

Distributional Regression

ACTL3143 & ACTL5111 Deep Learning for Actuaries
Patrick Laub

Lecture Outline

- **Introduction**
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- Ensembles

Why actuaries avoid neural networks

They're not *inherently interpretable*, so just we have to look at inputs and outputs from the black box.








Neural network models typically just output a prediction without any sense of its *confidence*.

Therefore we cannot *trust* the neural network models, which is a **dealbreaker**.




Now let's focus on actuarial problems.

Car insurance

Claim size prediction

|  Age |  Age |  Type |
|---|---|--|
| 25 | 3 |  Sedan |
| 40 | 5 |  SUV |
| 19 | 1 |  Sports Car |
| 60 | 10 |  Hatchback |

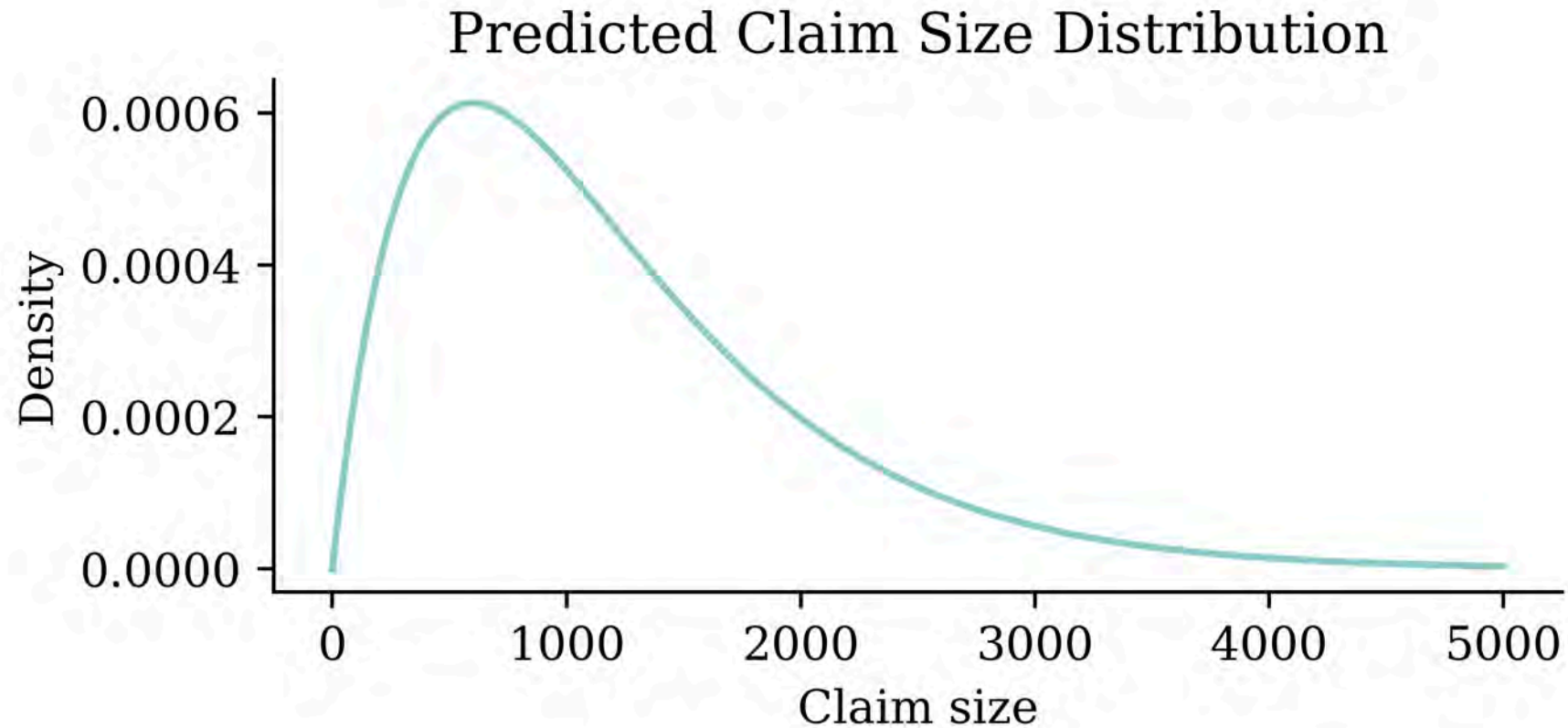


| Cost |
|---|
|  \$1,200 |
|  \$2,500 |
|  \$3,800 |
|  \$800 |

What's wrong? Not enough rows? Not enough columns?

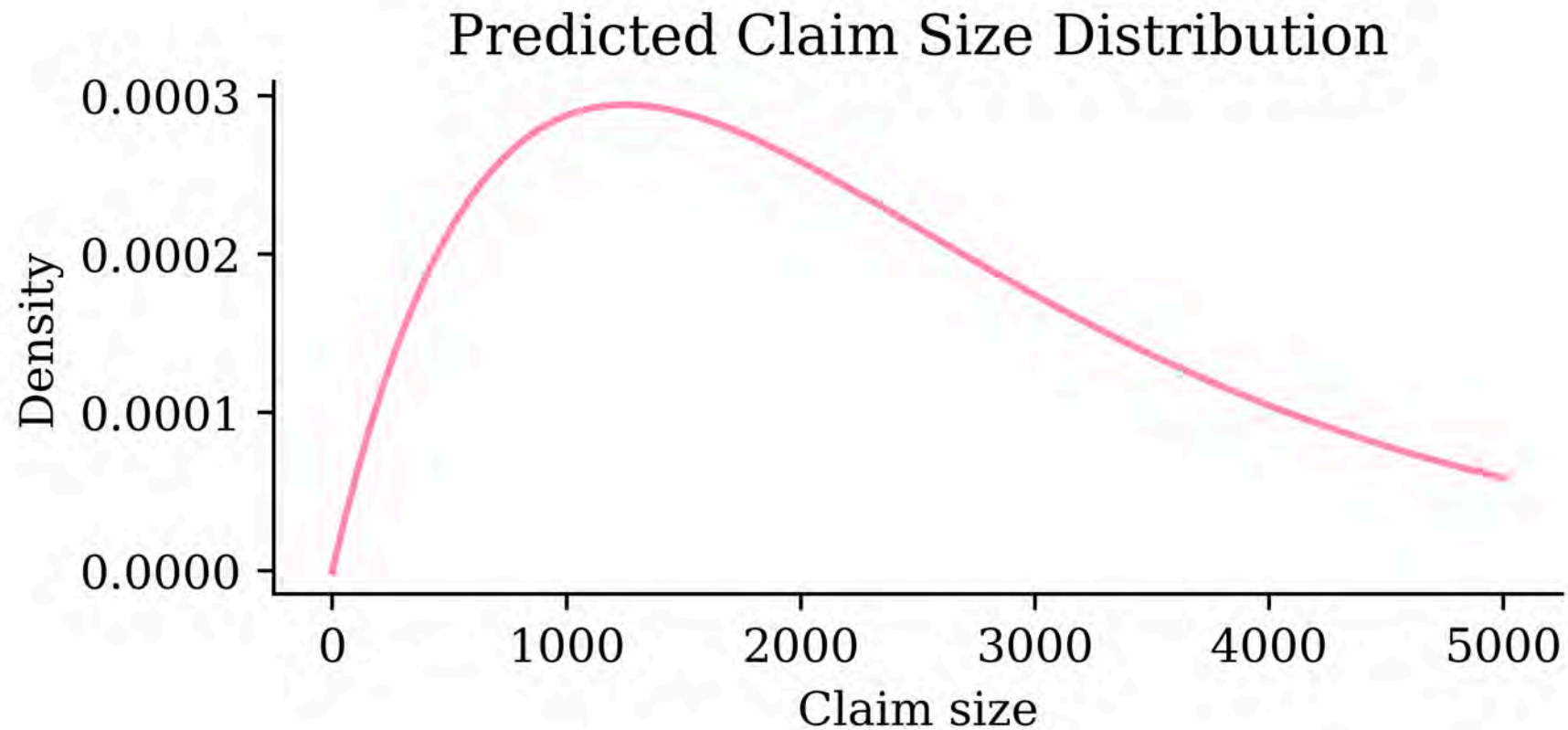
Distributional regression

Customer 1 = (25, 3, 🚗)



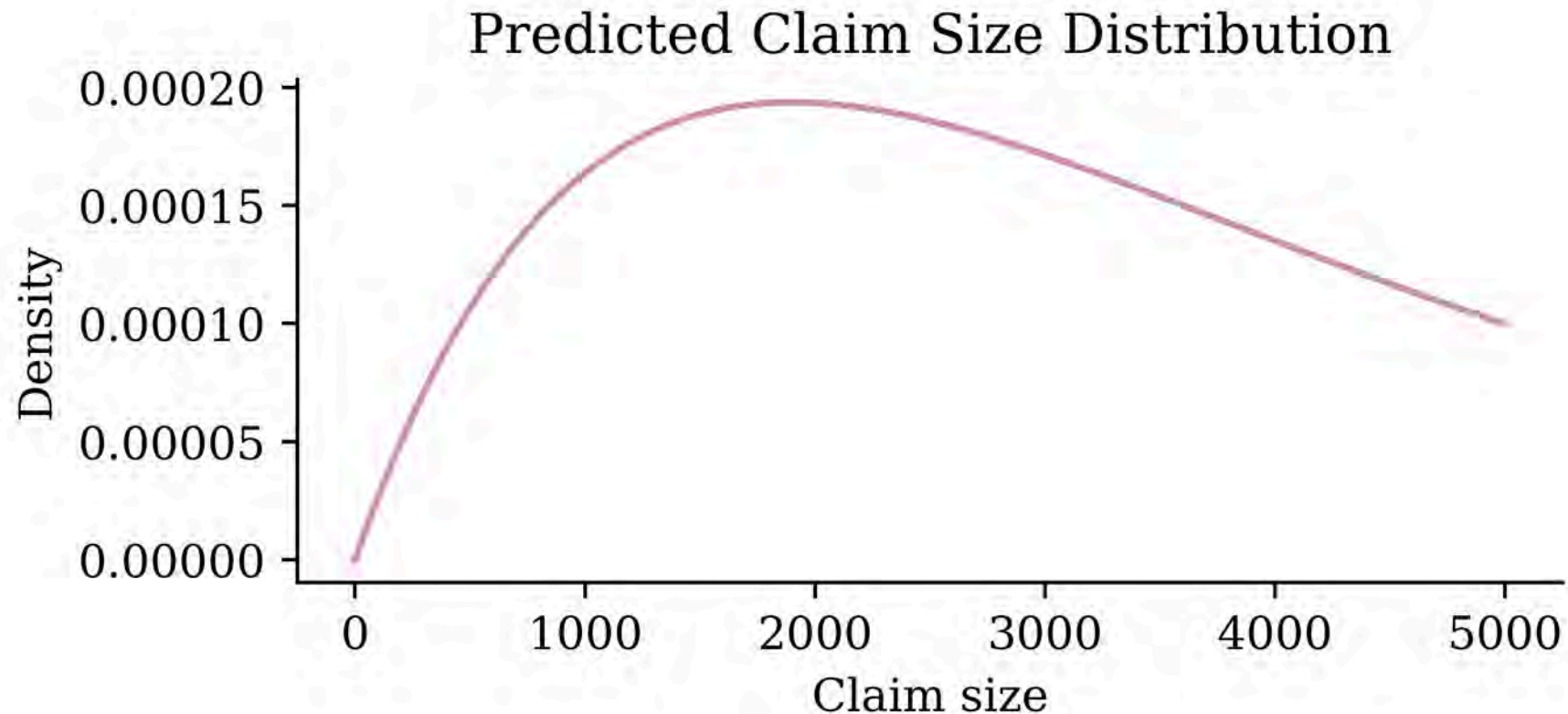
Distributional regression

Customer 2 = (40, 5, 🚗)



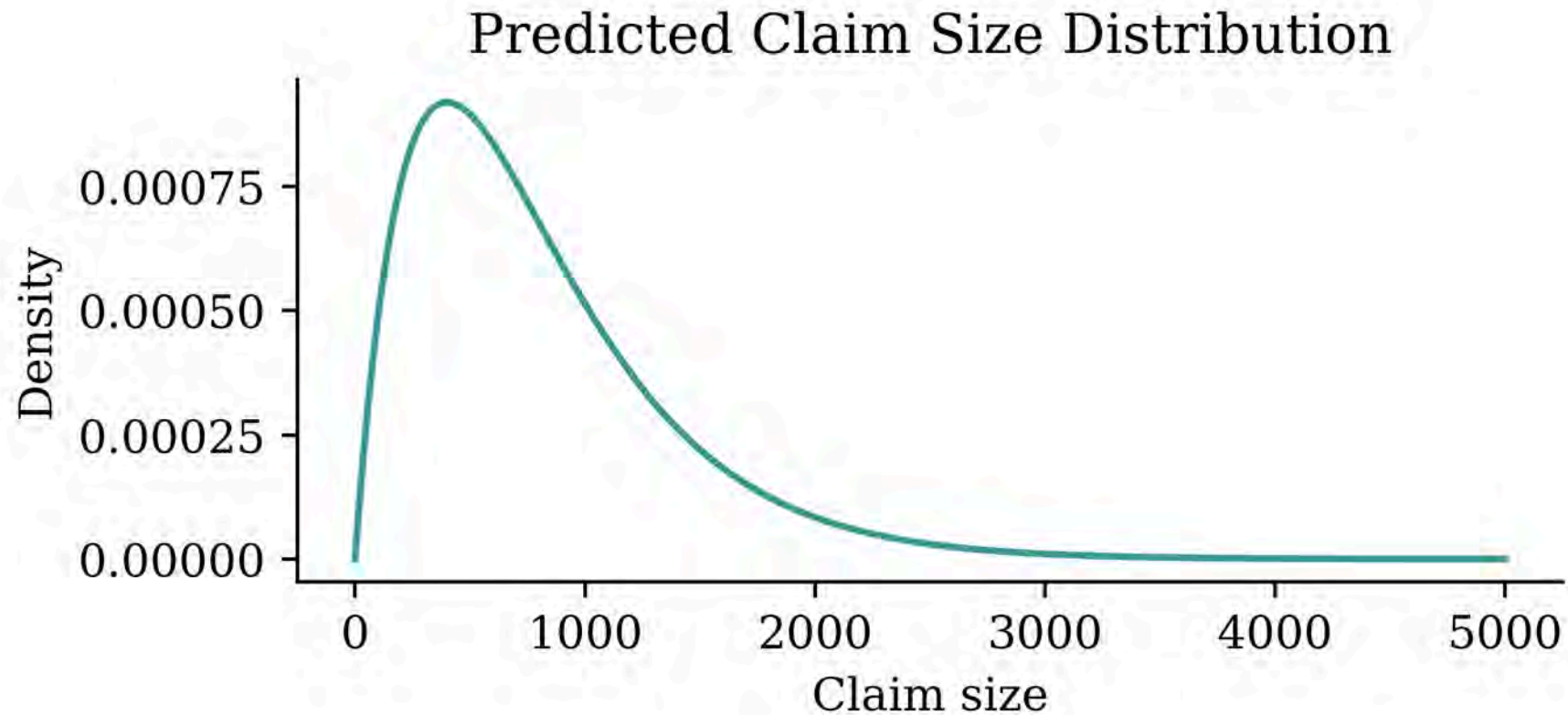
Distributional regression

Customer 3 = (19, 1, 🚗)



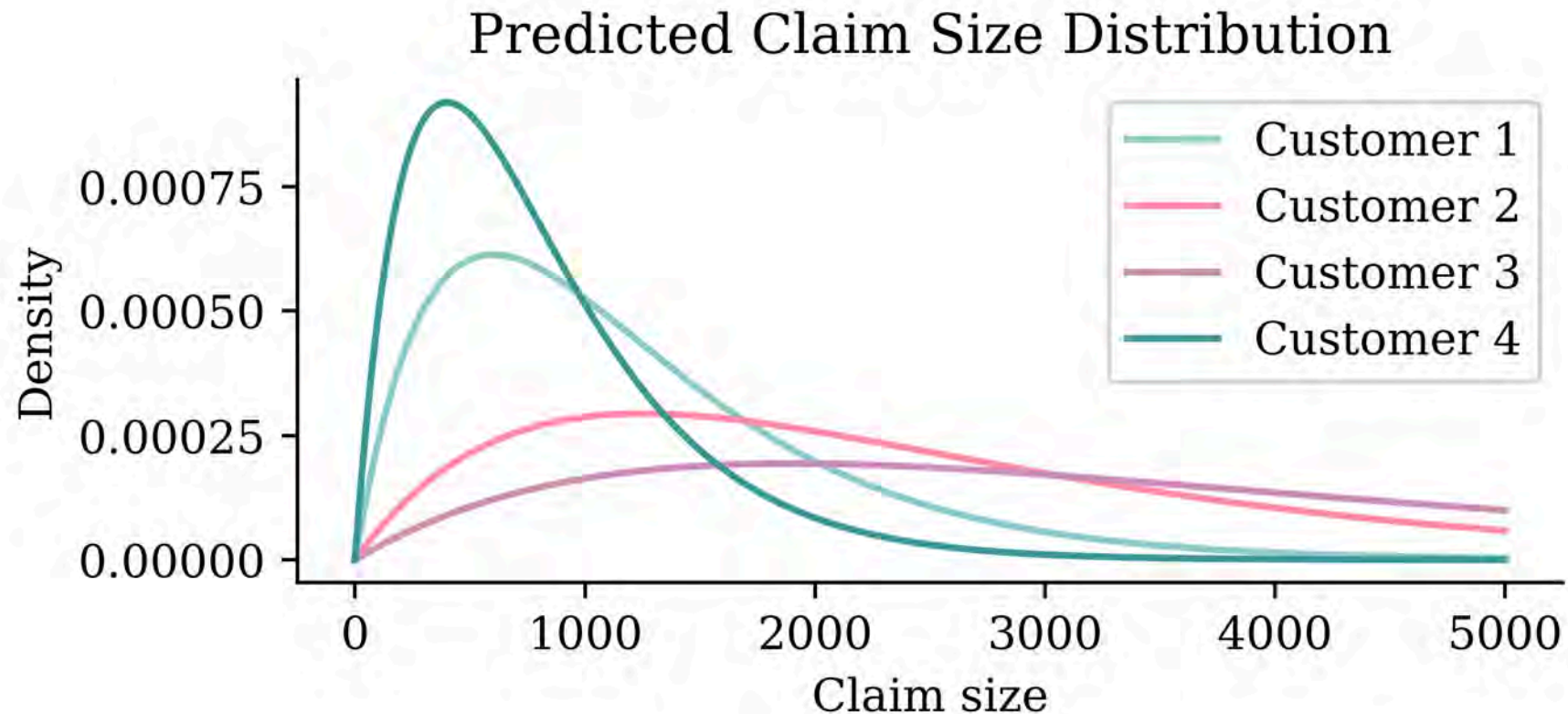
Distributional regression

Customer 4 = (60, 10, 🚗)






Distributional regression

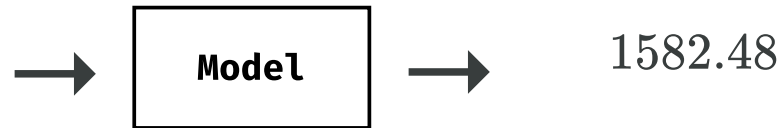
All customers






Traditional vs distributional regression

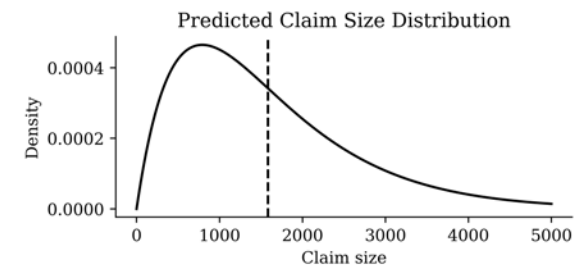
Traditional:

| | | |
|---|---|---|
|  |  |  |
| 35 | 12 | Sport |



Distributional:

| | | |
|---|---|---|
|  |  |  |
| 35 | 12 | Sport |



Baseline: a generalised linear model

A gamma GLM with a log link function:

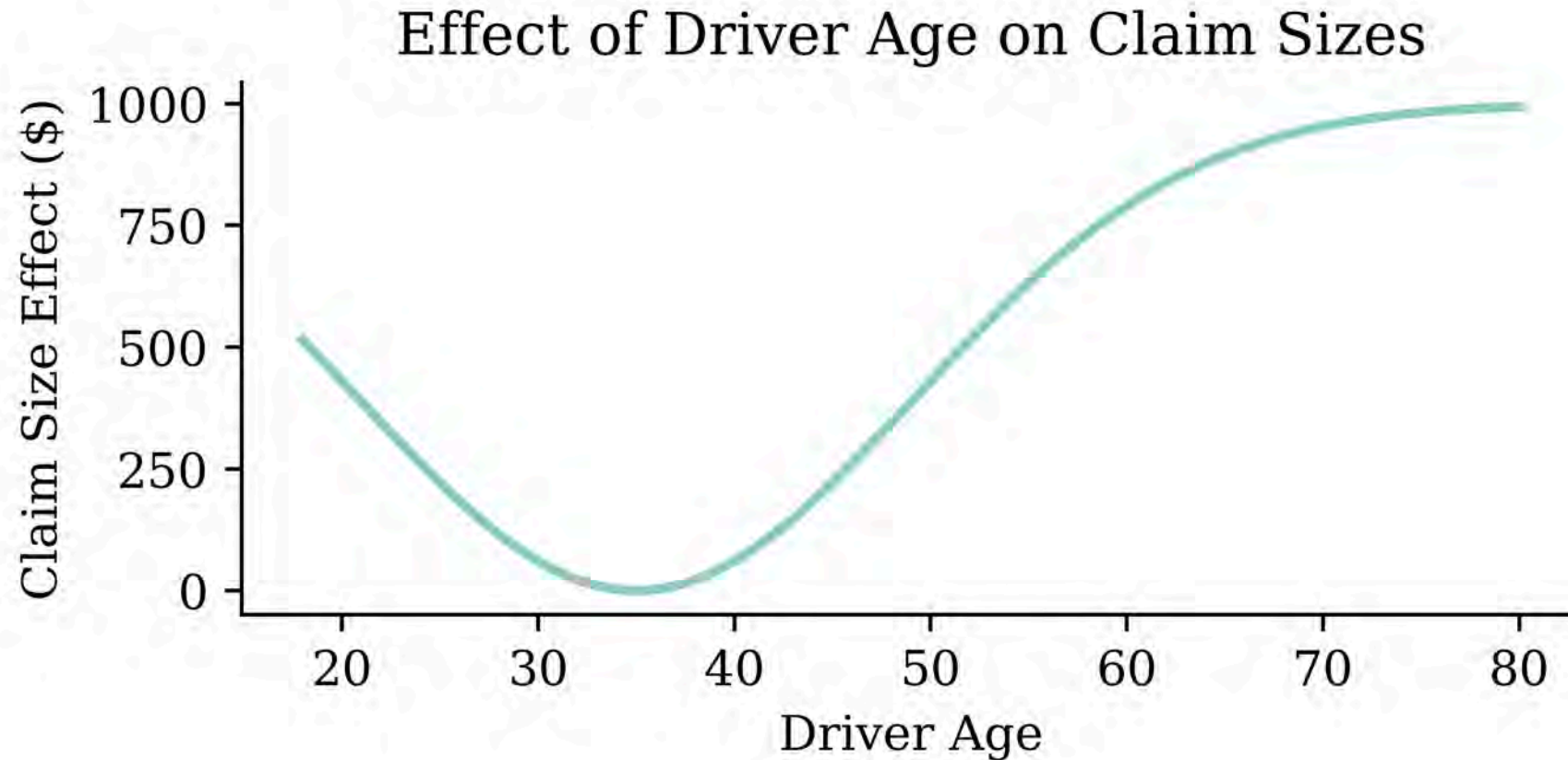
$$Y|\mathbf{X} \sim \text{Gamma}(\dots, \dots)$$
$$\mathbb{E}[Y|\mathbf{X}] = \exp\left\{\beta_0 + \beta_1 \cdot \text{Age} + \beta_2 \cdot \text{Car Age} + \beta_3 \cdot \text{Type}\right\}$$

A simple model, easy to train and interpret, but...

⚠ GLMs can be

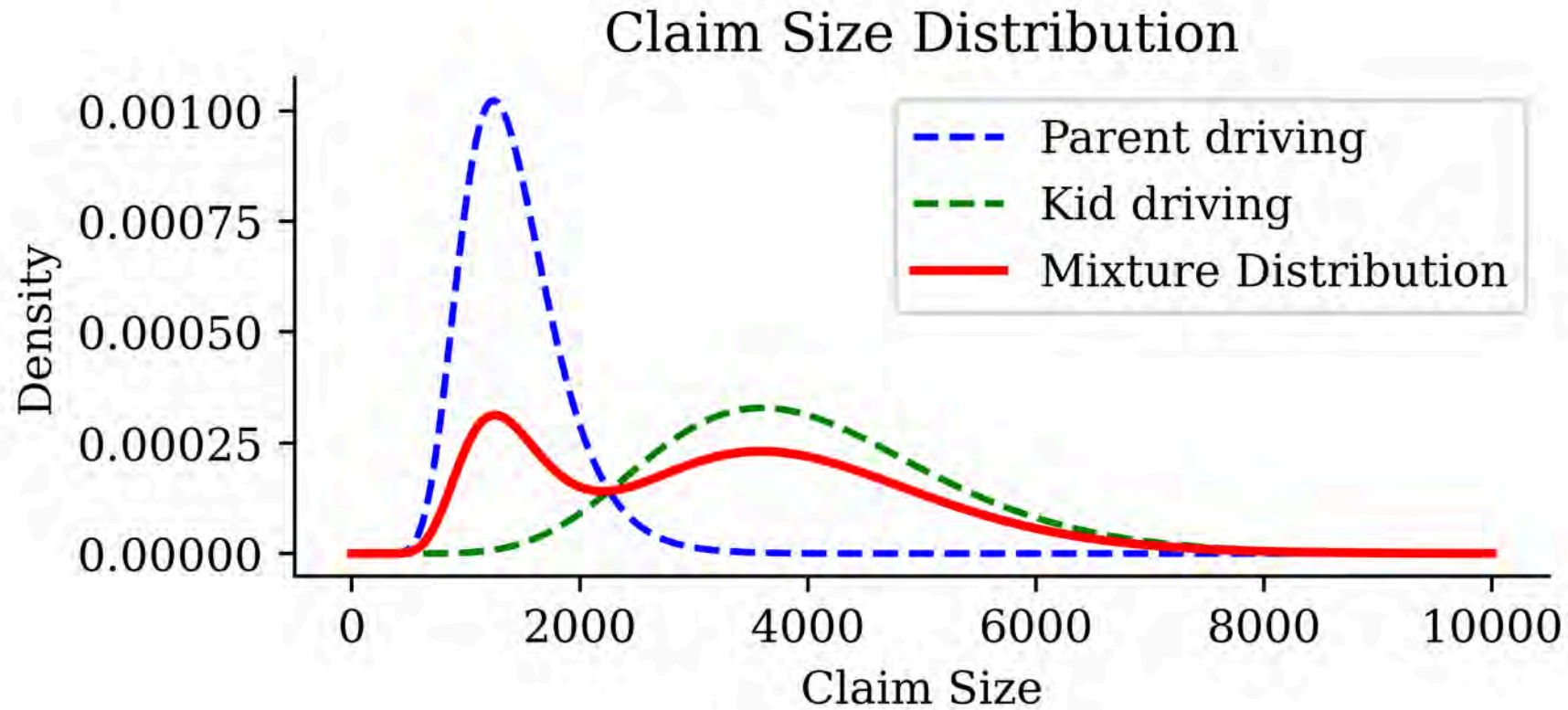
1. Bad at *regression*
2. Bad at *distributional* regression

Example 1: Non-monotonicity

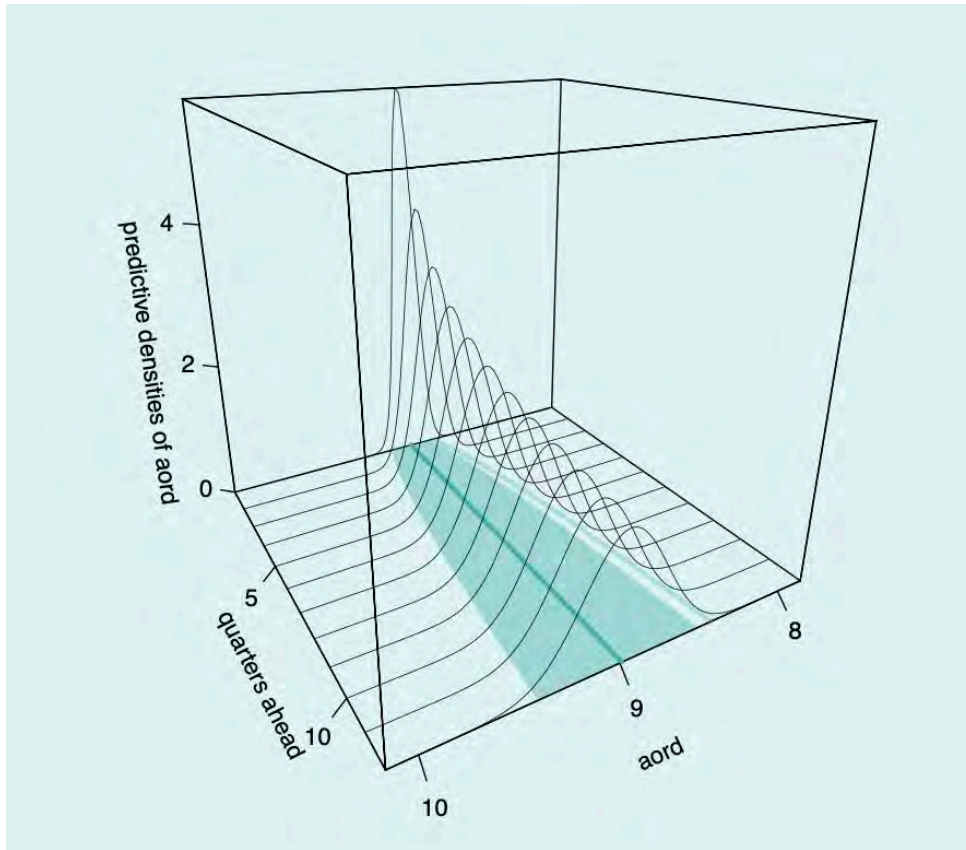


GLMs cannot (easily) do this → Use a neural network

Example 2: Multi-modality



Key idea



An example of distributional forecasting over the All Ordinaries Index

- Earlier machine learning models focused on point estimates.
- However, in many applications, we need to understand the distribution of the response variable.
- Each prediction becomes a *distribution* over the possible outcomes

We will focus on regression not classification

Classifiers already give us a probability, which is a big step up compared to regression models.

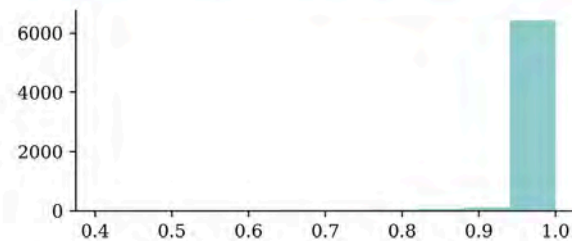
However, neural networks' "probabilities" can be overconfident.

Confidence of predictions

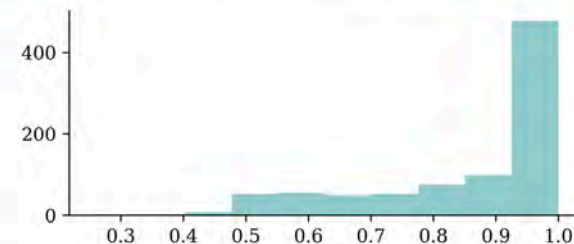
```
y_pred = keras.ops.convert_to_numpy(keras.activations.softmax(model(X_test)))
y_pred_class = np.argmax(y_pred, axis=1)
y_pred_prob = y_pred[np.arange(y_pred.shape[0]), y_pred_class]

confidence_when_correct = y_pred_prob[y_pred_class == y_test]
confidence_when_wrong = y_pred_prob[y_pred_class != y_test]
```

```
plt.hist(confidence_when_correct);
```



```
plt.hist(confidence_when_wrong);
```



We **already saw** a case of this.

See Guo et al. (2017), [On Calibration of Modern Neural Networks](#).

Lecture Outline

- Introduction
- **Traditional Regression**
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- 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 $\boldsymbol{\beta} = (\beta_0, \beta_1, \dots, \beta_p)$,
- β_0 is the intercept, β_1, \dots, β_p are the coefficients,
- $\boldsymbol{\beta}^*$ is the estimated coefficient vector,
- $\boldsymbol{x}_i = (1, x_{i1}, x_{i2}, \dots, x_{ip})$ 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

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

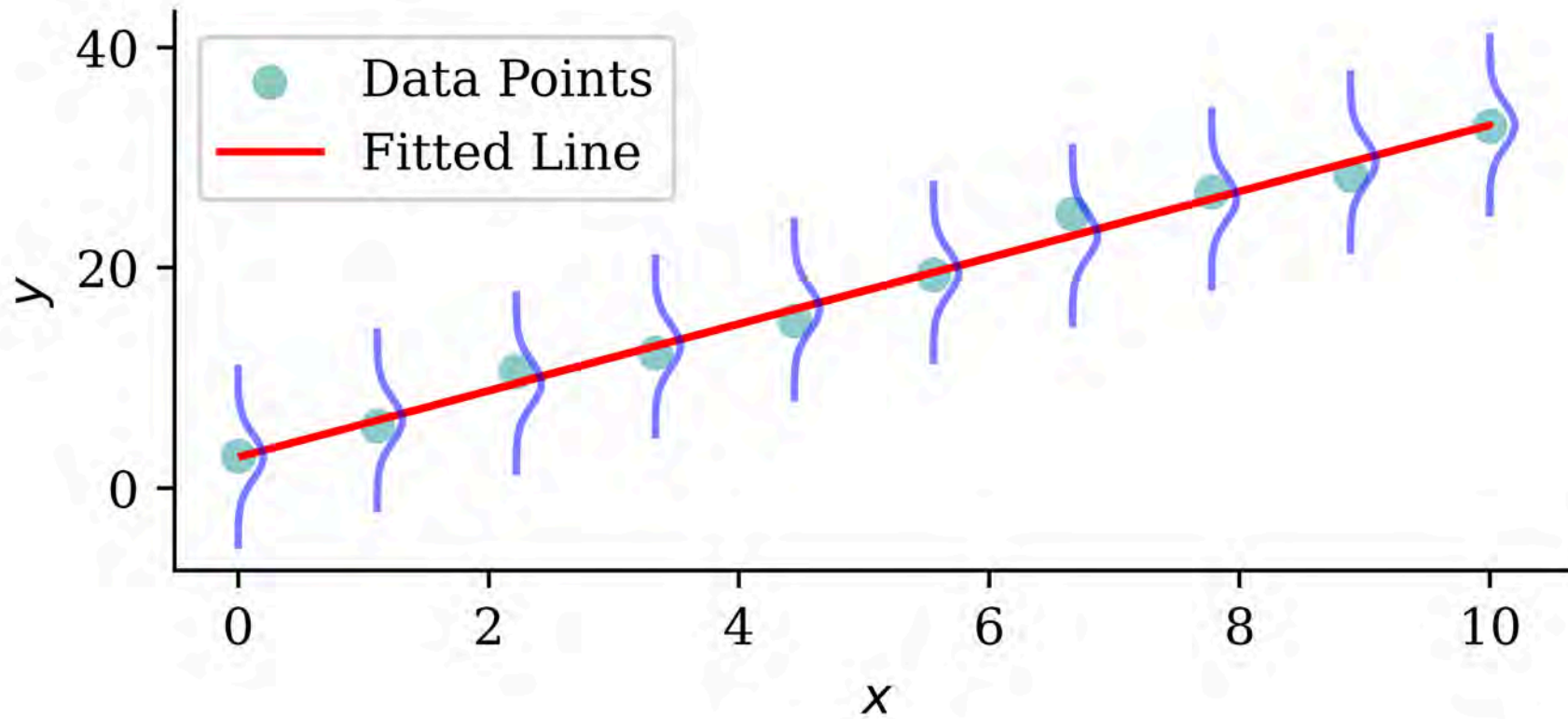
where $\varepsilon \sim \mathcal{N}(0, \sigma^2)$.

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 σ^2 is known.

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

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

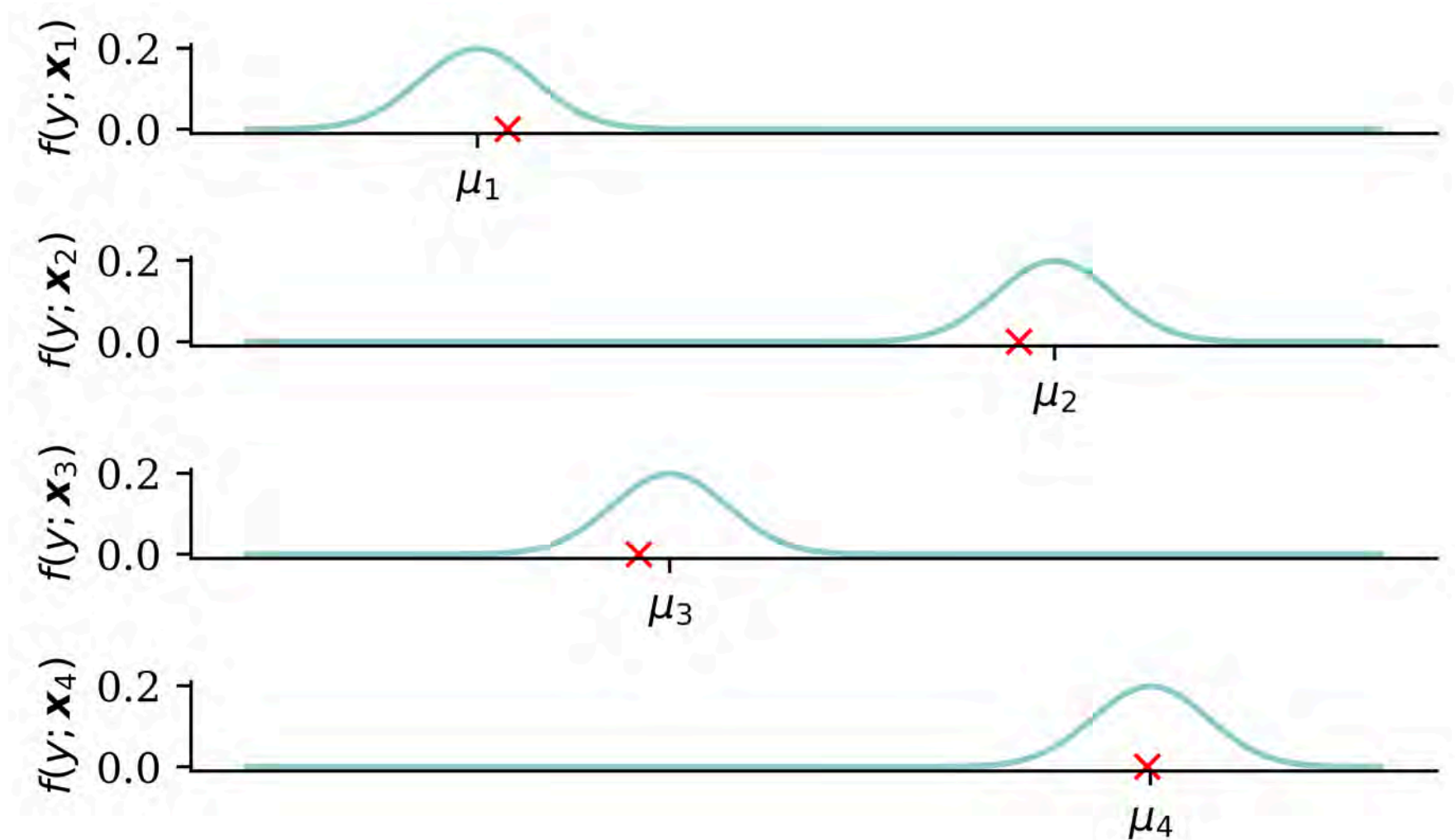
The likelihood function is

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

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

Perform maximum likelihood estimation to find $\boldsymbol{\beta}$.

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 σ^2 is fixed, minimising NLL is equivalent to minimising MSE:

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

Generalised Linear Model (GLM)

The GLM is often characterised by the mean prediction:

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

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 basically identical:

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

Poisson regression

A Poisson distribution with rate λ 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

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

Gamma regression

A Gamma distribution with mean μ and dispersion ϕ 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 ϕ 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 ϕ 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?

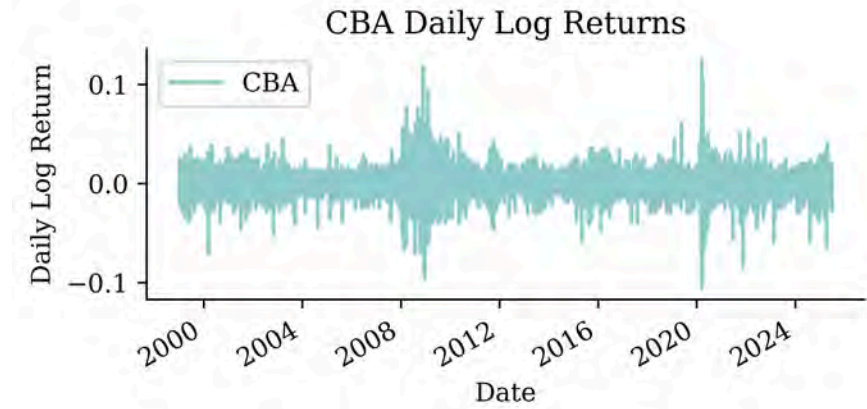
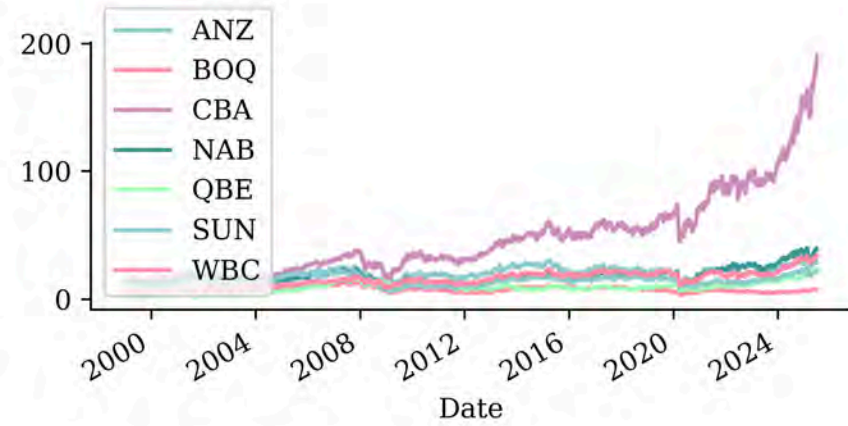
- GLMs are interpretable.
- GLMs are flexible (can handle different types of response variables).
- We get the full distribution of the response variable, not just the mean.

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

Lecture Outline

- Introduction
- Traditional Regression
- **Stochastic Forecasts**
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- Ensembles

Stock price forecasting



Noisy auto-regressive forecast

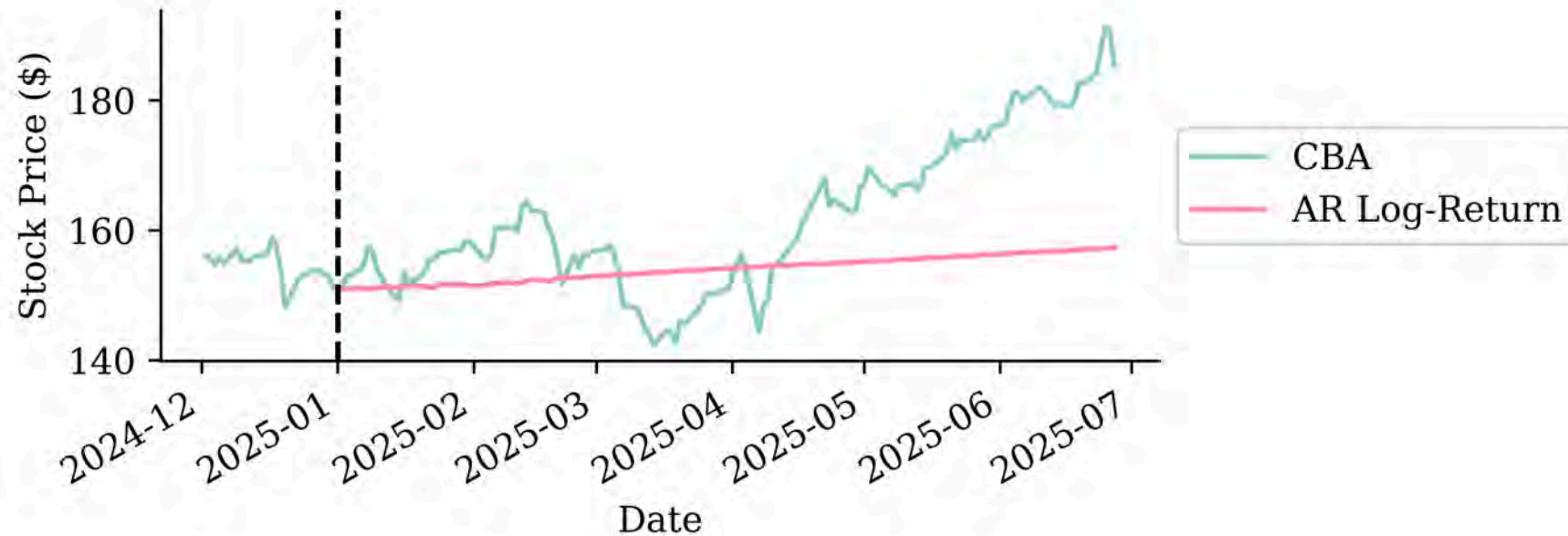
```

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)

```

Original forecast

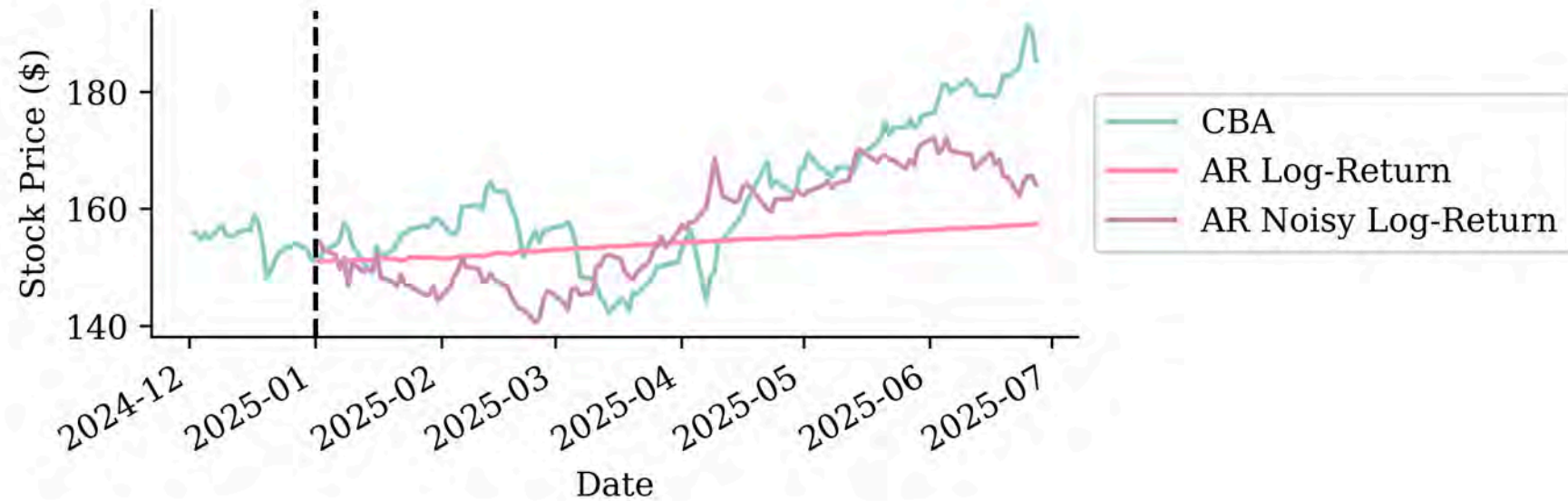
```
1 lr_forecast = noisy_autoregressive_forecast(lr, X_test, 0)
2 last_price = get_last_price(stock, cutoff_date="2024-12")
3 price_forecast = log_to_price(lr_forecast, last_price)
```



```
1 residuals = y_train.loc["2015":] - lr.predict(X_train.loc["2015":])
2 sigma = np.std(residuals)
```

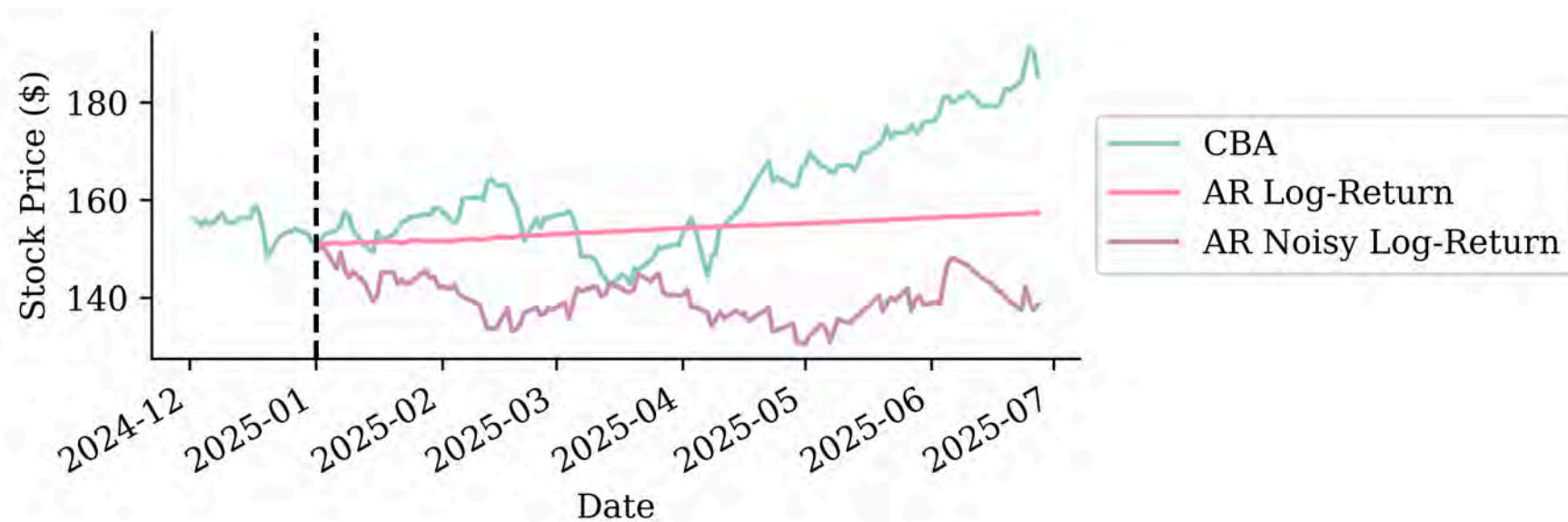
With noise

```
1 np.random.seed(1)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_test, sigma)
3 price_noisy = log_to_price(lr_noisy_forecast, last_price)
```



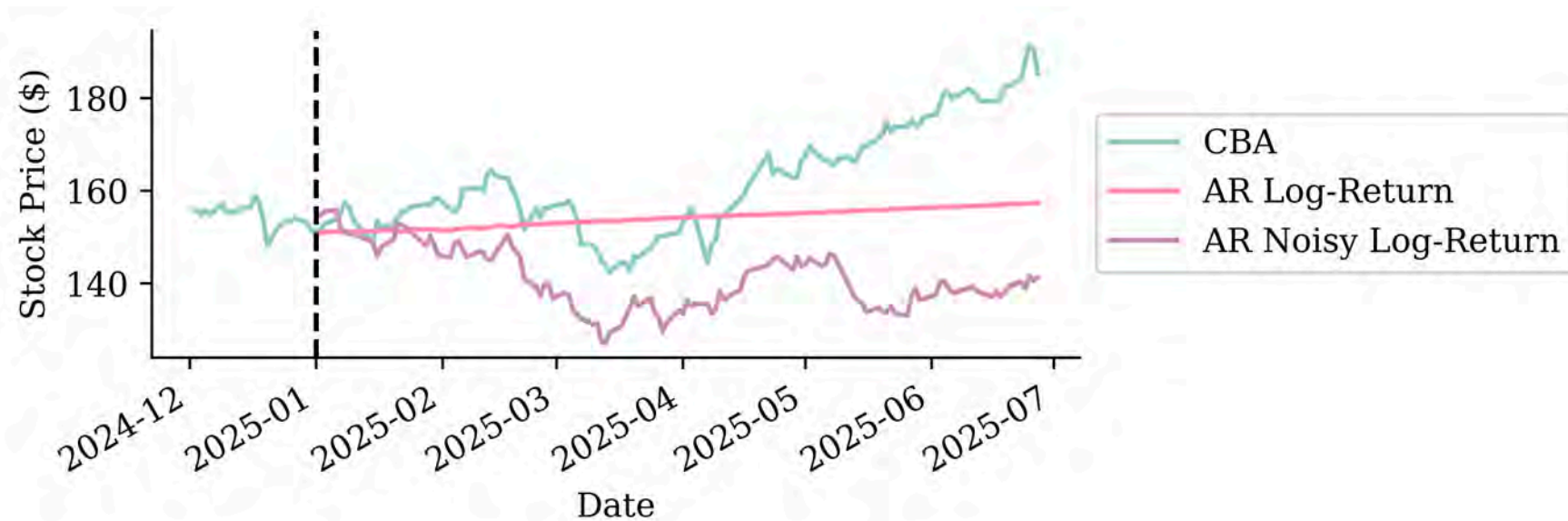
With noise

```
1 np.random.seed(2)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_test, sigma)
3 price_noisy = log_to_price(lr_noisy_forecast, last_price)
```



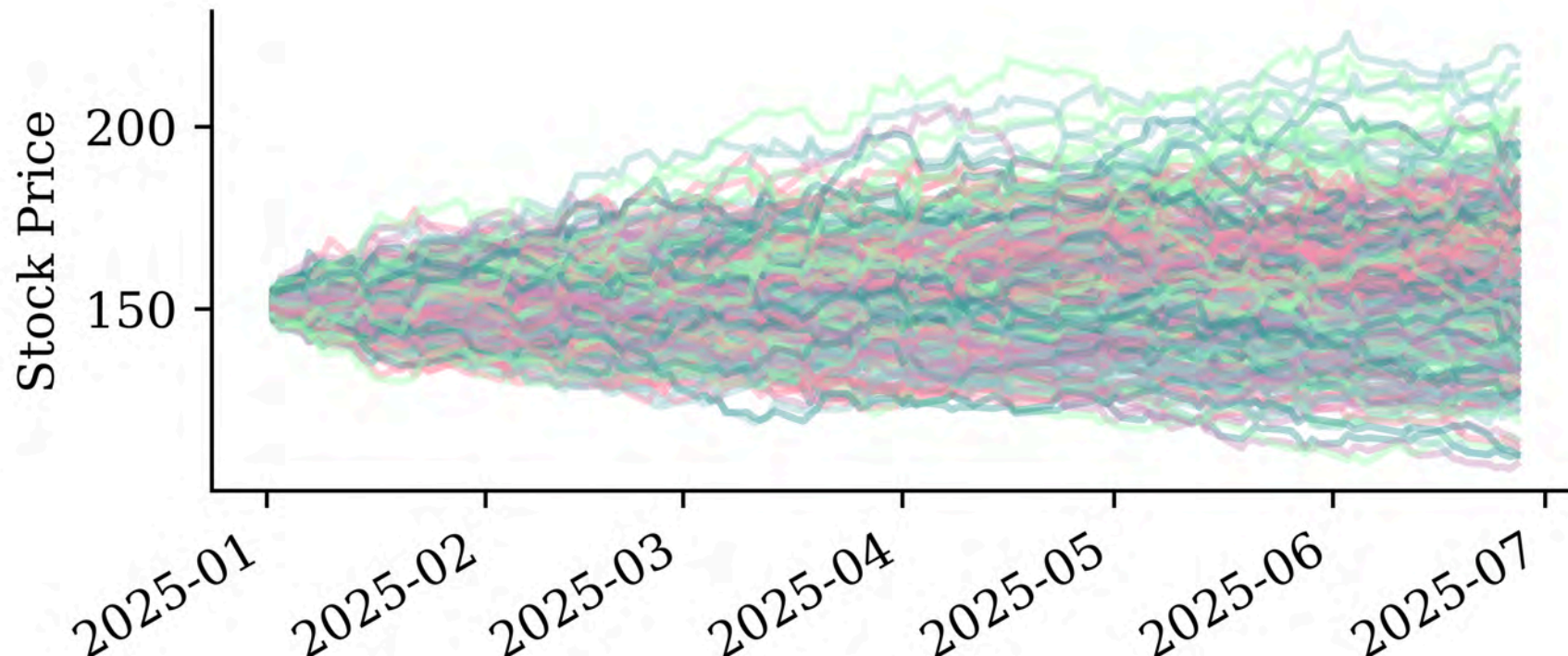
With noise

```
1 np.random.seed(3)
2 lr_noisy_forecast = noisy_autoregressive_forecast(lr, X_test, sigma)
3 price_noisy = log_to_price(lr_noisy_forecast, last_price)
```



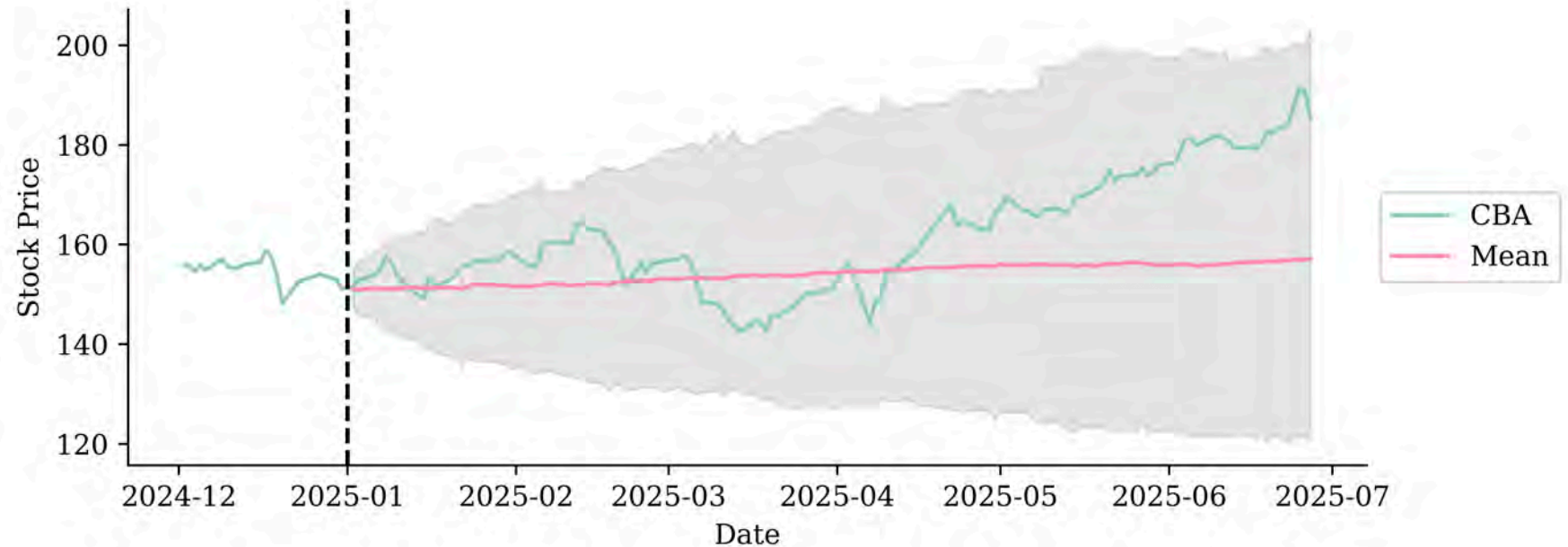
Many noisy forecasts

```
1 num_forecasts = 300
2 forecasts = []
3 for i in range(num_forecasts):
4     log_sim = noisy_autoregressive_forecast(lr, X_test, sigma)
5     price_sim = log_to_price(log_sim, last_price)
6     forecasts.append(price_sim)
7 noisy_forecasts = pd.concat(forecasts, axis=1)
8 noisy_forecasts.index = X_test.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)
```



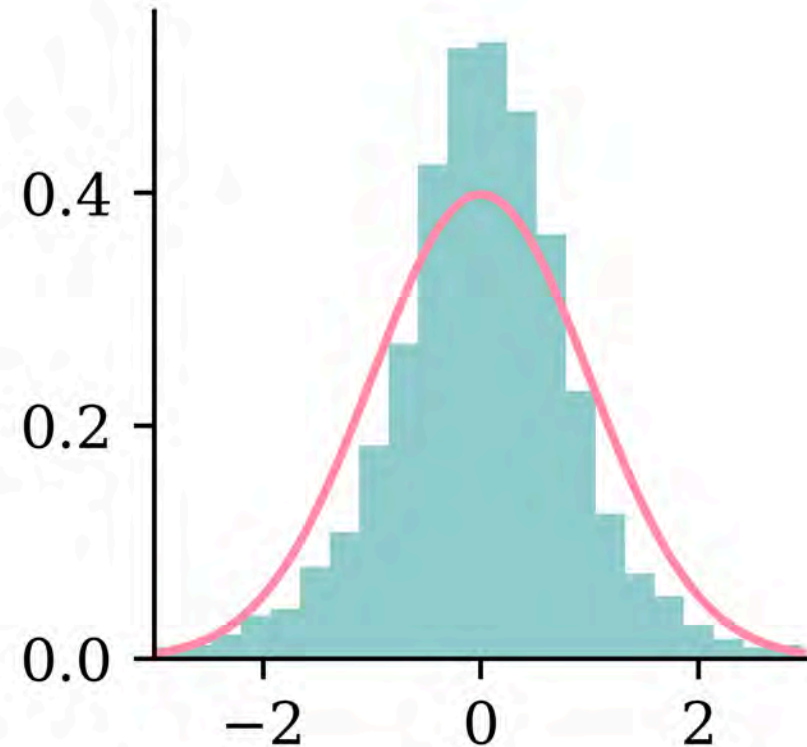
Residuals

```
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)
```

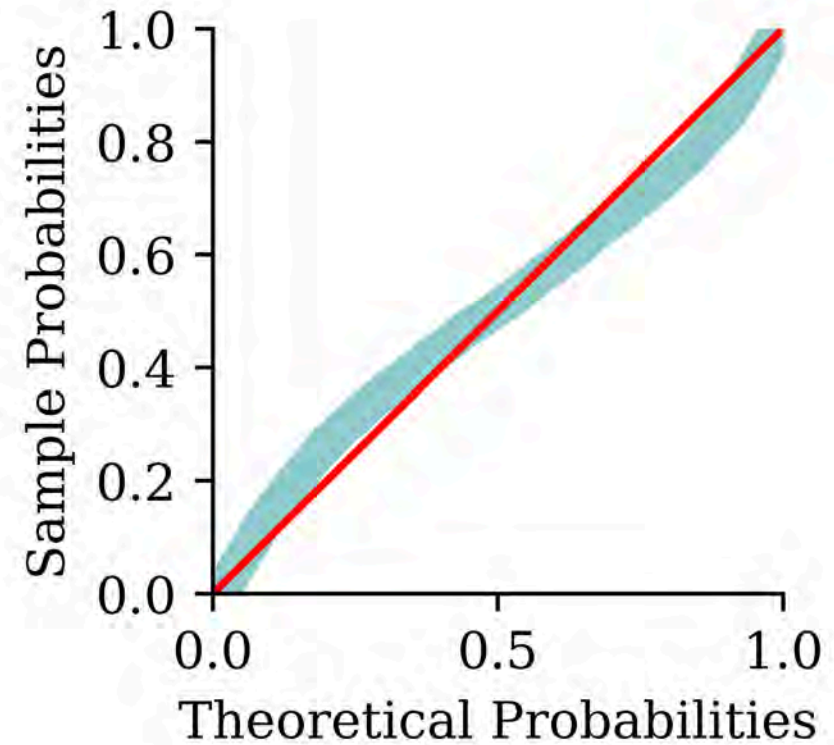
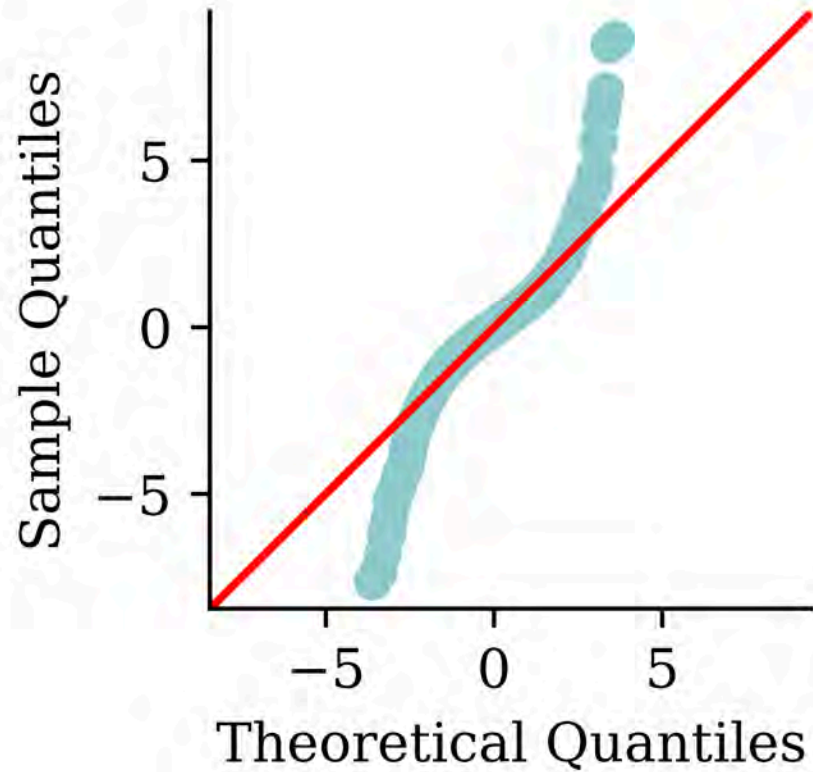
```
/Users/z3535837/Library/CloudStorage/Dropbox/Le
packages/scipy/stats/_axis_nan_policy.py:592:
UserWarning: scipy.stats.shapiro: For N >
5000, computed p-value may not be accurate.
Current N is 6582.
```

```
res = hypotest_fun_out(*samples, **kwds)
```

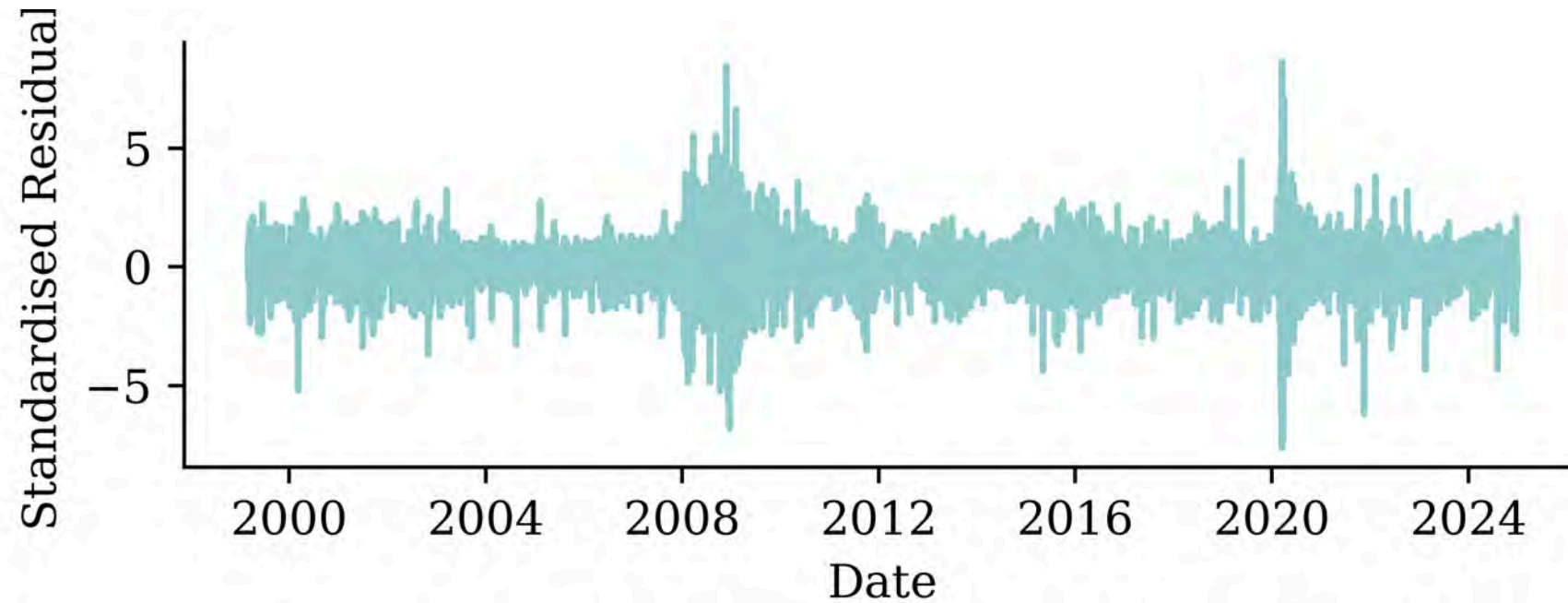
```
ShapiroResult(statistic=np.float64(0.9305463633
pvalue=np.float64(5.938320016173568e-48))
```



Q-Q plot and P-P plot

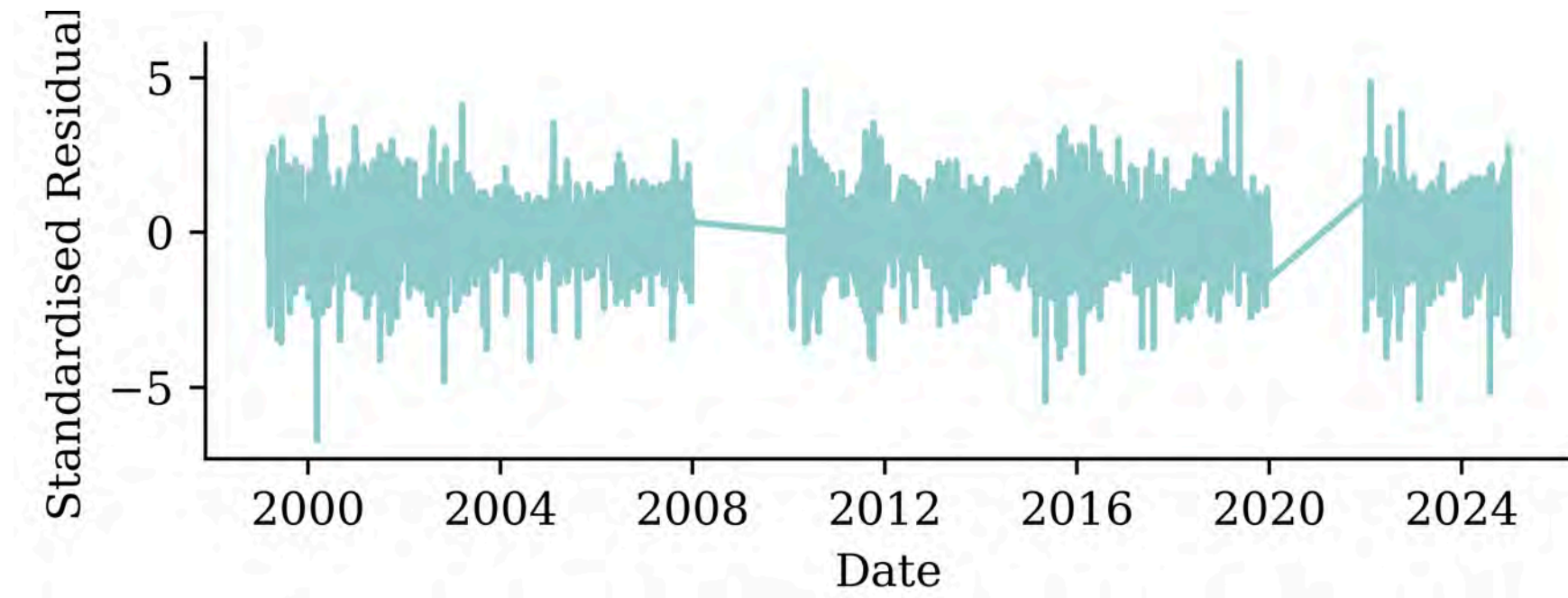


Residuals against time



Heteroskedasticity!

Retrain on less data

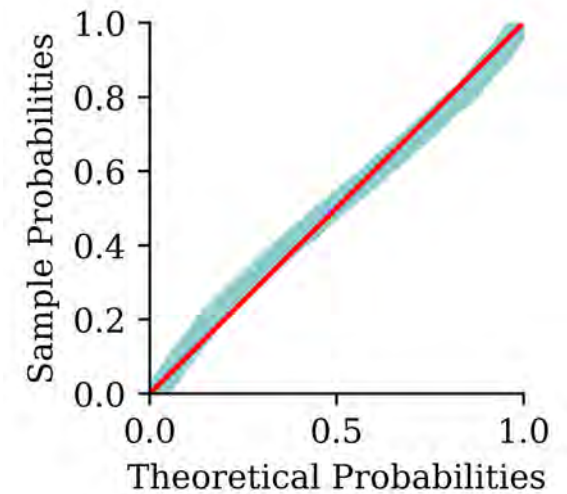
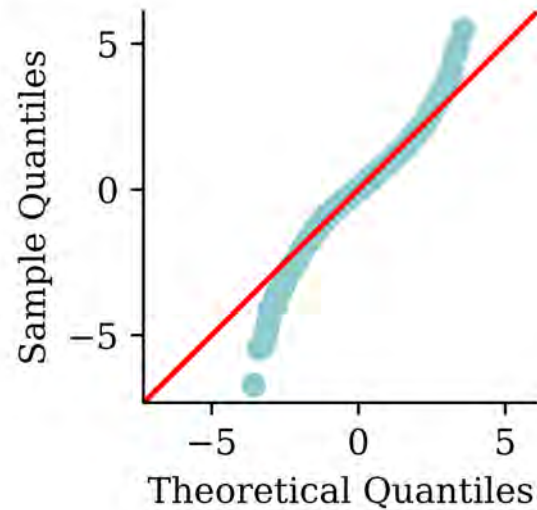
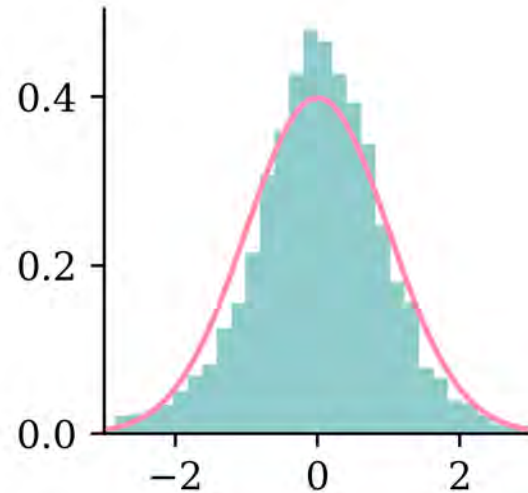


Refit the model without these crisis years.

Residual diagnostics

```
1 stats.shapiro(res2)
```

```
ShapiroResult(statistic=np.float64(0.9797545738507011), pvalue=np.float64(1.858481775102687e-27))
```



Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- **GLMs and Neural Networks**
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- Ensembles

French motor claim sizes

```

1 sev = pd.read_csv(' ../data/freMTPL2sev.csv')
2 cov = pd.read_csv(' ../data/freMTPL2freq.csv').drop(columns=['ClaimNb'])
3 sev = pd.merge(sev, cov, on='IDpol', how='left').drop(columns=["IDpol"]).dropna()
4 sev

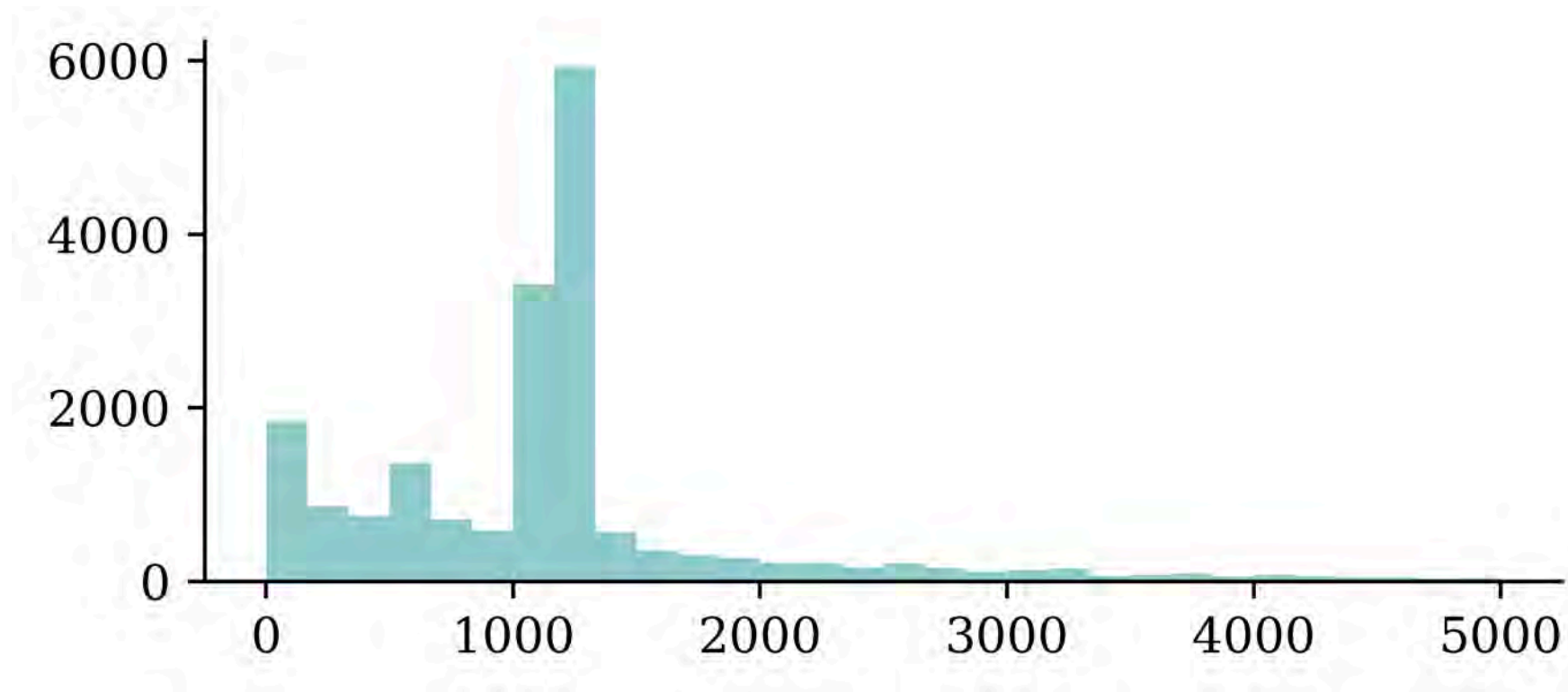
```

| | ClaimAmount | Exposure | VehPower | VehAge | DrivAge | BonusMalus | Veh |
|--------------|-------------|----------|----------|--------|---------|------------|-----|
| 0 | 995.20 | 0.59 | 11.0 | 0.0 | 39.0 | 56.0 | B12 |
| 1 | 1128.12 | 0.95 | 4.0 | 1.0 | 49.0 | 50.0 | B12 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 26637 | 767.55 | 0.43 | 6.0 | 0.0 | 67.0 | 50.0 | B2 |
| 26638 | 1500.00 | 0.28 | 7.0 | 2.0 | 36.0 | 60.0 | B12 |

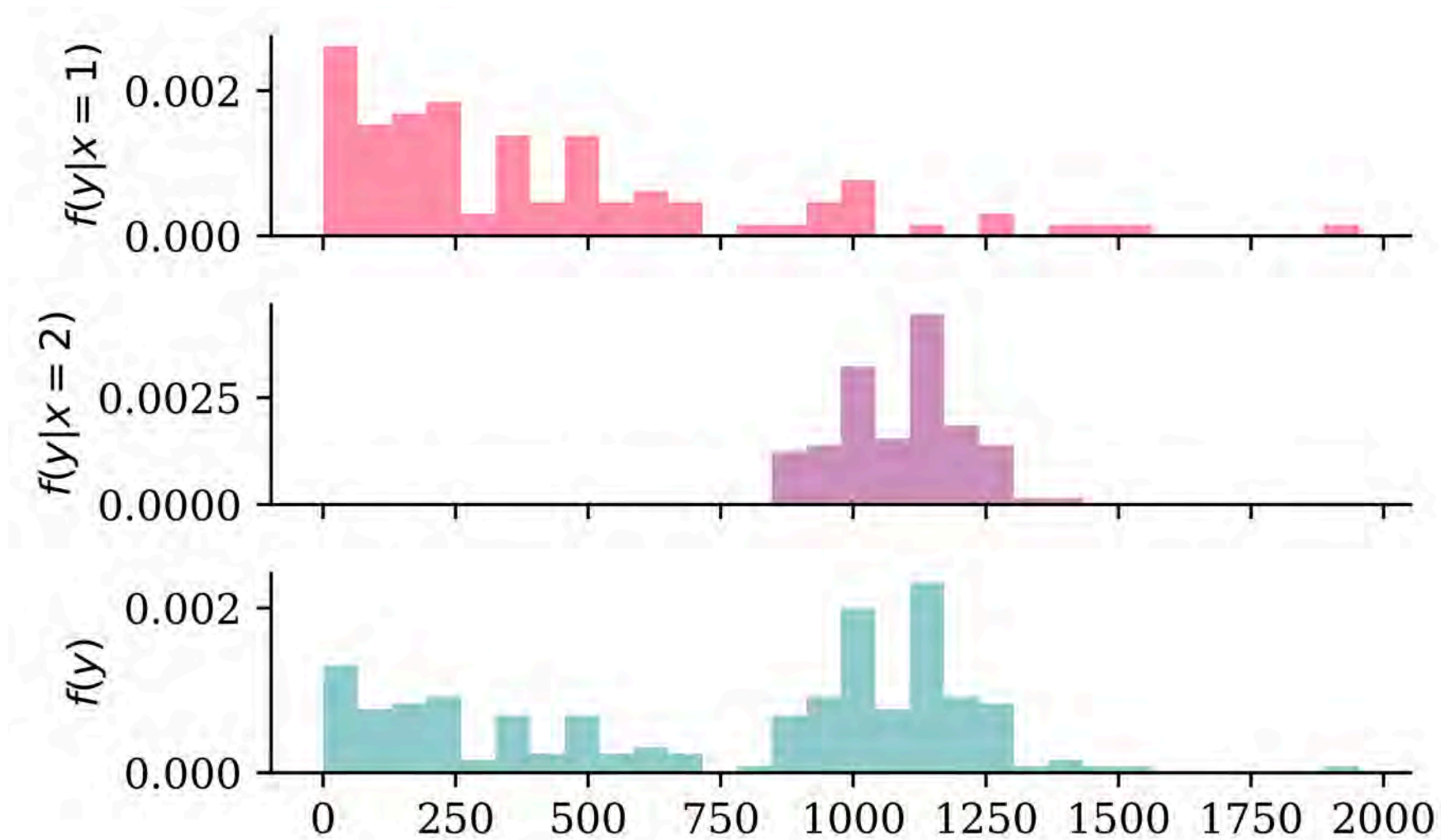
26444 rows × 11 columns

Preprocessing

```
1 X_train, X_test, y_train, y_test = train_test_split(  
2     sev.drop("ClaimAmount", axis=1), sev["ClaimAmount"], random_state=2023)  
3 ct = make_column_transformer((OrdinalEncoder(), ["Area", "VehGas"]),  
4     ("drop", ["VehBrand", "Region"]), remainder=StandardScaler())  
5 X_train = ct.fit_transform(X_train)  
6 X_test = ct.transform(X_test)  
7 plt.hist(y_train[y_train < 5000], bins=30);
```



Doesn't prove that $Y|X = x$ is multimodal



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 $\boldsymbol{x} = (1, x_1, x_2, x_3)$ as follows:

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

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 ϕ , 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 ϕ while estimating the regression coefficients.

Fitting Steps

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

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

Step 2. Estimate the dispersion parameter:

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

(Here, p is the number of coefficients in the model. If this p doesn't include the intercept, then the scaling should be $\frac{1}{n-(p+1)}$.)

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
ordinalencoder__Area -0.073226
...
remainder__BonusMalus 0.157204
remainder__Density    0.010539
Length: 9, dtype: float64

```

```

1 # Dispersion Parameter
2 mus = gamma_glm.predict(X_train_design)
3 residuals = y_train - mus
4 dof = (len(y_train)-X_train_design.shape[0])
5 phi_glm = np.sum(residuals**2/mus**2)/dof
6 print(phi_glm)

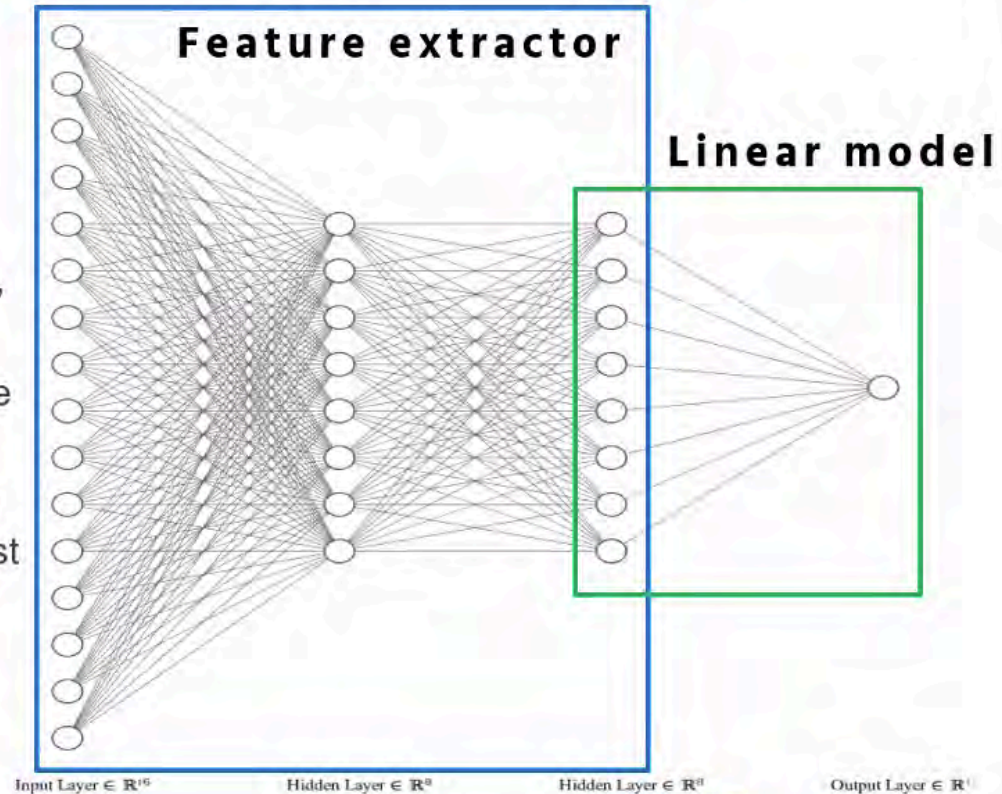
```

```
59.63363123735744
```

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



Institute
and Faculty
of Actuaries

11 April 2022

14

Combining GLM & ANN.

Source: Ronald Richman (2022), *Mind the Gap — Safely Incorporating Deep Learning Models into the Actuarial Toolkit*, IFoA seminar, Slide 14.



UNSW
SYDNEY

Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- **Combined Actuarial Neural Network**
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- Ensembles

CANN

The Combined Actuarial Neural Network is an actuarial neural network architecture proposed by [Schelldorfer and Wüthrich \(2019\)](#). We summarise the CANN approach as follows:

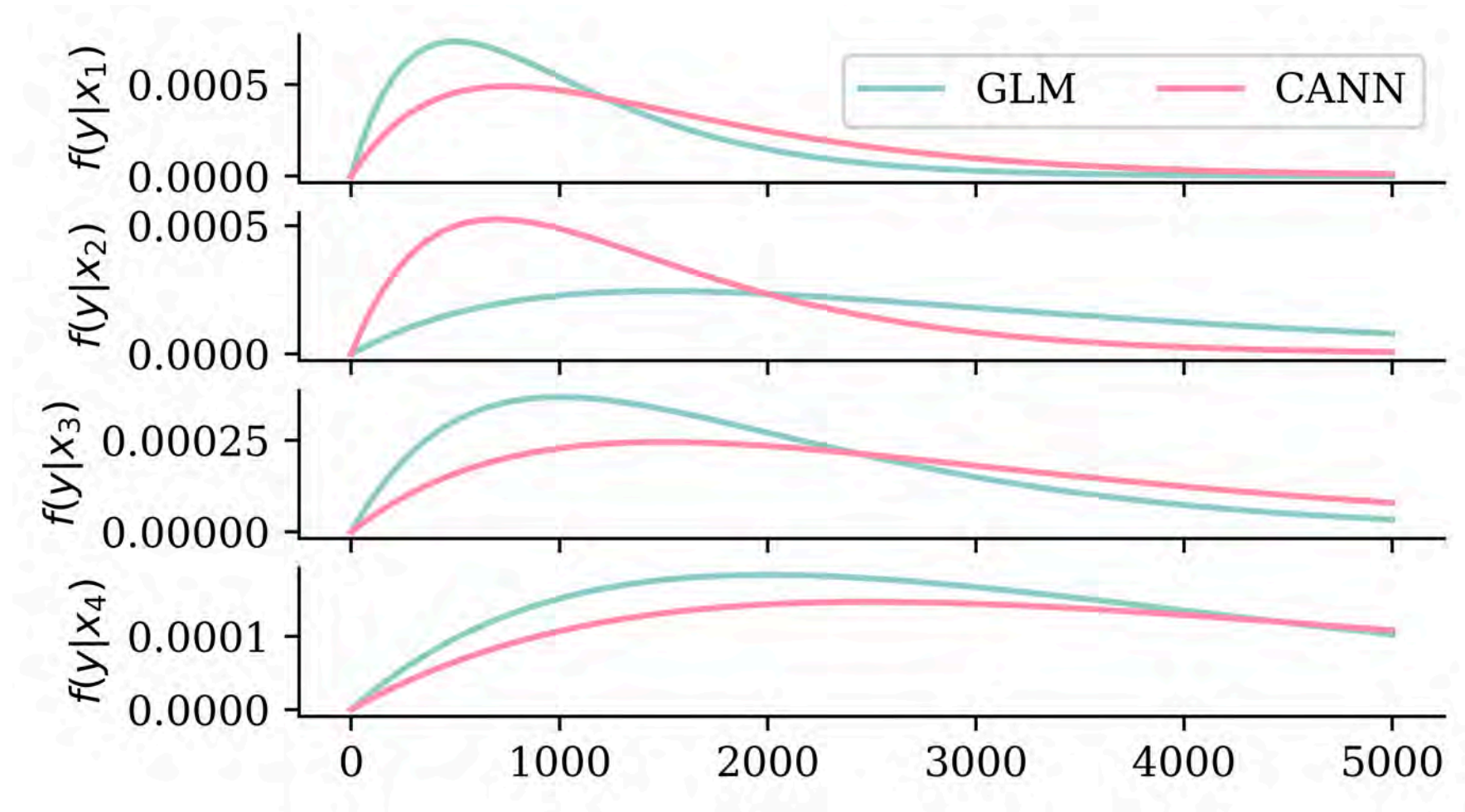
- Find coefficients β of the GLM with a link function $g(\cdot)$.
- Find weights $\mathbf{w} = (\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(K+1)})$ of a neural network $s : \mathbb{R}^p \rightarrow \mathbb{R}$.
- Given a new instance \mathbf{x} , we have

$$\mathbb{E}[Y | \mathbf{X} = \mathbf{x}] = g^{-1} \left(\langle \beta, \mathbf{x} \rangle + s(\mathbf{x}; \mathbf{w}) \right).$$

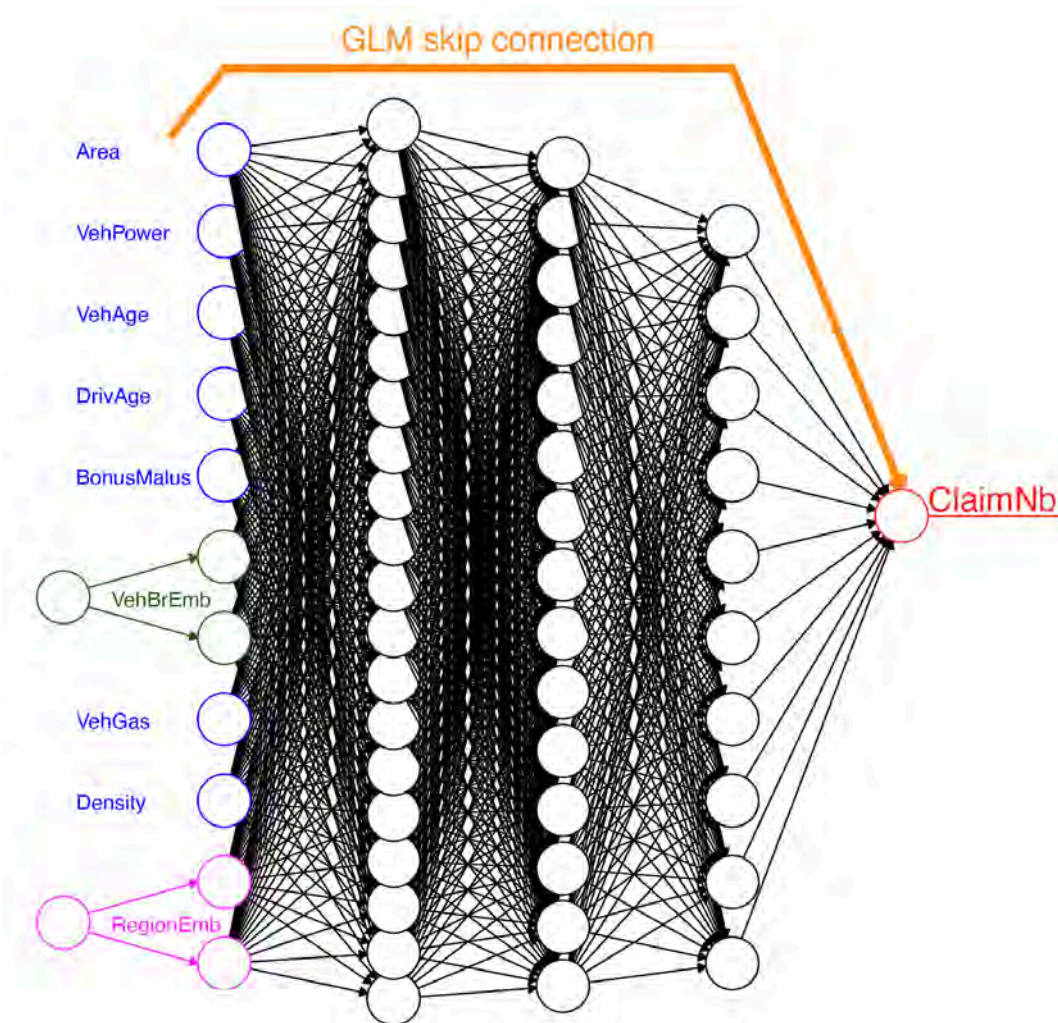
If just a sequential dense network, then

$$\mathbb{E}[Y | \mathbf{X} = \mathbf{x}] = g^{-1} \left(\langle \beta, \mathbf{x} \rangle + \left\langle \mathbf{w}^{(K+1)}, (\mathbf{z}^{(K)} \circ \dots \circ \mathbf{z}^{(1)})(\mathbf{x}) \right\rangle \right).$$

Shifting the predicted distributions



Architecture



The CANN architecture.

Source: Schelldorfer and Wüthrich (2019), [Nesting Classical Actuarial Models into Neural Networks](#), SSRN, Figure 8.

Code: Architecture

```

1 random.seed(1)
2 inputs = Input(shape=X_train.shape[1:])
3
4 # GLM part (won't be updated during training)
5 glm_weights = gamma_glm.params.iloc[1:].values.reshape((-1, 1))
6 glm_bias = gamma_glm.params.iloc[0]
7 glm_part = Dense(1, activation='linear', trainable=False,
8                 kernel_initializer=Constant(glm_weights),
9                 bias_initializer=Constant(glm_bias))(inputs)
10
11 # Neural network part
12 x = Dense(64, activation='leaky_relu')(inputs)
13 nn_part = Dense(1, activation='linear')(x)
14
15 # Combine GLM and CANN estimates
16 mu = keras.ops.exp(glm_part + nn_part)
17 cann = Model(inputs, mu)

```

Since this CANN predicts Gamma distributions, we use the Gamma NLL loss function.

```

1 def cann_negative_log_likelihood(y_true, y_pred):
2     return keras.ops.mean(keras.ops.log(y_pred) + y_true/y_pred)

```

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 = cann.predict(X_train, verbose=0).flatten()
2 residuals = y_train - mus
3 dof = (len(y_train)-(X_train.shape[1] + 1))
4 phi_cann = np.sum(residuals**2/mus**2) / dof
5 print(phi_cann)
```

19.928278221047712

Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- **Mixture Density Network**
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- 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 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_k and F_k 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 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 distribution 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) $\text{MDN}(\mathbf{x})$ outputs each distribution component's mixing weights and parameters of Y given the input features \mathbf{x} , i.e.,

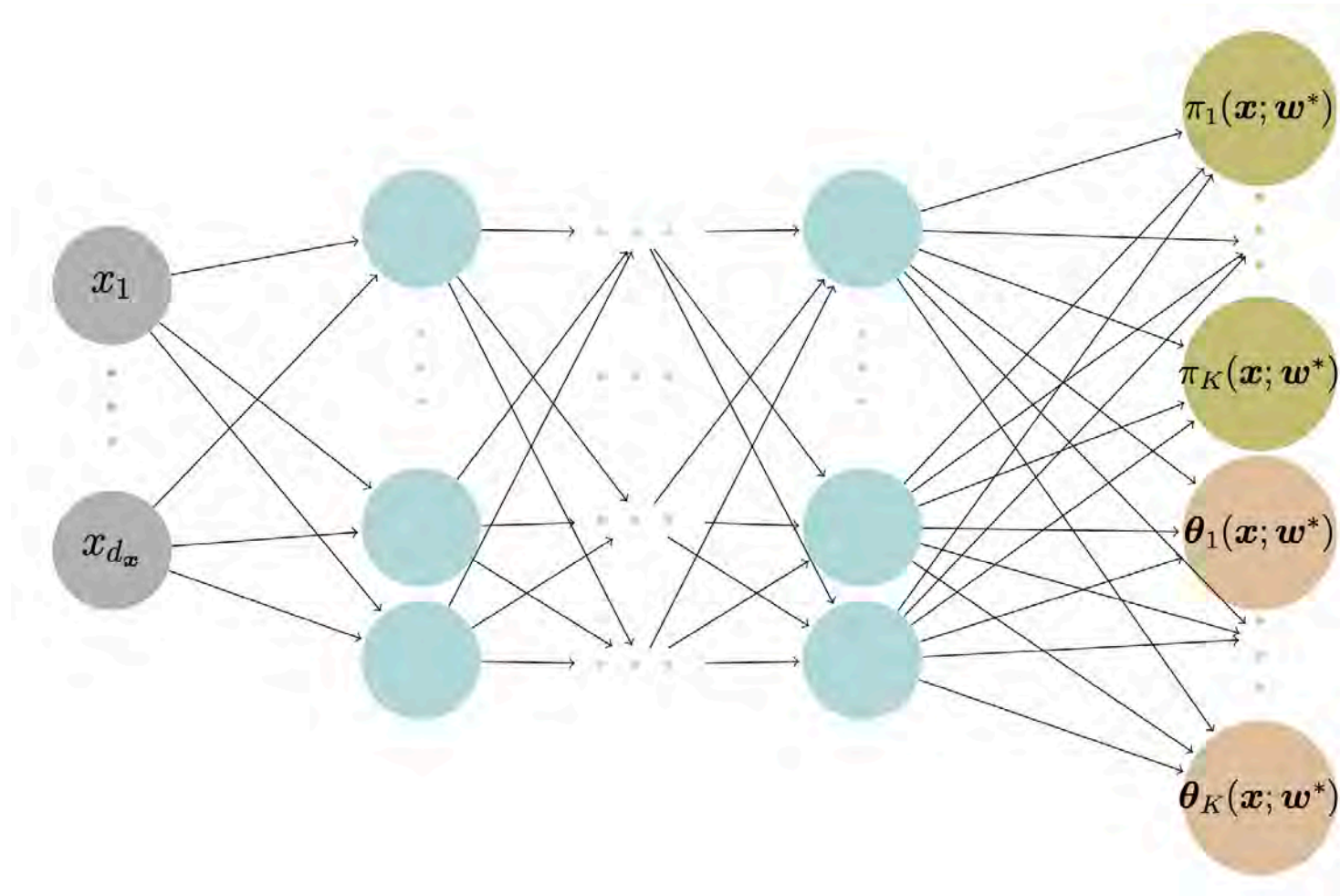
$$\text{MDN}(\mathbf{x}) = (\boldsymbol{\pi}(\mathbf{x}; \mathbf{w}^*), \boldsymbol{\theta}(\mathbf{x}; \mathbf{w}^*)),$$

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

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

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. $\theta_k(\mathbf{x}; \mathbf{w}^*) = (\theta_{k,1}(\mathbf{x}; \mathbf{w}^*), \dots, \theta_{k,|\theta_k|}(\mathbf{x}; \mathbf{w}^*))$ 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

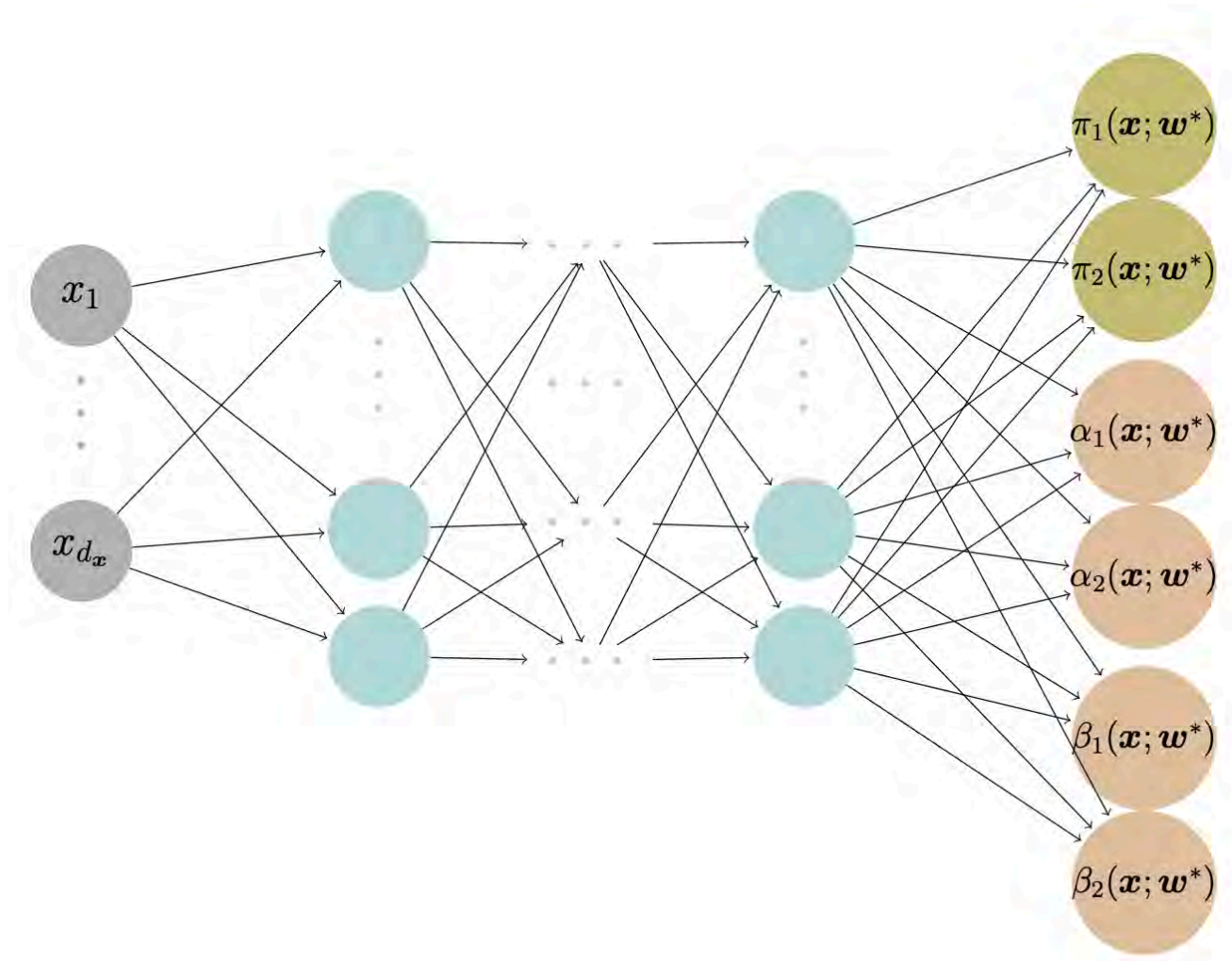
The aim is to find the optimum weights

$$\mathbf{w}^* = \arg \min_w \mathcal{L}(\mathcal{D}, \mathbf{w})$$

for the Gamma mixture density network $\text{MDN}(\mathbf{x})$ that outputs the mixing weights, shapes and scales of Y given the input features \mathbf{x} , i.e.,

$$\begin{aligned} \text{MDN}(\mathbf{x}) = & (\pi_1(\mathbf{x}; \mathbf{w}^*), \pi_2(\mathbf{x}; \mathbf{w}^*), \\ & \alpha_1(\mathbf{x}; \mathbf{w}^*), \alpha_2(\mathbf{x}; \mathbf{w}^*), \\ & \beta_1(\mathbf{x}; \mathbf{w}^*), \beta_2(\mathbf{x}; \mathbf{w}^*)). \end{aligned}$$

Architecture



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

Code: Architecture

The following code implements the gamma MDN from the previous slide.

```
1 import torch
2 import torch.nn as nn
3
4 random.seed(1)
5 torch.manual_seed(1)
6
7 class GammaMDN(nn.Module):
8     def __init__(self, input_dim, hidden_dim=64, n_components=2):
9         super().__init__()
10        self.hidden = nn.Sequential(
11            nn.Linear(input_dim, hidden_dim),
12            nn.ReLU(),
13            nn.Linear(hidden_dim, hidden_dim),
14            nn.ReLU(),
15        )
16        self.pi_layer = nn.Linear(hidden_dim, n_components)
17        self.alpha_layer = nn.Linear(hidden_dim, n_components)
18        self.beta_layer = nn.Linear(hidden_dim, n_components)
19
20    def forward(self, x):
21        h = self.hidden(x)
22        pis = torch.softmax(self.pi_layer(h), dim=-1)
23        alphas = torch.exp(self.alpha_layer(h))
24        betas = torch.exp(self.beta_layer(h))
```

Loss Function

The negative log-likelihood loss function is given by

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

where the $f_{Y|X}(y_i | \mathbf{x}, \mathbf{w})$ 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 & training

`torch.distributions` to the rescue.

```

1 from torch.distributions import Gamma, MixtureSameFamily, Categorical
2
3 def gamma_mixture_nll(pis, alphas, betas, y_true):
4     mixture = MixtureSameFamily(
5         mixture_distribution=Categorical(probs=pis),
6         component_distribution=Gamma(concentration=alphas, rate=betas))
7     return -mixture.log_prob(y_true).mean()

```

```

1 # Convert data to PyTorch tensors
2 X_train_t = torch.tensor(X_train.values, dtype=torch.float32)
3 y_train_t = torch.tensor(y_train.values, dtype=torch.float32)
4
5 # Split into train and validation (80/20)
6 n_val = int(0.2 * len(X_train_t))
7 X_val_mdn, y_val_mdn = X_train_t[-n_val:], y_train_t[-n_val:]
8 X_trn_mdn, y_trn_mdn = X_train_t[:-n_val], y_train_t[:-n_val]
9
10 optimizer = torch.optim.Adam(gamma_mdn.parameters())
11 loader = torch.utils.data.DataLoader(
12     torch.utils.data.TensorDataset(X_trn_mdn, y_trn_mdn),
13     batch_size=64, shuffle=True)
14
15 best_val_loss, patience_counter, best_state = float('inf'), 0, None
16
17 for epoch in range(100):

```

Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- **Metrics for Distributional Regression**
- Aleatoric and Epistemic Uncertainty
- Dropout
- Ensembles

Proper Scoring Rules

Definition

A *scoring rule* is the equivalent of a loss function for distributional regression.

Denote $S(F, y)$ to be the score given to the forecasted distribution F and an observation $y \in \mathbb{R}$.

Definition

A scoring rule is called *proper* if

$$\mathbb{E}_{Y \sim Q} S(Q, Y) \leq \mathbb{E}_{Y \sim Q} S(F, Y)$$

for all F and Q distributions.

It is called *strictly proper* if equality holds only if $F = Q$.

Example 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.

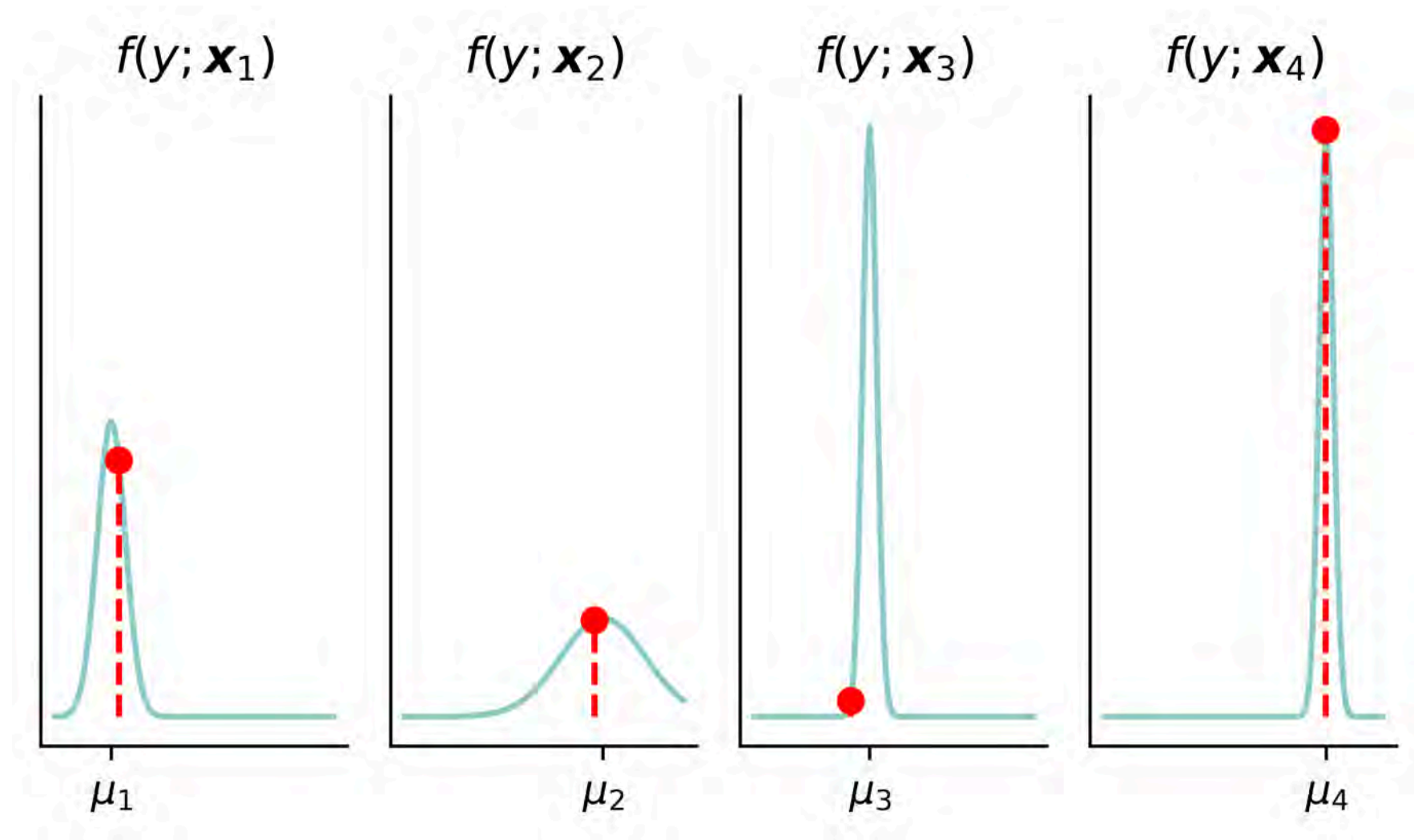
Continuous Ranked Probability Score (CRPS)

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 predicted c.d.f.

Likelihoods



Code: NLL

```

1  def gamma_nll(mean, dispersion, y):
2      # Calculate shape and scale parameters from mean and dispersion
3      shape = 1 / dispersion; scale = mean * dispersion
4
5      # Create a gamma distribution object
6      gamma_dist = stats.gamma(a=shape, scale=scale)
7
8      return -np.mean(gamma_dist.logpdf(y))
9
10 # GLM
11 X_test_design = sm.add_constant(X_test)
12 mus = gamma_glm.predict(X_test_design)
13 nll_glm = gamma_nll(mus, phi_glm, y_test)
14
15 # CANN
16 mus = cann.predict(X_test, verbose=0)
17 nll_cann = gamma_nll(mus, phi_cann, y_test)
18
19 # MDN
20 gamma_mdn.eval()
21 with torch.no_grad():
22     X_test_t = torch.tensor(X_test.values, dtype=torch.float32)
23     y_test_t = torch.tensor(y_test.values, dtype=torch.float32)
24     pis, alphas, betas = gamma_mdn(X_test_t)

```

```

1  print(f'GLM: {round(nll_glm, 2)}')
2  print(f'CANN: {round(nll_cann, 2)}')

```



Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- **Aleatoric and Epistemic Uncertainty**
- Dropout
- Ensembles

Categories of uncertainty

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

- Aleatoric uncertainty: the inherent variability associated with the data generating process.
- Epistemic uncertainty: the lack of knowledge, limited data information, parameter errors and model errors.

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.

Traditional regularisation

Say all the m weights (excluding biases) are in the vector $\boldsymbol{\theta}$. 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, activation="leaky_relu",
7             kernel_regularizer=L1(regulariser_strength)),
8         Dense(1, activation="exponential")
9     ])
10
11     model.compile("adam", "mse")
12     model.fit(X_train_sc, y_train, epochs=4, verbose=0)
13     return model
14
15 def l2_model(regulariser_strength=0.01):
16     random.seed(123)
17     model = Sequential([
18         Dense(30, activation="leaky_relu",
19             kernel_regularizer=L2(regulariser_strength)),
20         Dense(1, activation="exponential")
21     ])
22
23     model.compile("adam", "mse")
24     model.fit(X_train_sc, y_train, epochs=10, verbose=0)
```

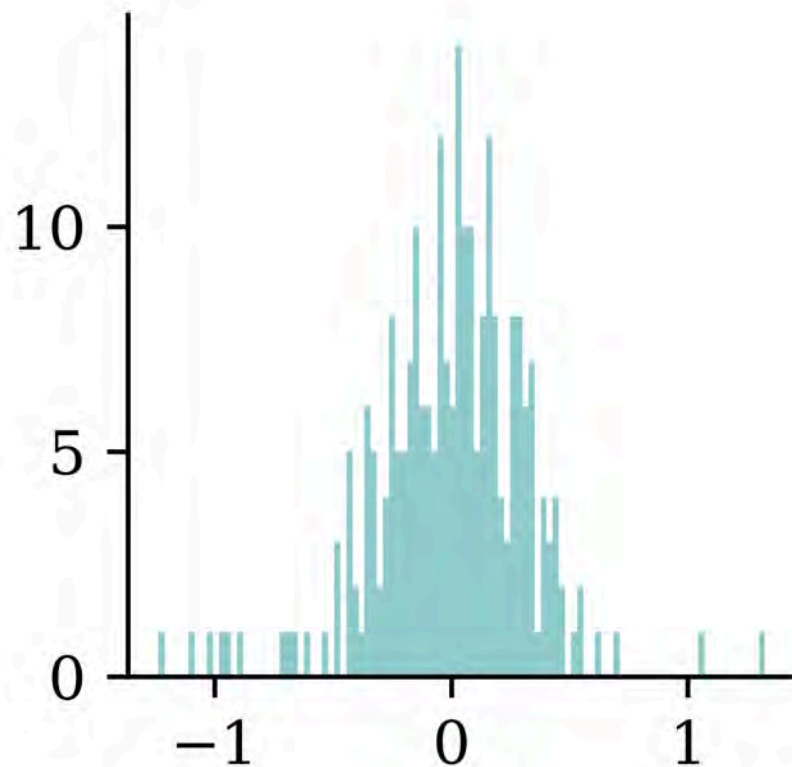
Weights before & after L^2

```

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

```

Number of weights almost 0: 0

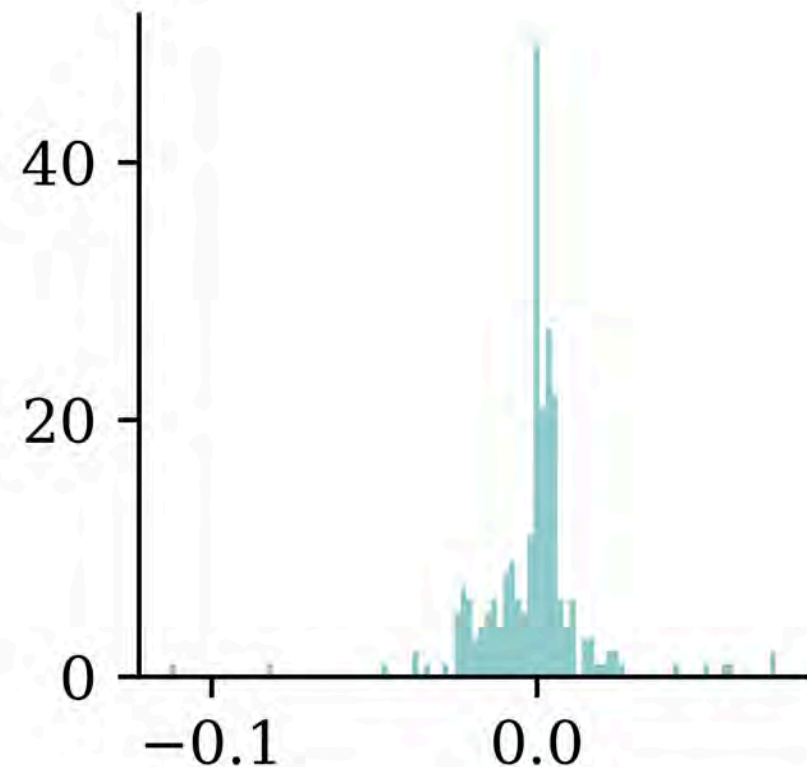


```

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

```

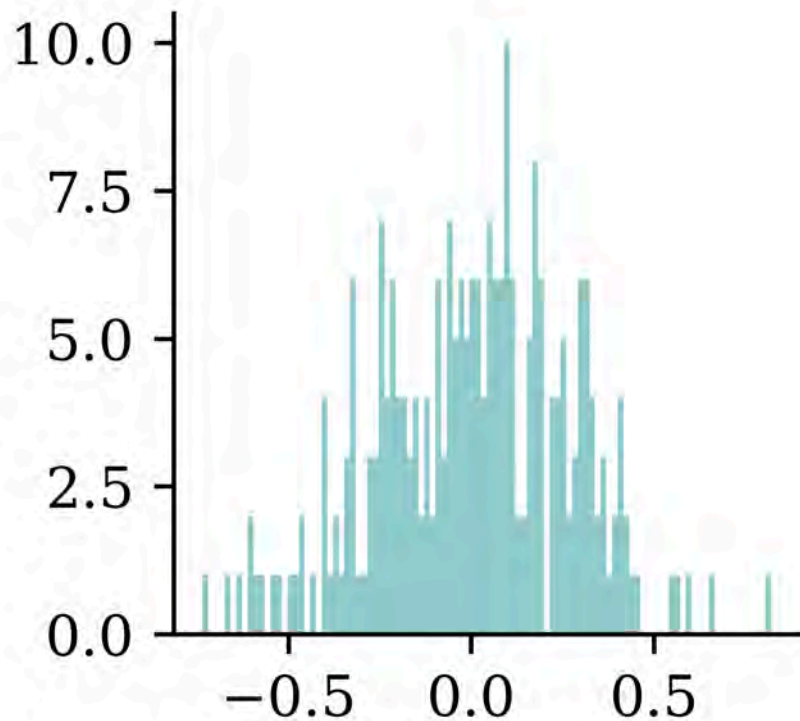
Number of weights almost 0: 0



Weights before & after L^1

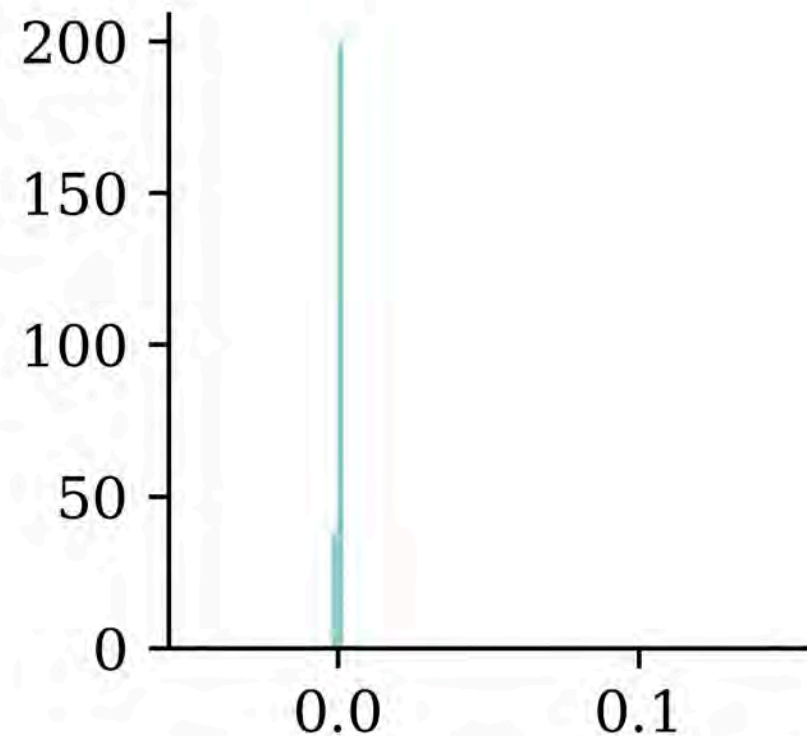
```
1 model = l1_model(0.0)
2 weights = model.layers[0].get_weights()[0]
3 print(f"Number of weights almost 0: {np.sum(weights == 0)}")
4 plt.hist(weights, bins=100);
```

Number of weights almost 0: 0



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

Number of weights almost 0: 16



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”.

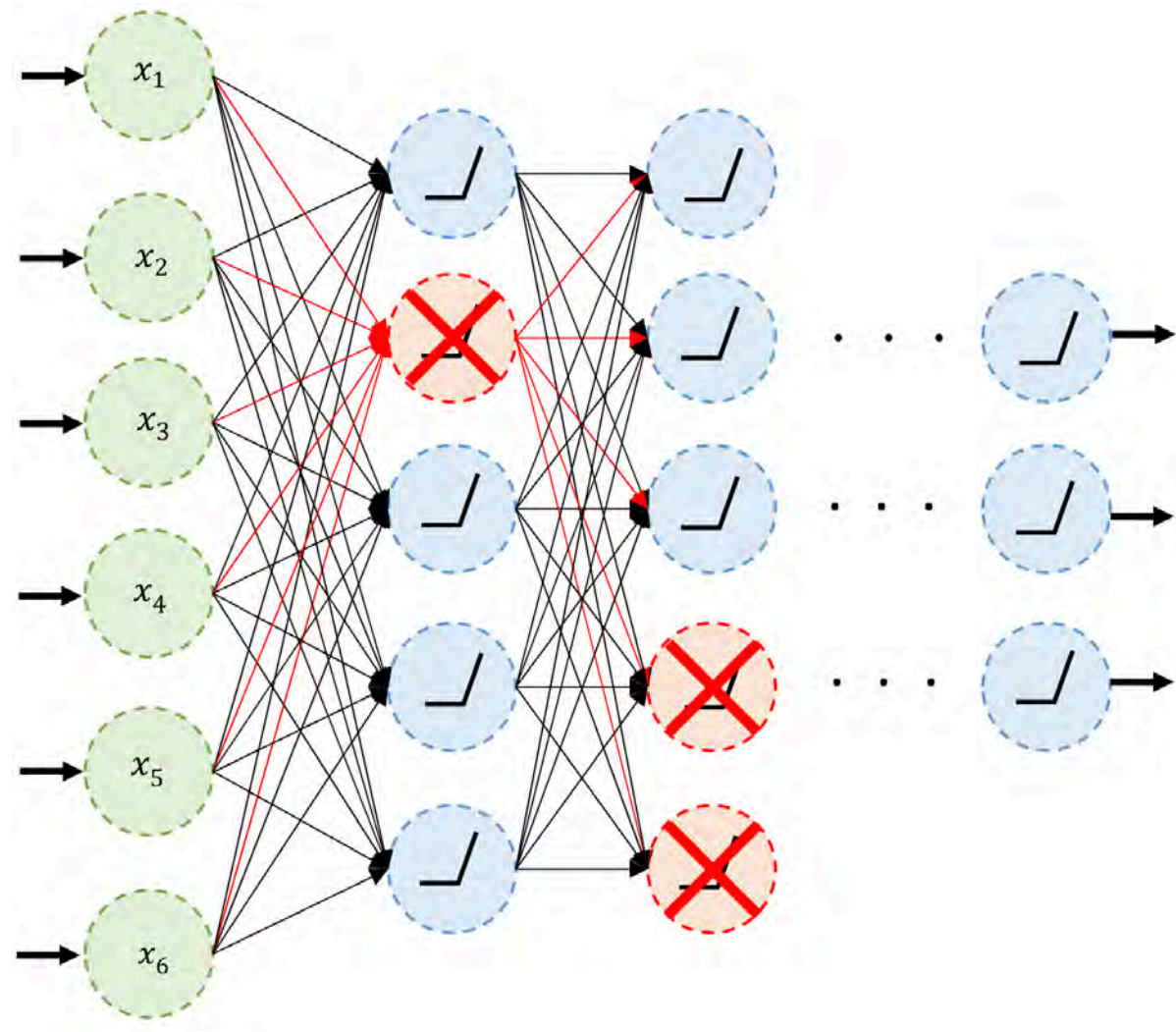
Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a scientist at Google, has dubbed this the “stretch pants” approach: instead 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

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- **Dropout**
- Ensembles



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.

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.

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"),
7     Dropout(0.2),
8     Dense(30, activation="leaky_relu"),
9     Dropout(0.2),
10    Dense(1, activation="exponential")
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([[1.45],
       [0.69],
       [1.57]], dtype=float32)
```

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

```
array([[1.45],
       [0.69],
       [1.57]], dtype=float32)
```

We can make the model think it is still training:

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

```
array([[1.64],
       [0.84],
       [1.84]], dtype=float32)
```

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

```
array([[1.35],
       [1.5 ],
       [1.8 ]], dtype=float32)
```

By setting the `training=True`, we can let dropout happen during prediction stage as well. This will change predictions for the same output different. This is known as the *Monte Carlo dropout* and can be used to generate a distribution of predictions.

Dropout Limitation

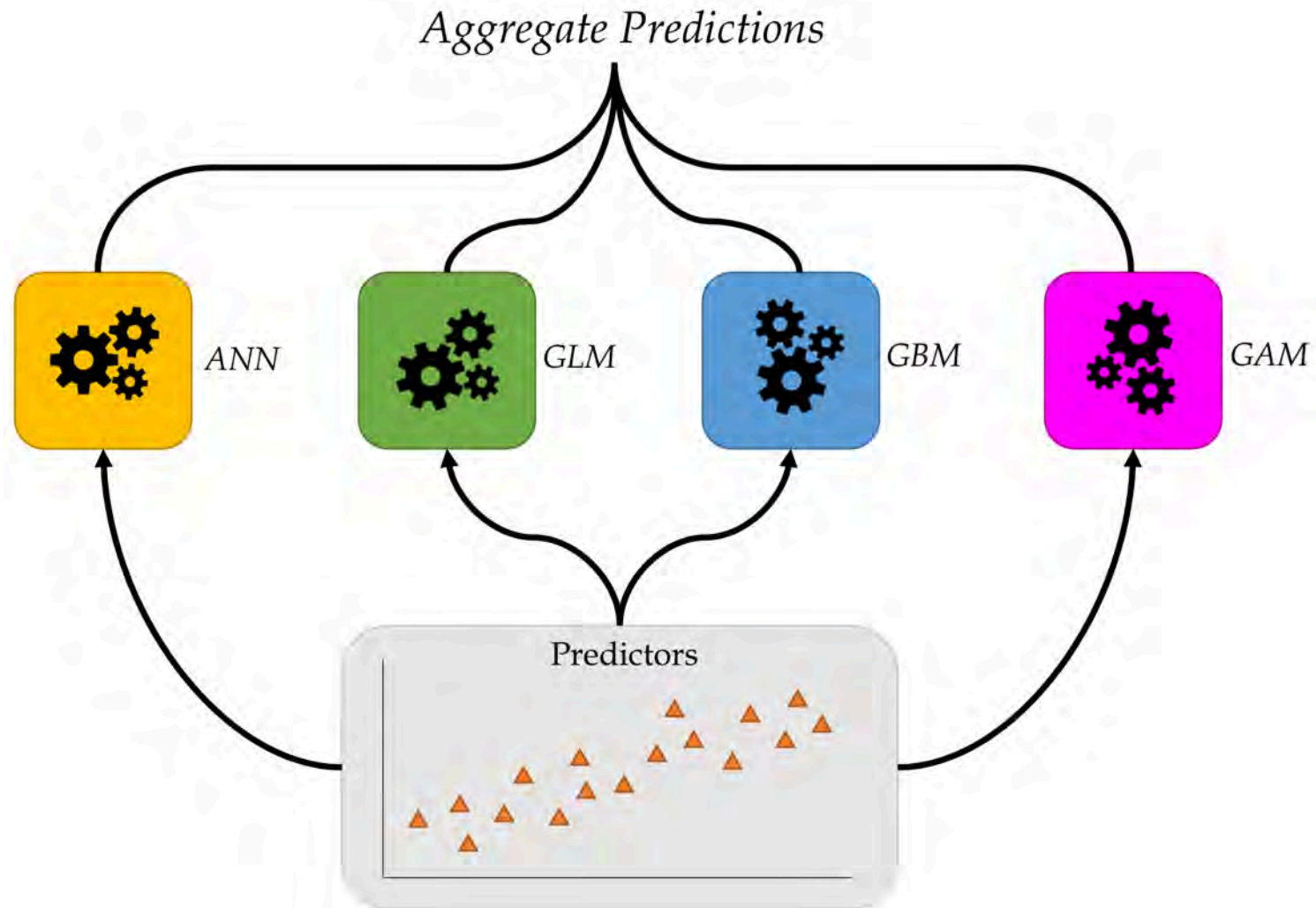
- **Increased Training Time:** Since dropout introduces noise into the training process, it can make the training process slower.
- **Sensitivity to Dropout Rates:** the performance of dropout is highly dependent on the chosen dropout rate.



Lecture Outline

- Introduction
- Traditional Regression
- Stochastic Forecasts
- GLMs and Neural Networks
- Combined Actuarial Neural Network
- Mixture Density Network
- Metrics for Distributional Regression
- Aleatoric and Epistemic Uncertainty
- Dropout
- **Ensembles**

Ensembles



Combine many models to get better predictions.

Source: Marcus Lautier (2022).

Deep Ensembles

Train M neural networks with different random initial weights independently (even in parallel).

```
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 M = 3
2 seeds = range(M)
3 models = []
4 for seed in seeds:
5     models.append(build_model(seed))
```

Deep Ensembles II

Say the trained weights for the NNs are $\mathbf{w}^{(1)}, \dots, \mathbf{w}^{(M)}$, then we get predictions

$$\{\hat{y}(\mathbf{x}; \mathbf{w}^{(m)})\}_{m=1}^M$$

```

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([[2.93],
       [0.9 ],
       [2.39],
       ...,
       [2.26],
       [2.54],
       [1.3 ]],

       [[1.9 ],
       [0.94],
       [0.8 ],
       ...,
       [1.76],
       [1.96],
       [2.61]])

```

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       : 3.14.3
IPython version     : 9.13.0
```

```
keras      : 3.14.1
matplotlib: 3.10.9
numpy      : 2.4.4
pandas     : 3.0.2
seaborn    : 0.13.2
scipy      : 1.17.1
torch      : 2.11.0
```

Glossary

- aleatoric and epistemic uncertainty
- combined actuarial neural network
- dead ReLU
- deep ensembles
- distributional forecasts
- dropout
- generalised linear model
- mixture density network
- mixture distribution
- Monte Carlo dropout
- proper scoring rule