainty: {epistemic_uncertainty}")
```

```
Aleatoric uncertainty: 12227515.0
Epistemic uncertainty: 1425106.75
```

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 $\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(D)}$.

Code: Deep Ensembles I

```
K = 1 # number of mixtures

def MDN_DE(num_ensembles):
  models = []
  for k in range(num_ensembles):
    # Ensure reproducibility
    random.seed(k); tf.random.set_seed(k)
    inputs = Input(shape=X_train.shape[1:])

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

    pis = Dense(1, activation='softmax')(x) # mixing weights
    alphas = Dense(1, activation='softplus')(x) # shape parameters
    betas = Dense(1, activation='softplus')(x) # scale parameters

    # Concatenate by columns: `y_pred` will now have 6 columns
    gamma_mdn_new = Model(inputs, Concatenate(axis=1)([pis, alphas, betas]))
    gamma_mdn_new.compile(optimizer="adam",
                          loss=gamma_mixture_NLL)
    gamma_mdn_new.fit(X_train, y_train,
        epochs=100, callbacks=[EarlyStopping(patience=10)],
        verbose=0, batch_size=64, validation_split=0.2)
    models.append(gamma_mdn_new)

  return(models)
```

UNSW SYDNEY

Code: Deep Ensembles II

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

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

```
D = 10 # number of MDNs
MDN_models = MDN_DE(D)

# Store all predictions in a list
weights = [0]*D; alphas = [0]*D; betas = [0]*D

# Store the parameters
for i in range(D):
  weights[i], alphas[i], betas[i] = MDN_models[i].predict(X_test[9:10], verbose=0)[0]

# Predict the means and variances
means = np.array(alphas)/np.array(betas)
variances = np.array(alphas)/np.array(betas)**2
```

UNSW SYDNEY

Code: Deep Ensembles III

3. Apply the variance decomposition

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

```python
aleatoric_uncertainty = np.mean(variances)
epistemic_uncertainty = np.var(means)

print(f"Aleatoric uncertainty: {aleatoric_uncertainty}")
print(f"Epistemic uncertainty: {epistemic_uncertainty}")
```

```
Aleatoric uncertainty: 75940600.0
Epistemic uncertainty: 16657899.0
```

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       : 3.11.9
IPython version      : 8.24.0

keras                : 3.3.3
matplotlib           : 3.8.4
numpy                : 1.26.4
pandas               : 2.2.2
seaborn              : 0.13.2
scipy                : 1.11.0
torch                : 2.0.1
tensorflow           : 2.16.1
tensorflow_probability: 0.24.0
tf_keras             : 2.16.0
```

UNSW SYDNEY

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