

# Distributional Regression

ACTL3143 & ACTL5111 Deep Learning for Actuaries  
Patrick Laub



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# Notation

- scalars are denoted by lowercase letters, e.g.,  $y$ ,
- vectors are denoted by bold lowercase letters, e.g.,

$$\mathbf{y} = (y_1, \dots, y_n),$$

- random variables are denoted by capital letters, e.g.,  $Y$
- random vectors are denoted by bold capital letters, e.g.,

$$\mathbf{X} = (X_1, \dots, X_p),$$

- matrices are denoted by bold uppercase non-italics letters, e.g.,

$$\mathbf{X} = \begin{pmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{np} \end{pmatrix}.$$



# Regression notation

- $n$  is the number of observations,  $p$  is the number of features,
- the true coefficients are  $\beta = (\beta_0, \beta_1, \dots, \beta_p)$ ,
- $\beta_0$  is the intercept,  $\beta_1, \dots, \beta_p$  are the coefficients,
- $\hat{\beta}$  is the estimated coefficient vector,
- $x_i = (1, x_{i1}, x_{i2}, \dots, x_{ip})$  is the feature vector for the  $i$ th observation,
- $y_i$  is the response variable for the  $i$ th observation,
- $\hat{y}_i$  is the predicted value for the  $i$ th observation,
- probability density functions (p.d.f.), probability mass functions (p.m.f.), cumulative distribution functions (c.d.f.).



# Traditional Regression

Multiple linear regression assumes the data-generating process is

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip} + \varepsilon$$

where  $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ .

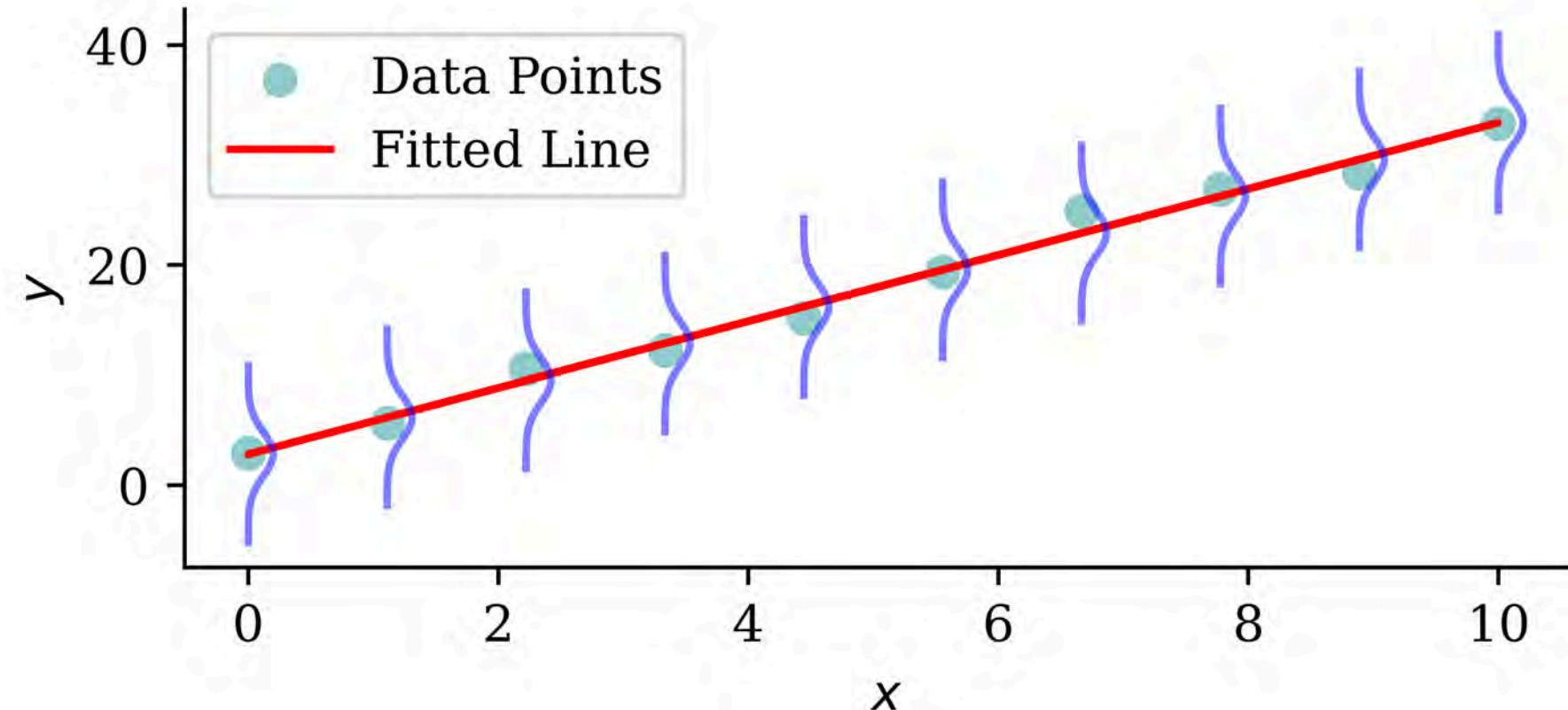
We estimate the coefficients  $\beta_0, \beta_1, \dots, \beta_p$  by minimising the sum of squared residuals or mean squared error

$$\text{RSS} := \sum_{i=1}^n (y_i - \hat{y}_i)^2, \quad \text{MSE} := \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2,$$

where  $\hat{y}_i$  is the predicted value for the  $i$ th observation.



# Visualising the distribution of each $Y$



# The probabilistic view

$$Y_i \sim \mathcal{N}(\mu_i, \sigma^2)$$

where  $\mu_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip}$ , and the  $\sigma^2$  is known.

The  $\mathcal{N}(\mu, \sigma^2)$  normal distribution has p.d.f.

$$f(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \mu)^2}{2\sigma^2}\right).$$

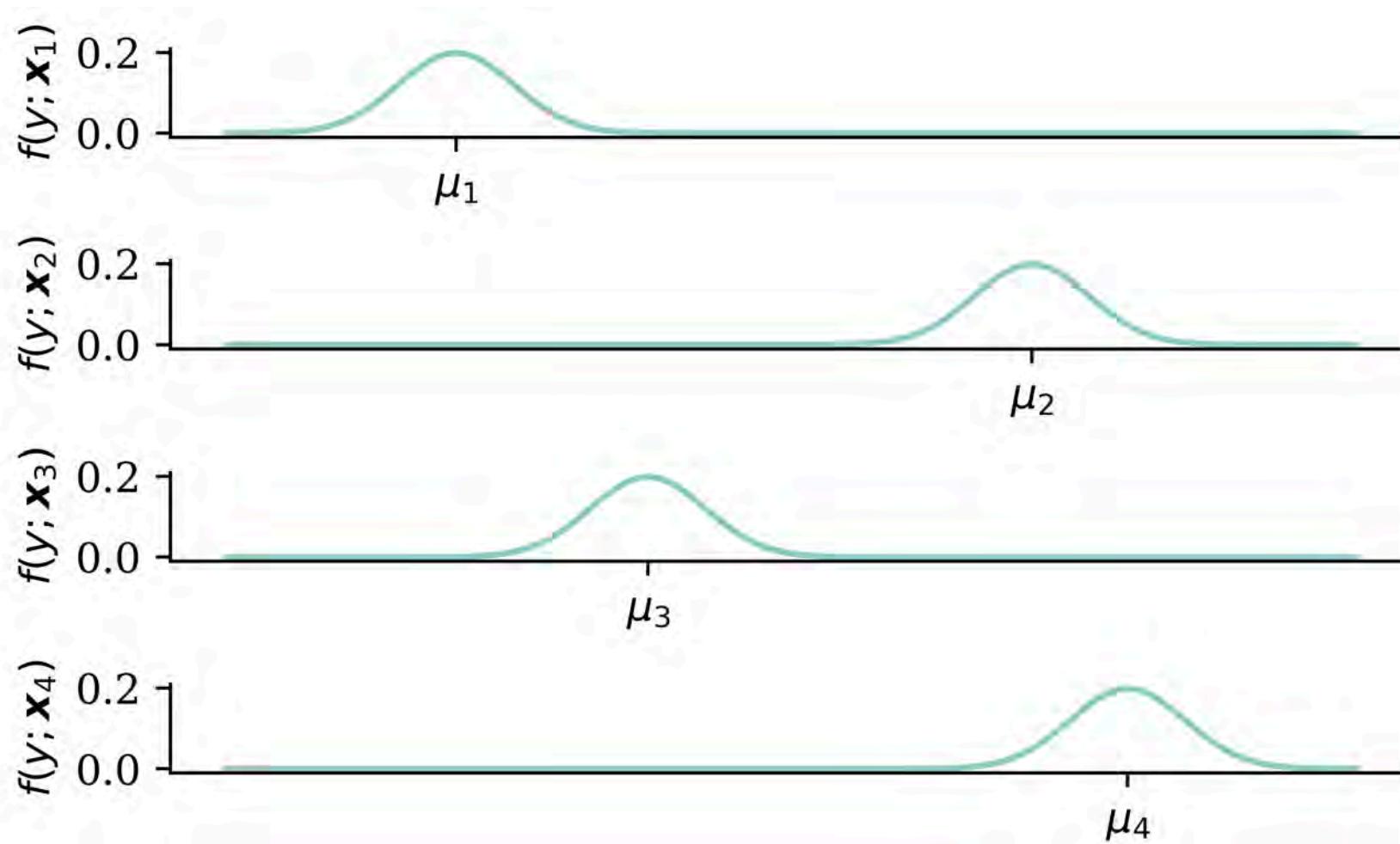
The likelihood function is

$$L(\boldsymbol{\beta}) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mu_i)^2}{2\sigma^2}\right)$$

$$\Rightarrow \ell(\boldsymbol{\beta}) = -\frac{n}{2} \log(2\pi) - \frac{n}{2} \log(\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \mu_i)^2.$$



# The predicted distributions



# The machine learning view

The negative log-likelihood  $\text{NLL}(\boldsymbol{\beta}) := -\ell(\boldsymbol{\beta})$  is to be minimised:

$$\text{NLL}(\boldsymbol{\beta}) = \frac{n}{2} \log(2\pi) + \frac{n}{2} \log(\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \mu_i)^2.$$

As  $\sigma^2$  is fixed, minimising NLL is equivalent to minimising MSE:

$$\begin{aligned}\widehat{\boldsymbol{\beta}} &= \arg \min_{\boldsymbol{\beta}} \text{NLL}(\boldsymbol{\beta}) \\ &= \arg \min_{\boldsymbol{\beta}} \frac{n}{2} \log(2\pi) + \frac{n}{2} \log(\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \mu_i)^2 \\ &= \arg \min_{\boldsymbol{\beta}} \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i(\mathbf{x}_i; \boldsymbol{\beta}))^2 \\ &= \arg \min_{\boldsymbol{\beta}} \text{MSE}(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{X}; \boldsymbol{\beta})).\end{aligned}$$



# Generalised Linear Model (GLM)

The GLM is often characterised by the mean prediction:

$$\mu(\boldsymbol{x}; \boldsymbol{\beta}) = g^{-1} (\langle \boldsymbol{\beta}, \boldsymbol{x} \rangle)$$

where  $g$  is the link function.

Common GLM distributions for the response variable include:

- Normal distribution with identity link (just MLR)
- Bernoulli distribution with logit link (logistic regression)
- Poisson distribution with log link (Poisson regression)
- Gamma distribution with log link



# Logistic regression

A Bernoulli distribution with parameter  $p$  has p.m.f.

$$f(y) = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases} = p^y(1 - p)^{1-y}.$$

Our model is  $Y|\mathbf{X} = \mathbf{x}$  follows a Bernoulli distribution with parameter

$$\mu(\mathbf{x}; \boldsymbol{\beta}) = \frac{1}{1 + \exp(-\langle \boldsymbol{\beta}, \mathbf{x} \rangle)} = \mathbb{P}(Y = 1 | \mathbf{X} = \mathbf{x}).$$

The likelihood function, using  $\mu_i := \mu(\mathbf{x}_i; \boldsymbol{\beta})$ , is

$$L(\boldsymbol{\beta}) = \prod_{i=1}^n \begin{cases} \mu_i & \text{if } y_i = 1 \\ 1 - \mu_i & \text{if } y_i = 0 \end{cases} = \prod_{i=1}^n \mu_i^{y_i} (1 - \mu_i)^{1-y_i}.$$



# Binary cross-entropy loss

$$L(\boldsymbol{\beta}) = \prod_{i=1}^n \mu_i^{y_i} (1 - \mu_i)^{1-y_i} \Rightarrow \ell(\boldsymbol{\beta}) = \sum_{i=1}^n \left( y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i) \right).$$

The negative log-likelihood is

$$\text{NLL}(\boldsymbol{\beta}) = - \sum_{i=1}^n \left( y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i) \right).$$

The binary cross-entropy loss is identical:

$$\text{BCE}(\mathbf{y}, \boldsymbol{\mu}) = - \sum_{i=1}^n \left( y_i \log(\mu_i) + (1 - y_i) \log(1 - \mu_i) \right).$$



# Poisson regression

A Poisson distribution with rate  $\lambda$  has p.m.f.

$$f(y) = \frac{\lambda^y \exp(-\lambda)}{y!}.$$

Our model is  $Y|\mathbf{X} = \mathbf{x}$  is Poisson distributed with parameter

$$\mu(\mathbf{x}; \boldsymbol{\beta}) = \exp(\langle \boldsymbol{\beta}, \mathbf{x} \rangle).$$

The likelihood function is

$$L(\boldsymbol{\beta}) = \prod_{i=1}^n \frac{\mu_i^{y_i} \exp(-\mu_i)}{y_i!}$$

$$\Rightarrow \ell(\boldsymbol{\beta}) = \sum_{i=1}^n \left( -\mu_i + y_i \log(\mu_i) - \log(y_i!) \right).$$



# Poisson loss

The negative log-likelihood is

$$\text{NLL}(\boldsymbol{\beta}) = \sum_{i=1}^n \left( \mu_i - y_i \log(\mu_i) + \log(y_i!) \right).$$

The Poisson loss is

$$\text{Poisson}(\mathbf{y}, \boldsymbol{\mu}) = \sum_{i=1}^n \left( \mu_i - y_i \log(\mu_i) \right).$$



# Gamma regression

A gamma distribution with mean  $\mu$  and dispersion  $\phi$  has p.d.f.

$$f(y; \mu, \phi) = \frac{(\mu\phi)^{-\frac{1}{\phi}}}{\Gamma\left(\frac{1}{\phi}\right)} y^{\frac{1}{\phi}-1} e^{-\frac{y}{\mu\phi}}$$

Our model is  $Y|\mathbf{X} = \mathbf{x}$  is gamma distributed with a dispersion of  $\phi$  and a mean of  $\mu(\mathbf{x}; \boldsymbol{\beta}) = \exp(\langle \boldsymbol{\beta}, \mathbf{x} \rangle)$ .

The likelihood function is

$$L(\boldsymbol{\beta}) = \prod_{i=1}^n \frac{(\mu_i\phi)^{-\frac{1}{\phi}}}{\Gamma\left(\frac{1}{\phi}\right)} y_i^{\frac{1}{\phi}-1} \exp\left(-\frac{y_i}{\mu_i\phi}\right)$$

$$\Rightarrow \ell(\boldsymbol{\beta}) = \sum_{i=1}^n \left[ -\frac{1}{\phi} \log(\mu_i\phi) - \log \Gamma\left(\frac{1}{\phi}\right) + \left(\frac{1}{\phi} - 1\right) \log(y_i) - \frac{y_i}{\mu_i\phi} \right].$$



# Gamma loss

The negative log-likelihood is

$$\text{NLL}(\boldsymbol{\beta}) = \sum_{i=1}^n \left[ \frac{1}{\phi} \log(\mu_i \phi) + \log \Gamma \left( \frac{1}{\phi} \right) - \left( \frac{1}{\phi} - 1 \right) \log(y_i) + \frac{y_i}{\mu_i \phi} \right].$$

Since  $\phi$  is a nuisance parameter

$$\text{NLL}(\boldsymbol{\beta}) = \sum_{i=1}^n \left[ \frac{1}{\phi} \log(\mu_i) + \frac{y_i}{\mu_i \phi} \right] + \text{const} \propto \sum_{i=1}^n \left[ \log(\mu_i) + \frac{y_i}{\mu_i} \right].$$

## Note

As  $\log(\mu_i) = \log(y_i) - \log(y_i/\mu_i)$ , we could write an alternative version

$$\text{NLL}(\boldsymbol{\beta}) \propto \sum_{i=1}^n \left[ \log(y_i) - \log\left(\frac{y_i}{\mu_i}\right) + \frac{y_i}{\mu_i} \right] \propto \sum_{i=1}^n \left[ \frac{y_i}{\mu_i} - \log\left(\frac{y_i}{\mu_i}\right) \right].$$



# Why do actuaries use GLMs?

- GLMs are interpretable.
- GLMs are flexible (can handle different types of response variables).
- We get the full distribution of the response variable, not just the mean.

This last point is particularly important for analysing worst-case scenarios.

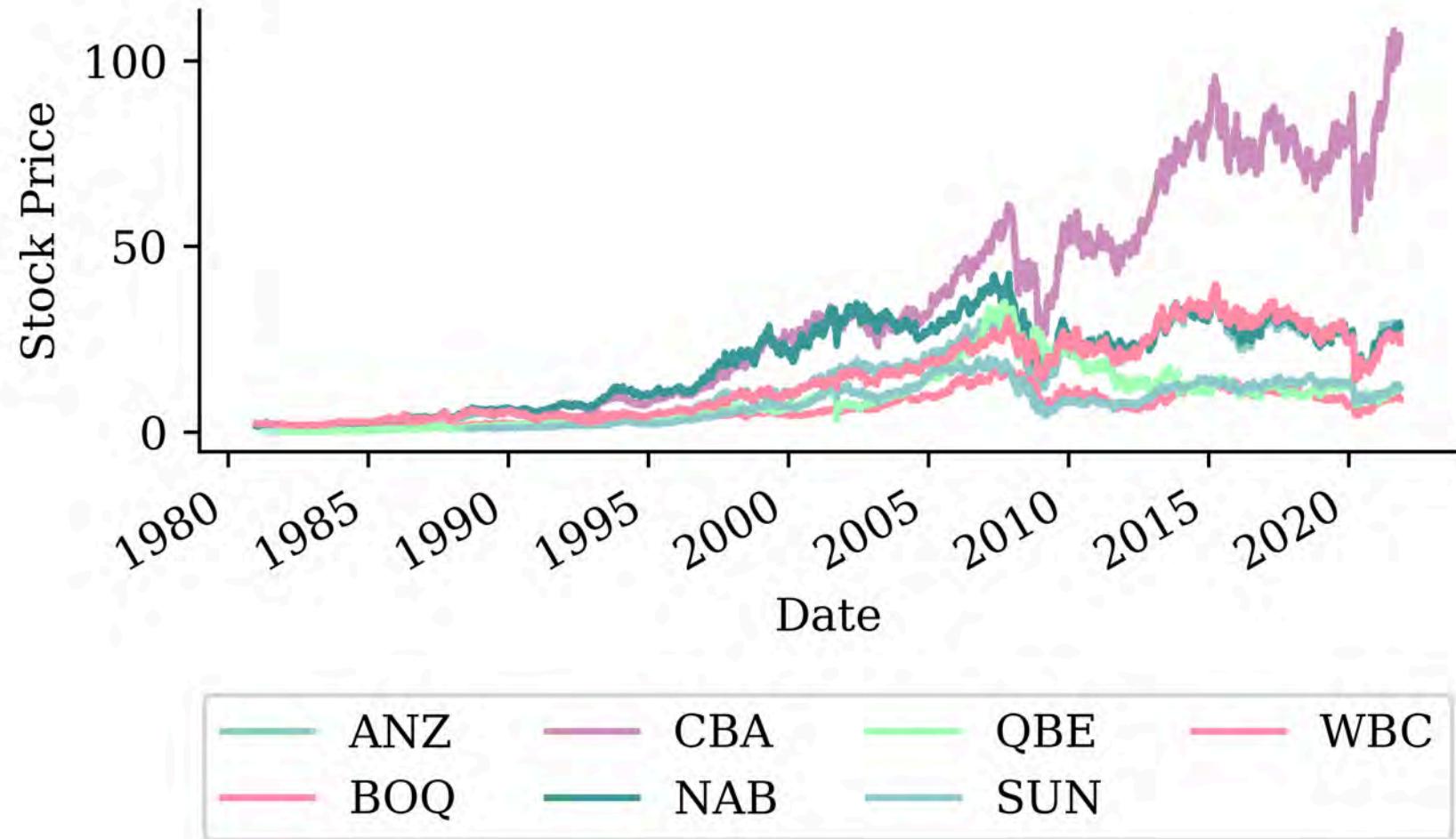


# Lecture Outline

- Traditional Regression
- **Stochastic Forecasts**
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# Stock price forecasting



# Noisy auto-regressive forecast

```

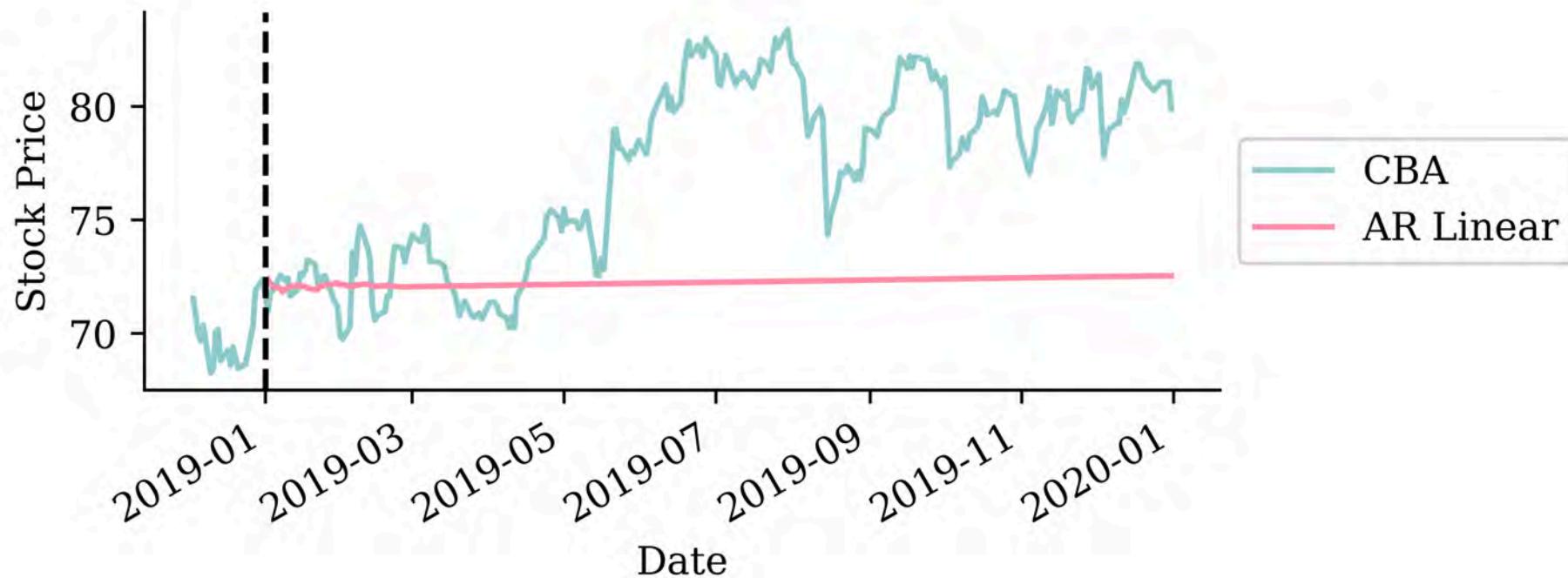
1 def noisy_autoregressive_forecast(model, X_val, sigma, suppress=False):
2     """
3         Generate a multi-step forecast using the given model.
4     """
5     multi_step = pd.Series(index=X_val.index, name="Multi Step")
6
7     # Initialize the input data for forecasting
8     input_data = X_val.iloc[0].values.reshape(1, -1)
9
10    for i in range(len(multi_step)):
11        # Ensure input_data has the correct feature names
12        input_df = pd.DataFrame(input_data, columns=X_val.columns)
13        if suppress:
14            next_value = model.predict(input_df, verbose=0)
15        else:
16            next_value = model.predict(input_df)
17
18            next_value += np.random.normal(0, sigma)
19
20            multi_step.iloc[i] = next_value
21
22            # Append that prediction to the input for the next forecast
23            if i + 1 < len(multi_step):
24                input_data = np.append(input_data[:, 1:], next_value).reshape(1, -1)
25
26    return multi_step

```



# Original forecast

```
1 lr_forecast = noisy_autoregressive_forecast(lr, X_val, 0)
```

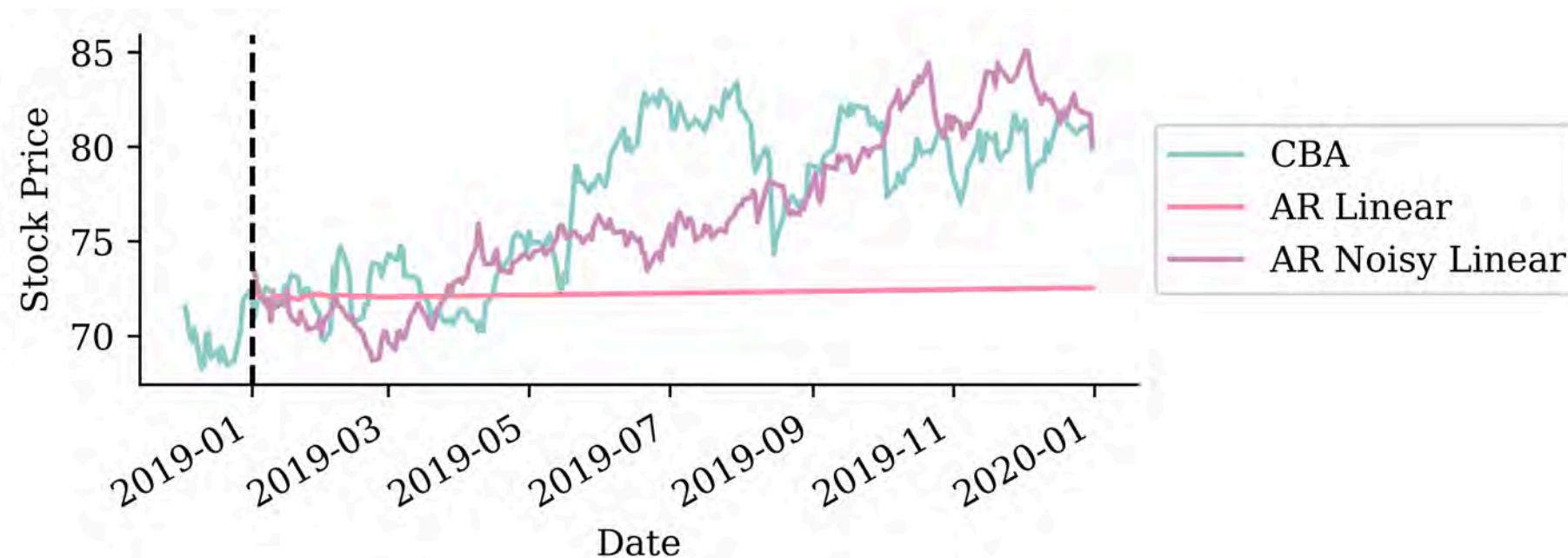


```
1 residuals = y_train - lr.predict(X_train)  
2 sigma = np.std(residuals)
```



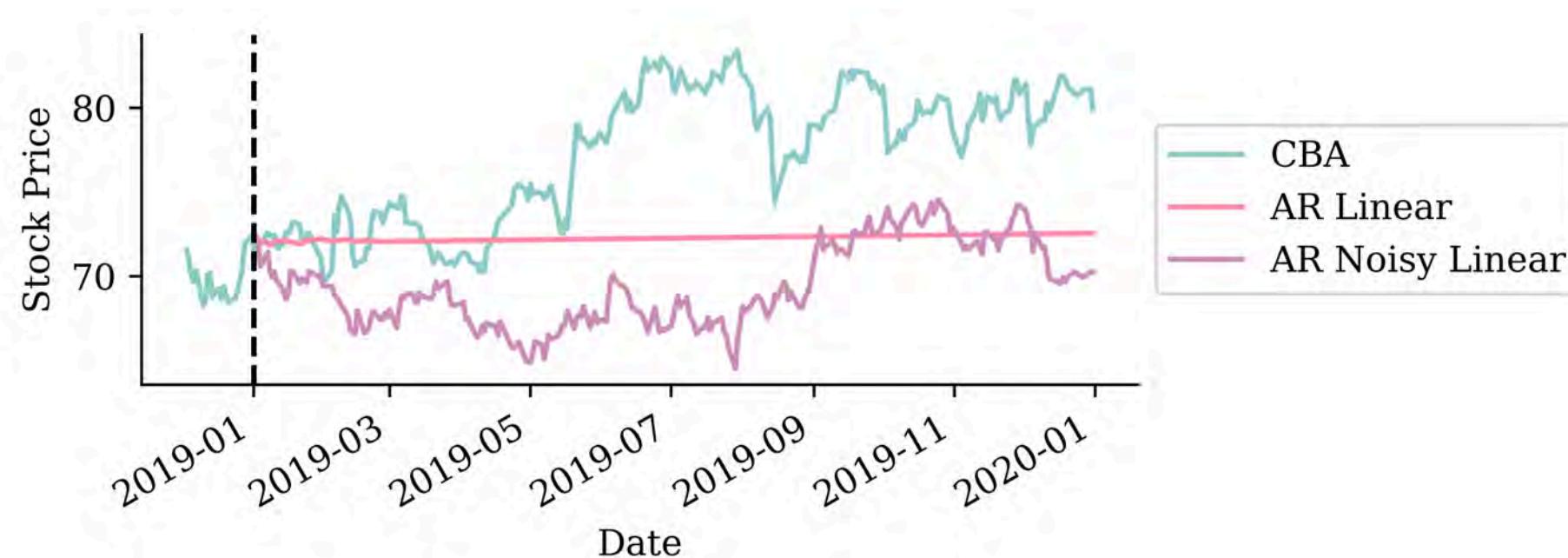
# With noise

```
1 np.random.seed(1)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_val, sigma)
```



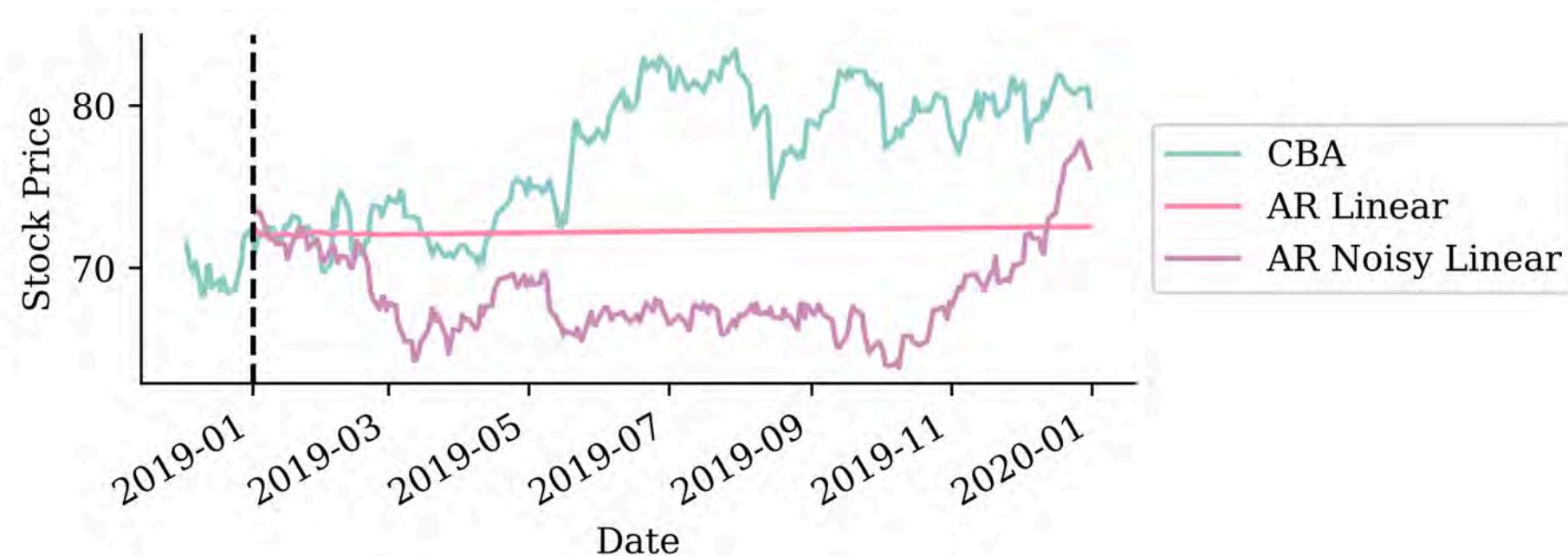
# With noise

```
1 np.random.seed(2)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_val, sigma)
```



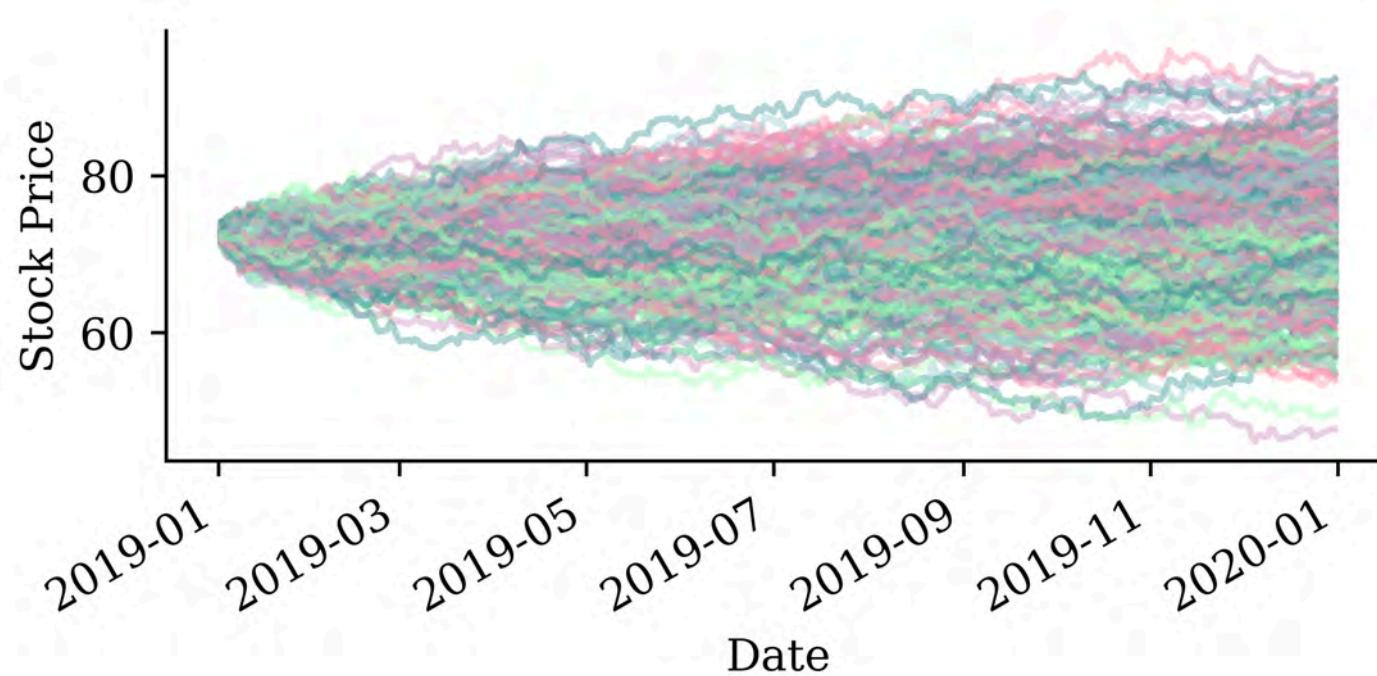
# With noise

```
1 np.random.seed(3)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_val, sigma)
```



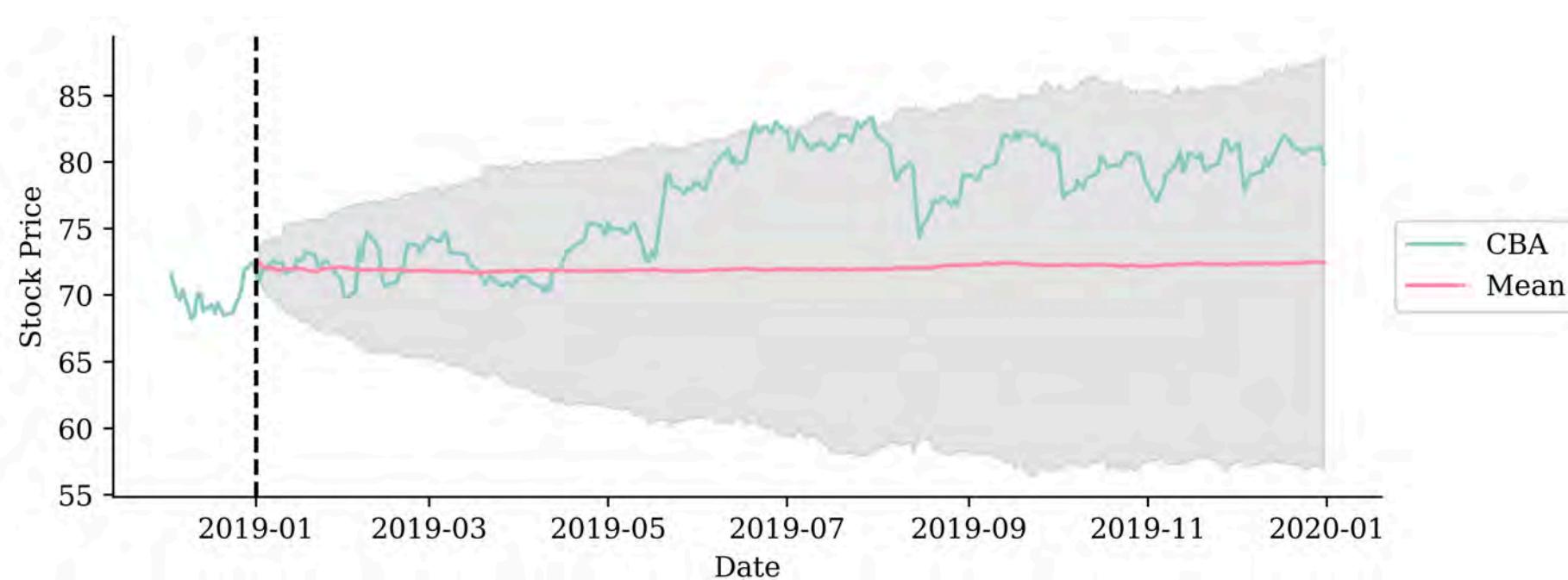
# Many noisy forecasts

```
1 num_forecasts = 500
2 forecasts = []
3 for i in range(num_forecasts):
4     forecasts.append(noisy_autoregressive_forecast(lr, X_val, sigma) * 100)
5 noisy_forecasts = pd.concat(forecasts, axis=1)
6 noisy_forecasts.index = X_val.index
```



# 95% “prediction intervals”

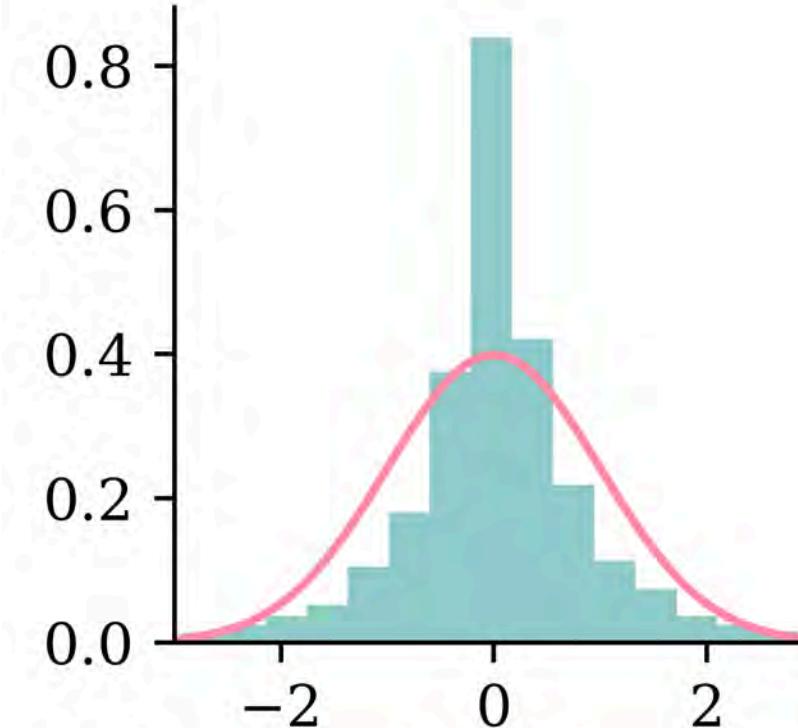
```
1 # Calculate quantiles for the forecasts
2 lower_quantile = noisy_forecasts.quantile(0.025, axis=1)
3 upper_quantile = noisy_forecasts.quantile(0.975, axis=1)
4 mean_forecast = noisy_forecasts.mean(axis=1)
```



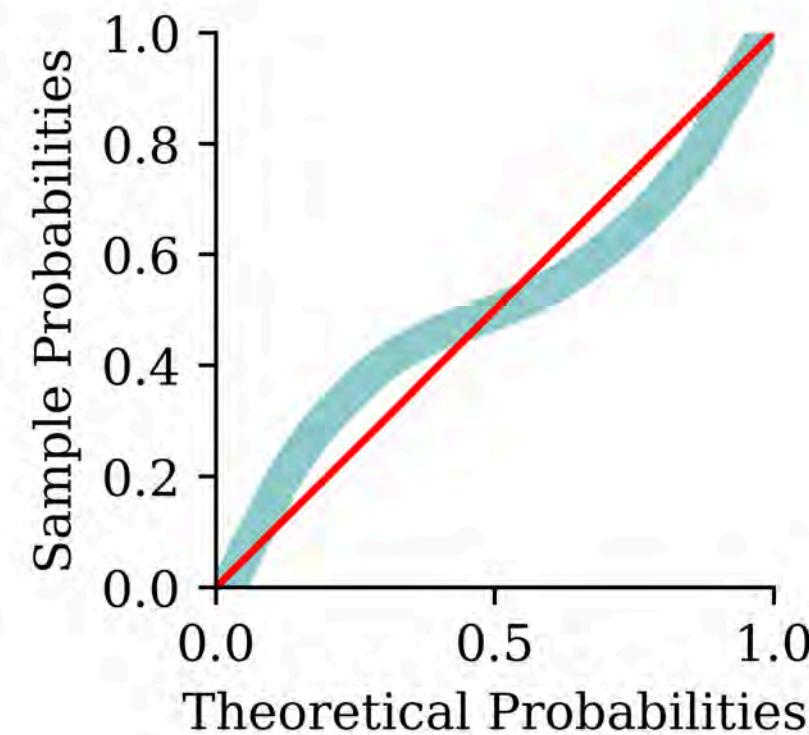
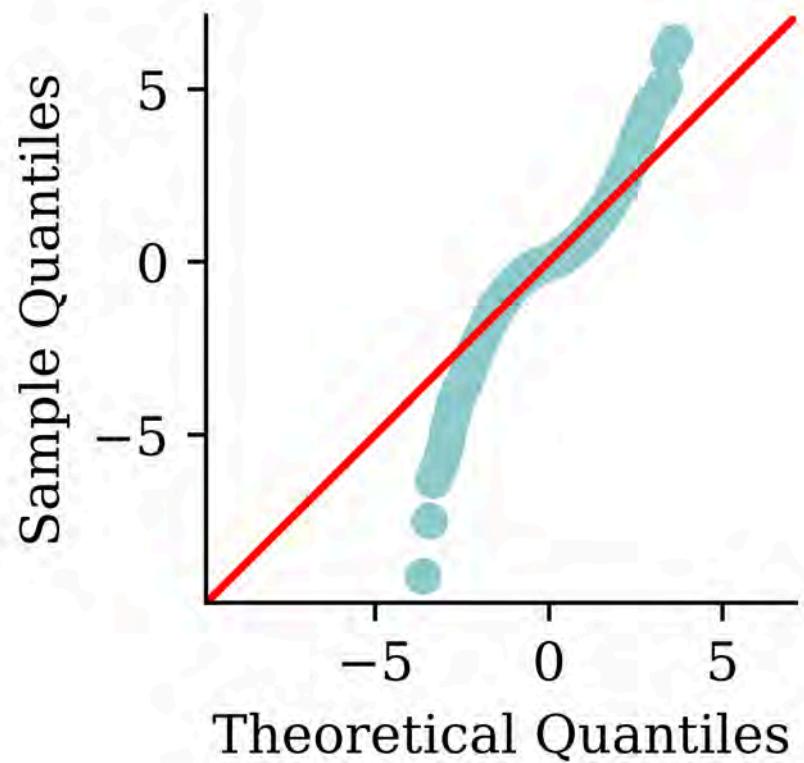
# Residuals

```
1 y_pred = lr.predict(X_train)
2 residuals = y_train - y_pred
3 residuals -= np.mean(residuals)
4 residuals /= np.std(residuals)
5 stats.shapiro(residuals)
```

```
ShapiroResult(statistic=0.9038059115409851,
pvalue=0.0)
```



# Q-Q plot and P-P plot



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- **GLMs and Neural Networks**
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# 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



# 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

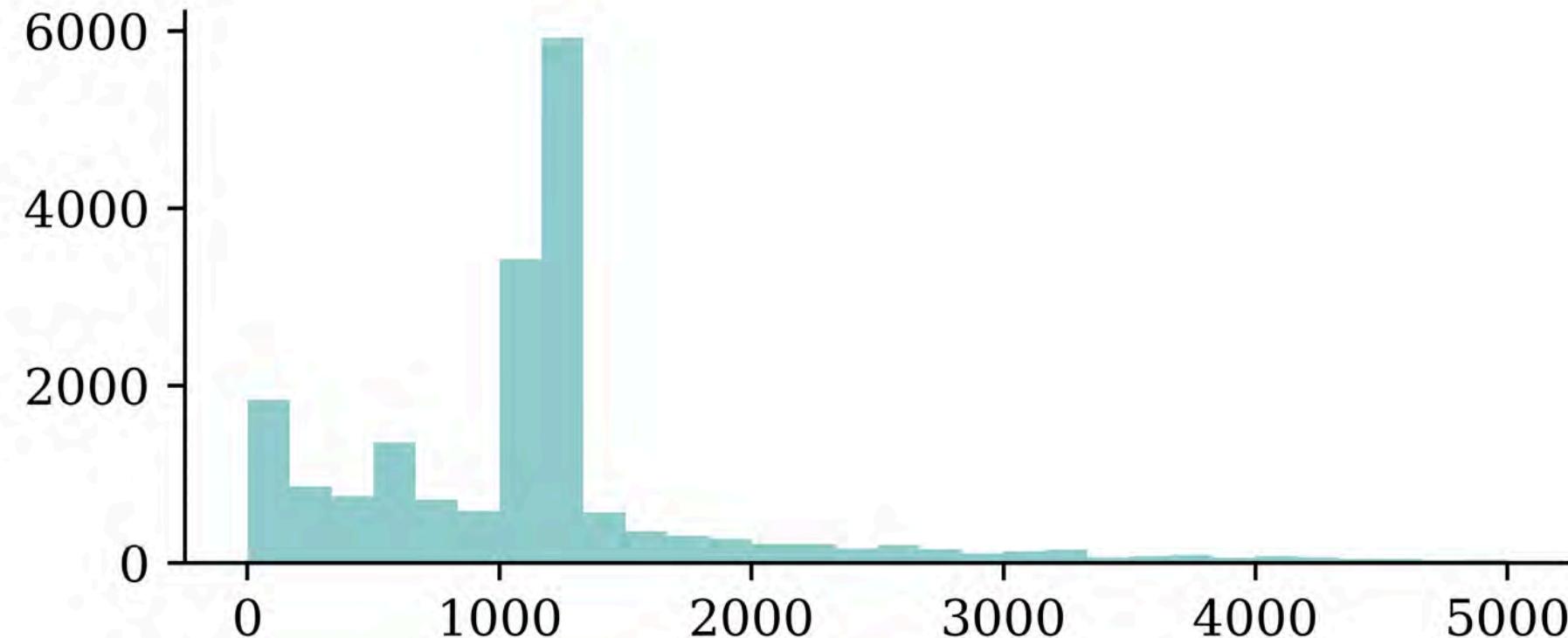
```
1 ct = make_column_transformer(  
2     (OrdinalEncoder(), ["Area", "VehGas"]),
3     ("drop", ["VehBrand", "Region"])),
4     remainder=StandardScaler(),
5     verbose_feature_names_out=False
6 )
7
8 X_train = ct.fit_transform(X_train)
9 X_test = ct.transform(X_test)
```

- **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);
```



# Gamma GLM

Suppose a fitted gamma GLM model has

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

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

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

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



# Gamma GLM loss

If  $Y | \mathbf{X} = \mathbf{x}$  is a gamma r.v. with mean  $\mu(\mathbf{x}; \boldsymbol{\beta})$  and dispersion parameter  $\phi$ , we can minimise the negative log-likelihood (NLL)

$$\text{NLL} \propto \sum_{i=1}^n \log \mu(\mathbf{x}_i; \boldsymbol{\beta}) + \frac{y_i}{\mu(\mathbf{x}_i; \boldsymbol{\beta})} + \text{const},$$

i.e., we ignore the dispersion parameter  $\phi$  while estimating the regression coefficients.



# Fitting Steps

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

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

Step 2. Estimate the dispersion parameter:

$$\phi = \frac{1}{n-p} \sum_{i=1}^n \frac{(y_i - \mu(\mathbf{x}_i; \boldsymbol{\beta}))^2}{\mu(\mathbf{x}_i; \boldsymbol{\beta})^2}$$



# 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()

```

```

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 # Dispersion Parameter
2 mus = gamma_glm.predict(X_train_design)
3 residuals = y_train - mus
4 variance = mus**2
5 dof = (len(y_train)-X_train.shape[1])
6 phi_glm = np.sum(residuals**2/variance)
7 print(phi_glm)

```

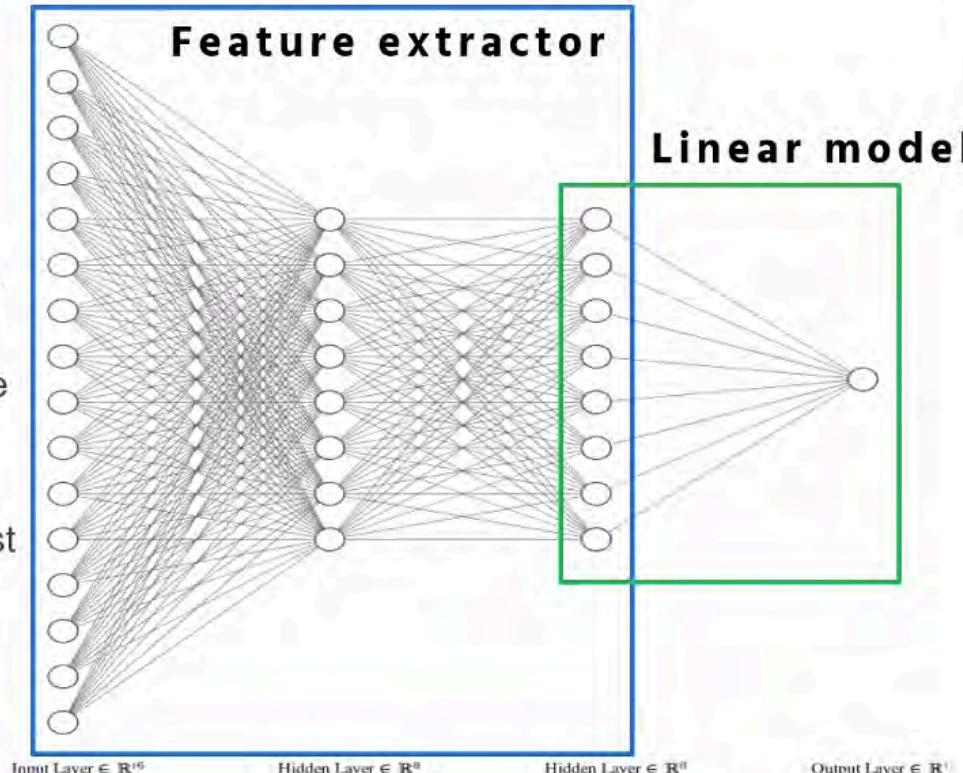
59.6306232357824



# ANN can feed into a GLM

## FCN generalizes GLM

- Intermediate layers = representation learning, guided by supervised objective.
- Last layer = (generalized) linear model, where input variables = new representation of data
- No need to use GLM – strip off last layer and use learned features in, for example, XGBoost
- Or mix with traditional method of fitting GLM



11 April 2022

14

Combining GLM & ANN.

# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- **Combined Actuarial Neural Network**
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# CANN

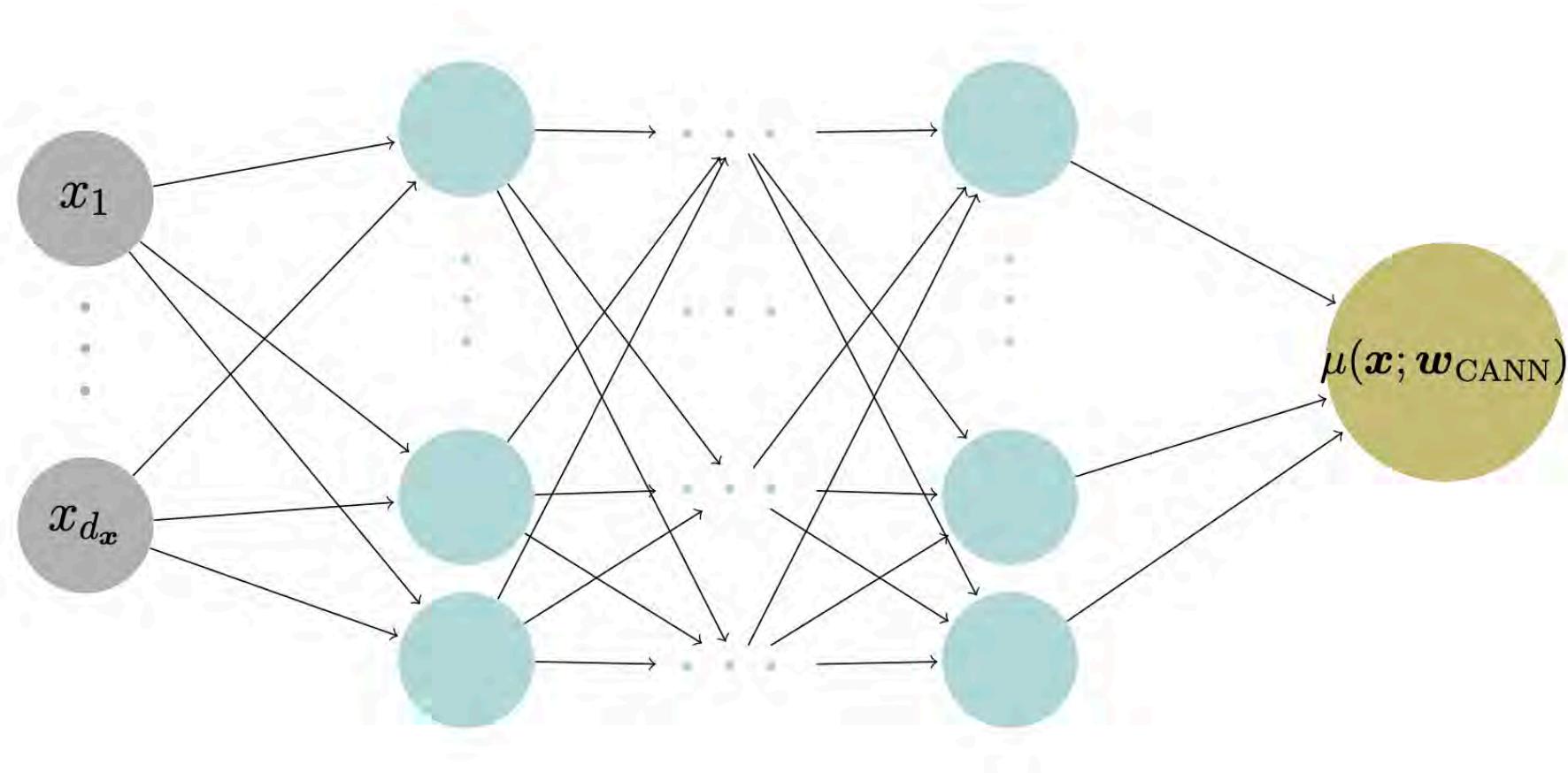
The Combined Actuarial Neural Network is a novel actuarial neural network architecture proposed by Schelldorfer and Wüthrich (2019). We summarise the CANN approach as follows:

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

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



# Architecture



CANN approach.



# Code: Architecture

```
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:].values
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=100,
4     callbacks=[EarlyStopping(patience=10)],
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 = y_train - mus
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)
```

0.0



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- **Mixture Density Network**
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# 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 p.d.f. and the c.d.f 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  $\pi_k$ '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.,

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

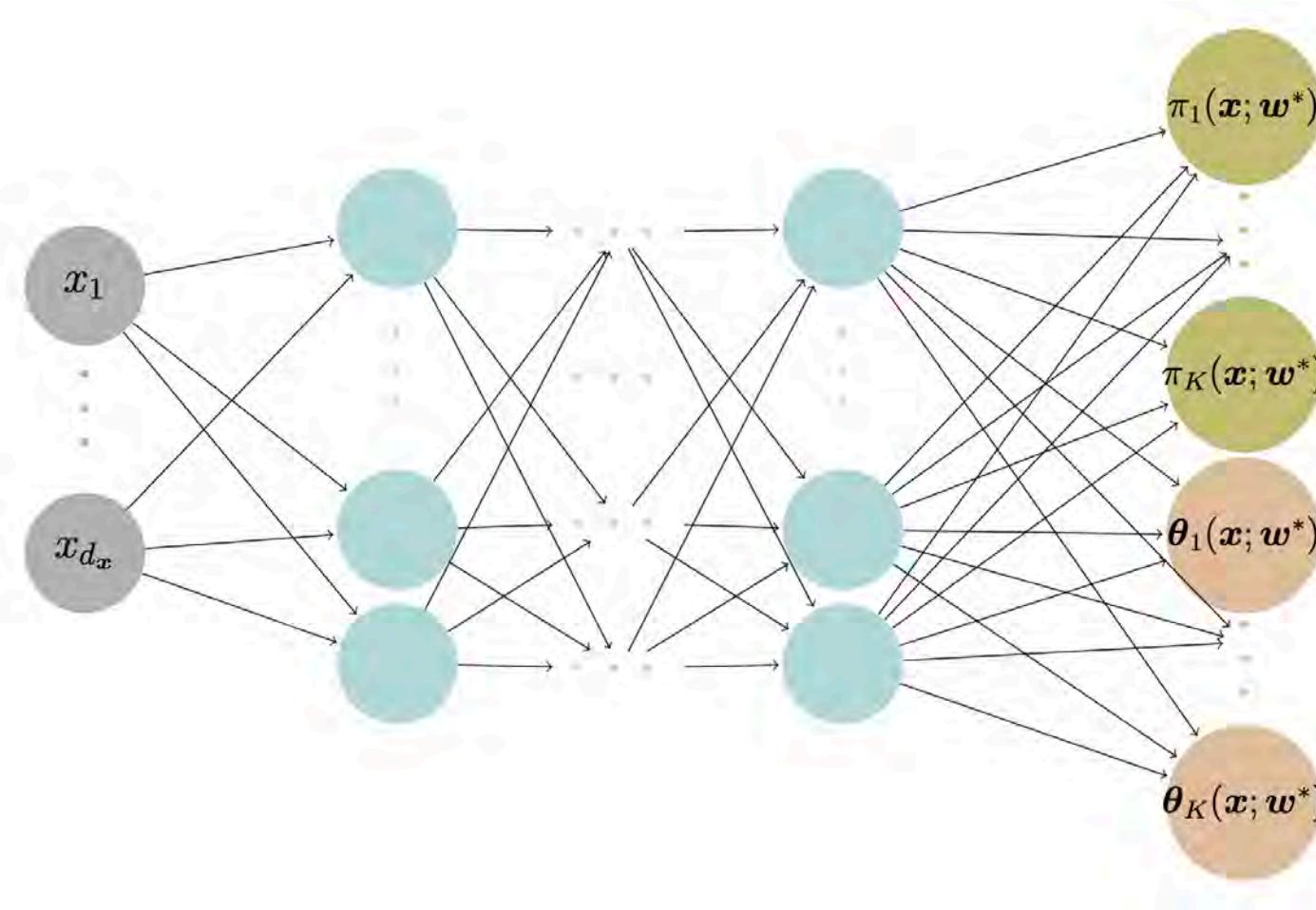
where  $\mathbf{w}^*$  is the networks' weights found by minimising the following negative log-likelihood loss function

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

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



# Mixture Density Network



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

The density of the actual claim amount  $Y | \mathbf{X} = \mathbf{x}$  follows

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

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



# Output

The aim is to find the optimum weights

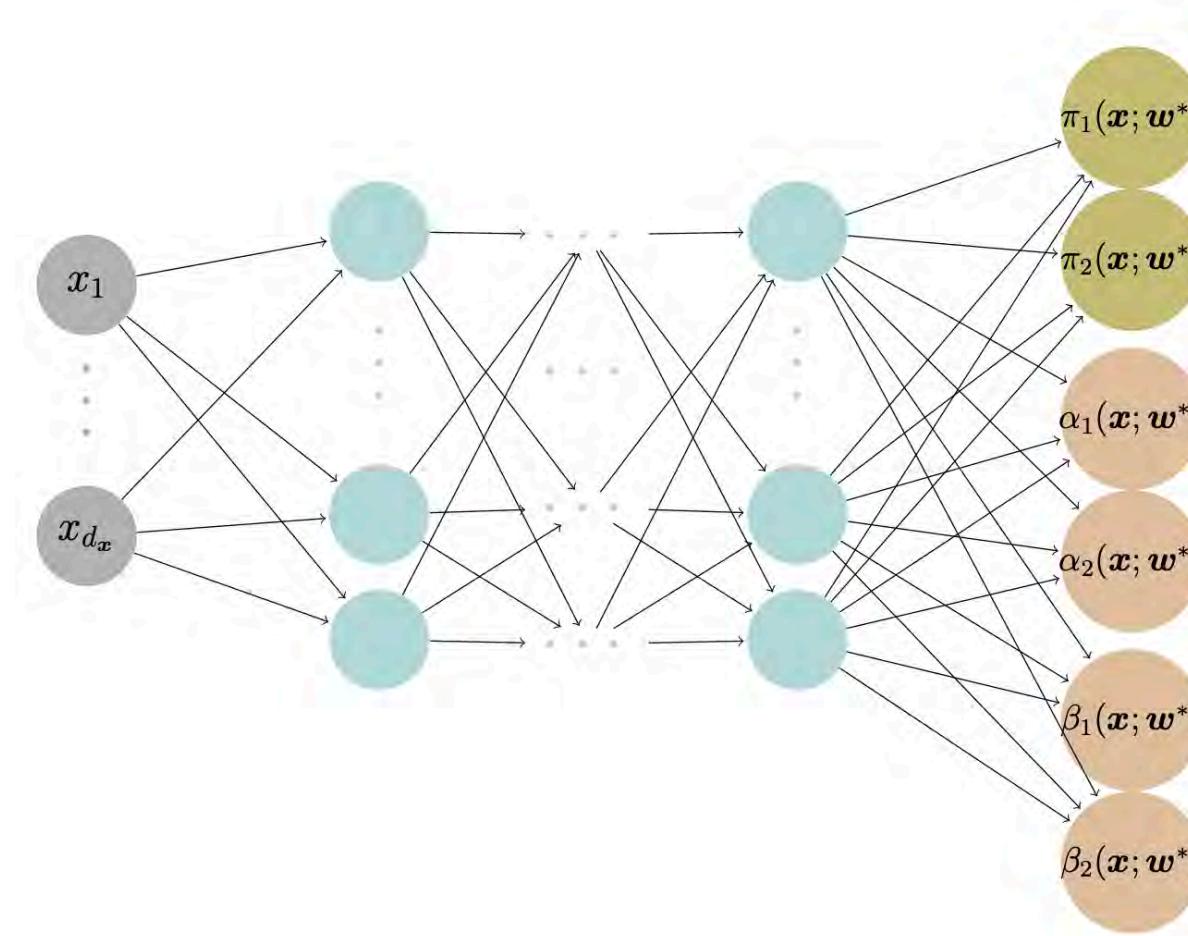
$$\boldsymbol{w}^* = \arg \min_{\boldsymbol{w}} \mathcal{L}(\mathcal{D}, \boldsymbol{w})$$

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



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



# Code: Import “legacy” Keras (for now)

```
1 import tf_keras
```

## Keras 3 breaks Tensorflow Probability upon import #1774

Closed

AjaniStewart opened this issue on Dec 5, 2023 · 4 comments



jburnim commented on Dec 6, 2023

Member

...

TensorFlow Probability 0.23.0 is not compatible with Keras 3.

And it looks like TensorFlow 2.15.0 is also not compatible with Keras 3. When I run those pip commands, upgrading Keras gives the error message:

```
tensorflow 2.15.0 requires keras<2.16,>=2.15.0, but you have keras 3.0.0 which is incompatible
```

It will be possible to import TensorFlow Probability 0.24.0 with TensorFlow 2.16.0 and Keras 3 installed. But please note that TensorFlow Probability 0.24.0 will continue to use Keras 2, which means that tf-keras 2.16.0 will also have to be installed.



UNSW  
SYDNEY

# 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 = tf_keras.layers.Input(shape=X_train.shape[1:])
5
6 # Two hidden layers
7 x = tf_keras.layers.Dense(64, activation='relu')(inputs)
8 x = tf_keras.layers.Dense(64, activation='relu')(x)
9
10 pis = tf_keras.layers.Dense(2, activation='softmax')(x) # Mixing weights
11 alphas = tf_keras.layers.Dense(2, activation='exponential')(x) # Shape parameters
12 betas = tf_keras.layers.Dense(2, activation='exponential')(x) # Scale parameters
13
14 # `y_pred` will now have 6 columns
15 gamma_mdn = tf_keras.Model(inputs, tf_keras.layers.concatenate(axis=1)([pis, alphas, bet
```



# 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=100,
6     callbacks=[EarlyStopping(patience=10)],
7     verbose=0,
8     batch_size=64,
9     validation_split=0.2)
```



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- **Metrics for Distributional Regression**
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# Proper Scoring Rules

## Definition

*The scoring rule  $S : \mathcal{F} \times \mathbb{R} \rightarrow \overline{\mathbb{R}}$  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: nan

MDN: 8.67



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- **Aleatoric and Epistemic Uncertainty**
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- Ensembles



# Categories of uncertainty

There are two major categories 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

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

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



# Epistemic Uncertainty

## Qualitative Definition

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

## Quantitative Definition

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

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



# Sources of uncertainty

*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.



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- **Avoiding Overfitting**
- Dropout
- Dead ReLU neurons
- Ensembles



# Traditional regularisation

Say all the  $m$  weights (excluding biases) are in the vector  $\theta$ . If we change the loss function to

$$\text{Loss}_{1:n} = \frac{1}{n} \sum_{i=1}^n \text{Loss}_i + \lambda \sum_{j=1}^m |\theta_j|$$

this would be using  $L^1$  regularisation. A loss like

$$\text{Loss}_{1:n} = \frac{1}{n} \sum_{i=1}^n \text{Loss}_i + \lambda \sum_{j=1}^m \theta_j^2$$

is called  $L^2$  regularisation.



# Regularisation in Keras

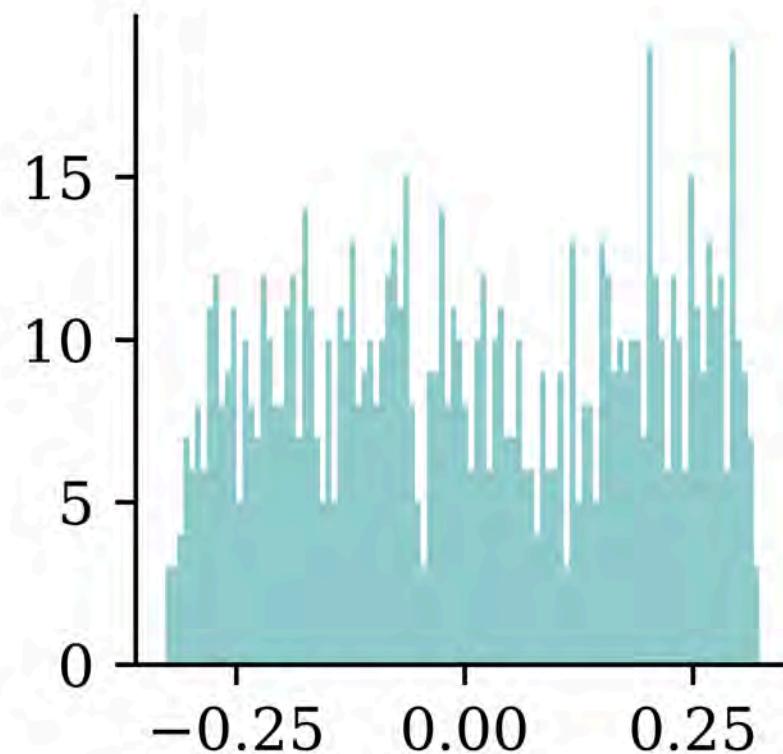
```
1 from keras.regularizers import L1, L2
2
3 def l1_model(regulariser_strength=0.01):
4     random.seed(123)
5     model = Sequential([
6         Dense(30,
7             activation="leaky_relu",
8             name="hidden1"),
9         Dense(30,
10            activation="leaky_relu",
11            kernel_regularizer=L1(regulariser_strength),
12            name="hidden2"),
13         Dense(1,
14             activation="exponential",
15             name="output")
16     ])
17
18     model.compile("adam", "mse")
19     model.fit(X_train_sc, y_train, epochs=4, verbose=0)
20     return model
```



# Weights before & after $\ell_1$

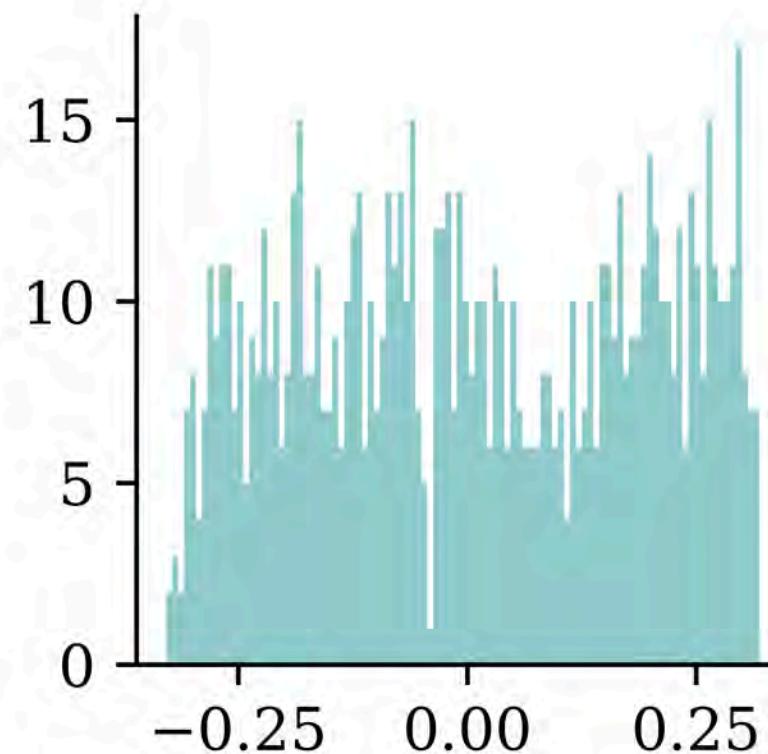
```
1 model = l1_model(0.0)
2 weights = model.get_layer("hidden2").get_weights()
3 print(f"Number of weights almost 0: {np.sum(np.isclose(weights, 0))}")
4 plt.hist(weights, bins=100);
```

Number of weights almost 0: 0



```
1 model = l1_model(1.0)
2 weights = model.get_layer("hidden2").get_weights()
3 print(f"Number of weights almost 0: {np.sum(np.isclose(weights, 0))}")
4 plt.hist(weights, bins=100);
```

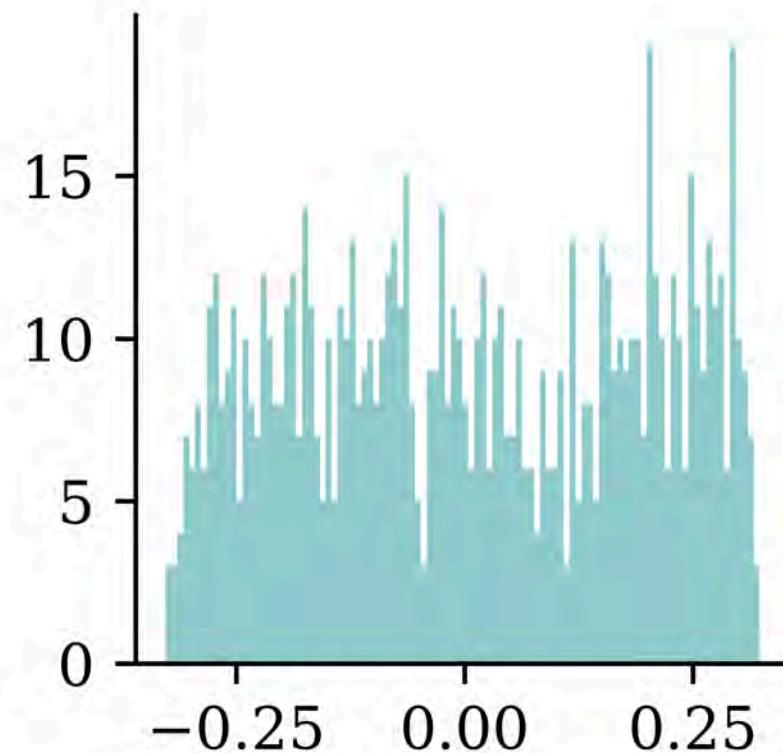
Number of weights almost 0: 0



# Weights before & after $\ell_2$

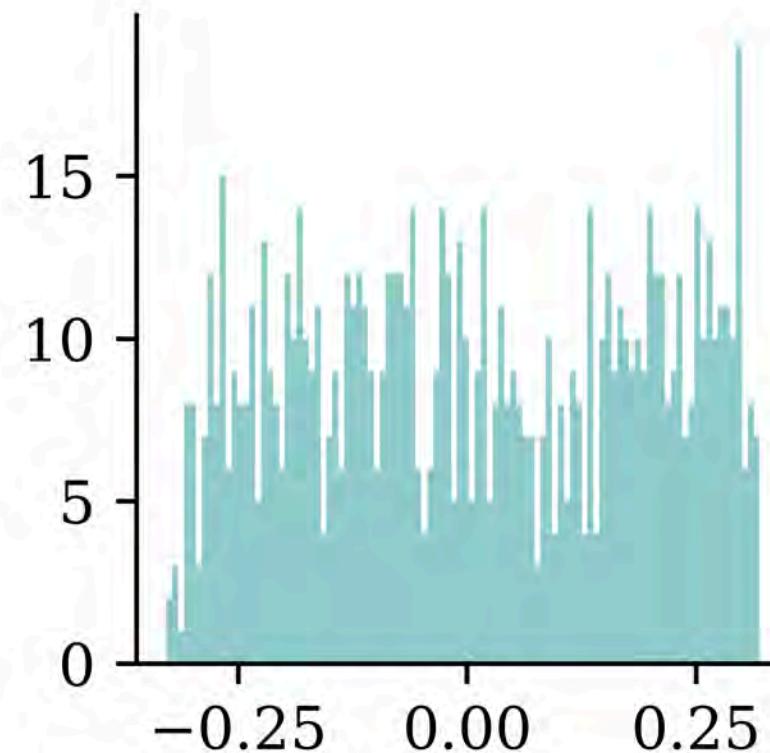
```
1 model = l2_model(0.0)
2 weights = model.get_layer("hidden2").get_weights()
3 print(f"Number of weights almost 0: {np.sum(np.isclose(weights, 0))}")
4 plt.hist(weights, bins=100);
```

Number of weights almost 0: 0



```
1 model = l2_model(1.0)
2 weights = model.get_layer("hidden2").get_weights()
3 print(f"Number of weights almost 0: {np.sum(np.isclose(weights, 0))}")
4 plt.hist(weights, bins=100);
```

Number of weights almost 0: 0



# Early-stopping regularisation

A very different way to regularize iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping... It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch”.

Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a scientist at Google, has dubbed this the “stretch pants” approach: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

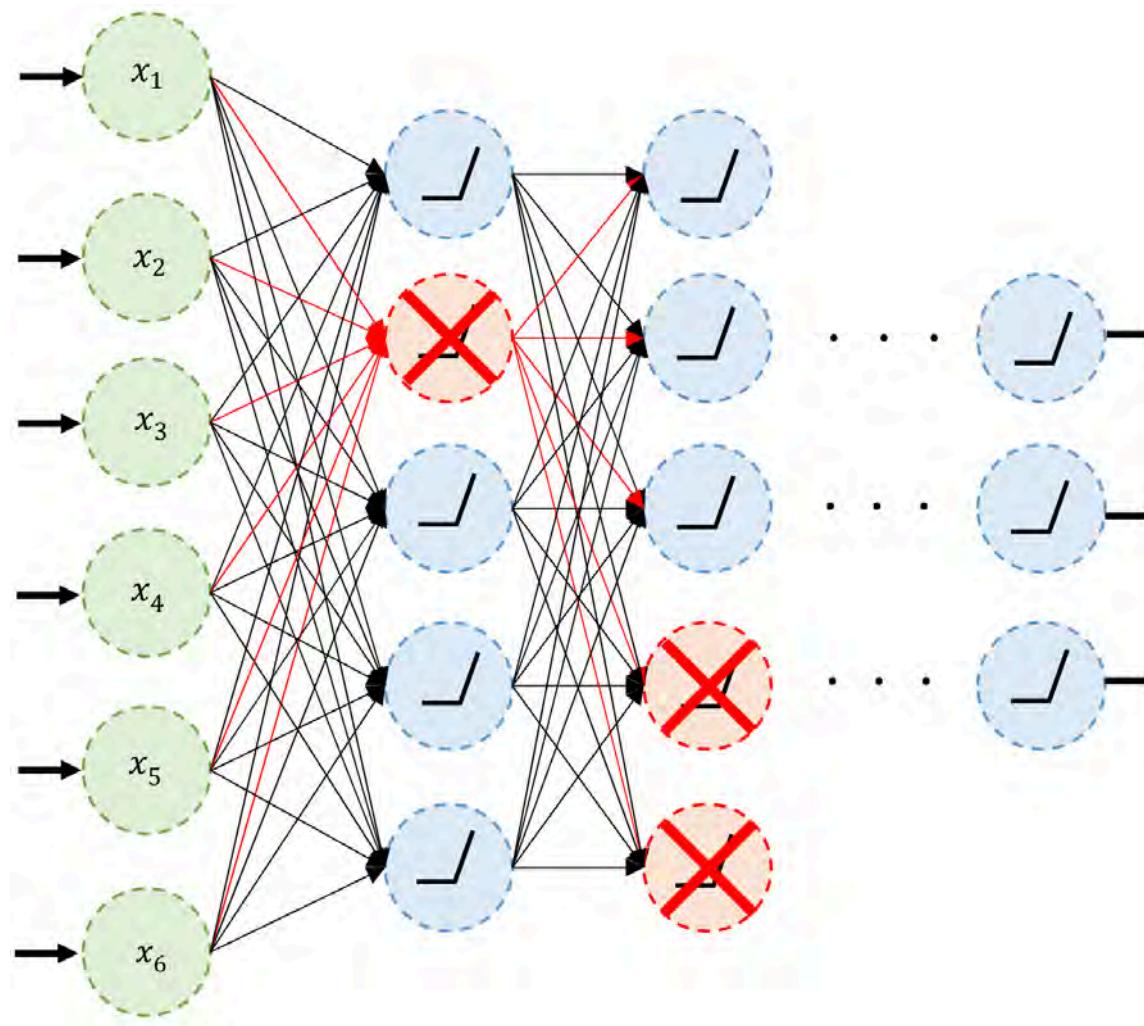


# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- **Dropout**
- Dead ReLU neurons
- Ensembles



# Dropout



An example of neurons dropped during training:



Sources: Marcus Lautier (2022).



UNSW  
SYDNEY

# 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



UNSW  
SYDNEY

## 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



UNSW  
SYDNEY

# Code: Dropout

Dropout is just another layer in Keras.

```
1 from keras.layers import Dropout
2
3 random.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_sc, y_train, epochs=4, verbose=0);
```



# Code: Dropout after training

Making predictions is the same as any other model:

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

```
array([[1.0587903],
       [1.2814349],
       [0.9994641]], dtype=float32)
```

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

```
array([[1.0587903],
       [1.2814349],
       [0.9994641]], dtype=float32)
```

We can make the model think it is still training:

```
1 model(X_train_sc.head(3),
2       training=True).numpy()
```

```
array([[1.082524  ],
       [0.74211466],
       [1.1583111 ]], dtype=float32)
```

```
1 model(X_train_sc.head(3),
2       training=True).numpy()
```

```
array([[1.0132376],
       [1.2697867],
       [0.7800578]], 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.



# Lecture Outline

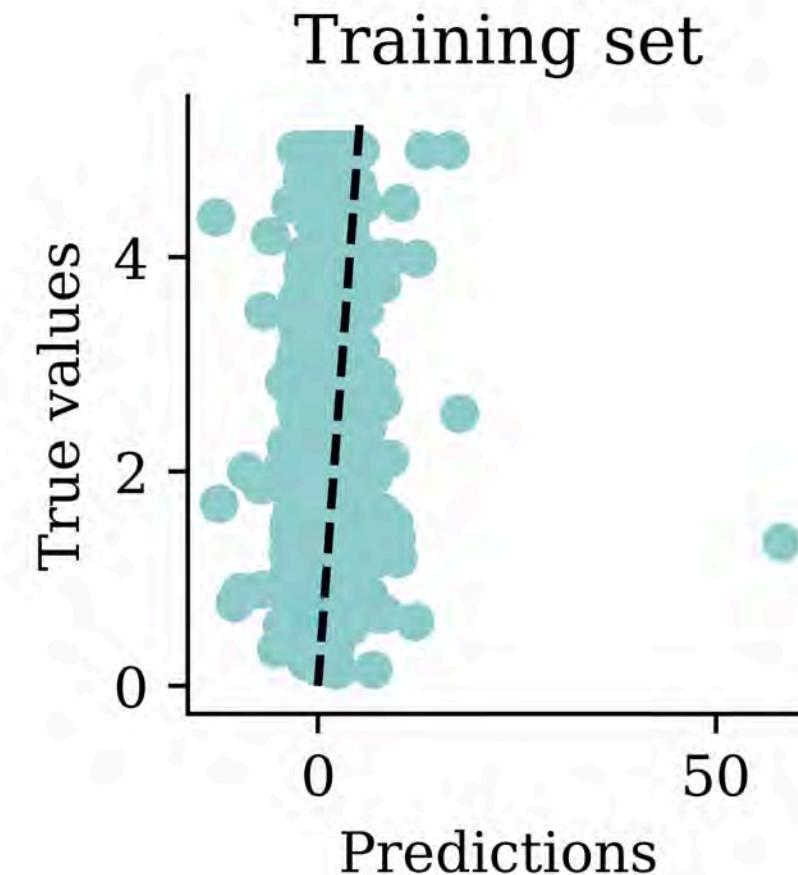
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- **Dead ReLU neurons**
- Ensembles



# My first ANN for California housing

```
1 random.seed(123)
2
3 model = Sequential([
4     Dense(30,
5         activation="relu",
6         name="hidden"),
7     Dense(1)
8 ])
9
10 model.compile("adam", "mse")
11 hist = model.fit(X_train, y_train, epoch
12 hist.history["loss"]
```

```
[25089.478515625,
12.956829071044922,
13.395614624023438,
7.074806213378906,
5.800335884094238]
```



# Find dead ReLU neurons

```
1 acts = model.get_layer("hidden")(X_train).numpy()  
2 acts[:3]  
  
array([[261.458    , 502.33704   , 93.64283   , ... , 537.54865   , 325.7366   ,  
       398.99435  ],  
      [ 18.983932,  52.9067   ,  0.        , ... , 28.361092,  10.988864,  
       58.194595],  
      [266.2954   , 517.58154   , 98.64309   , ... , 553.68005   , 336.69986   ,  
       411.61124  ]], dtype=float32)
```

```
1 acts.mean(axis=0)  
  
array([219.19604, 434.435   , 81.71622, ... , 457.9593   , 275.58286,  
      348.6399  ], dtype=float32)
```

```
1 np.sum(acts.mean(axis=0) == 0)
```

7



# Trying different seeds

Create a function which counts the number of dead ReLU neurons in the first hidden layer for a given seed:

```
1 def count_dead(seed):
2     random.seed(seed)
3     hidden = Dense(30, activation="relu")
4     acts = hidden(X_train).numpy()
5     return np.sum(acts.mean(axis=0) == 0)
```

Then we can try out different seeds:

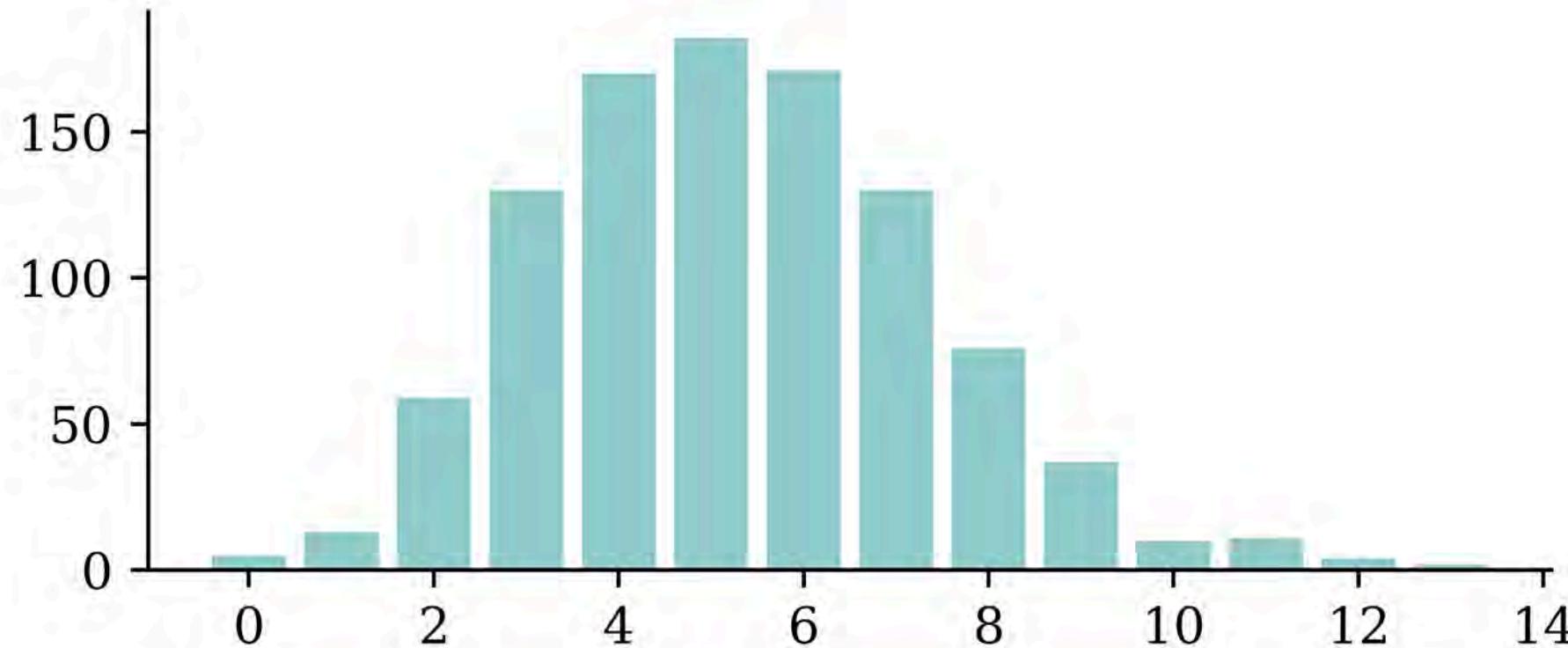
```
1 num_dead = [count_dead(seed) for seed in range(1_000)]
2 np.median(num_dead)
```

5.0



# Look at distribution of dead ReLUs

```
1 labels, counts = np.unique(num_dead, return_counts=True)
2 plt.bar(labels, counts, align='center');
```



# Lecture Outline

- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Avoiding Overfitting
- Dropout
- Dead ReLU neurons
- **Ensembles**



# Deep Ensembles

Train  $D$  neural networks with different random weights initialisations independently (can do it in parallel).

The trained weights are  $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(D)}$ .

```

1 def build_model(seed):
2     random.seed(seed)
3     model = Sequential([
4         Dense(30, activation="leaky_relu"),
5         Dense(1, activation="exponential")
6     ])
7     model.compile("adam", "mse")
8
9     es = EarlyStopping(restore_best_weights=True, patience=5)
10    model.fit(X_train_sc, y_train, epochs=1_000,
11                callbacks=[es], validation_data=(X_val_sc, y_val), verbose=False)
12    return model

```

```

1 N = 10
2 seeds = range(N)
3 models = []
4 for seed in seeds:
5     models.append(build_model(seed))

```



# Deep Ensembles II

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

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

```

1 y_preds = []
2 for model in models:
3     y_preds.append(model.predict(X_test_sc, verbose=0))
4
5 y_preds = np.array(y_preds)
6 y_preds

```

```

array([[ [3.2801466 ],
       [0.76298356],
       [2.4068608 ],
       ...,
       [2.3385763 ],
       [2.1730225 ],
       [1.096715  ]],

       [[3.1832185 ],
       [0.72296774],
       [2.5727806 ],
       ...,
       [2.3812106 ],
       [2.27971    ],
       [1.06247    ]], ...

```



# 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          : 3.9.0  
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
- deep ensembles
- CANN
- GLM
- MDN
- mixture distribution
- posterior sampling
- proper scoring rule

