

# Categorical Variables

ACTL3143 & ACTL5111 Deep Learning for Actuaries  
Patrick Laub



# Lecture Outline

- **Preprocessing**
- French Motor Claims & Poisson Regression
- Ordinal Variables
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure



# Keras model methods

- **compile**: specify the loss function and optimiser
- **fit**: learn the parameters of the model
- **predict**: apply the model
- **evaluate**: apply the model and calculate a metric

```
1 random.seed(12)
2 model = Sequential()
3 model.add(Dense(1, activation="relu"))
4 model.compile("adam", "poisson")
5 model.fit(X_train, y_train, verbose=0)
6 y_pred = model.predict(X_val, verbose=0)
7 print(model.evaluate(X_val, y_val, verbose=0))
```

4.944334506988525



# Scikit-learn model methods

- **fit**: learn the parameters of the model
- **predict**: apply the model
- **score**: apply the model and calculate a metric

```
1 model = LinearRegression()  
2 model.fit(X_train, y_train)  
3 y_pred = model.predict(X_val)  
4 print(model.score(X_val, y_val))
```

-0.666850597951445



# Scikit-learn preprocessing methods

- **fit**: learn the parameters of the transformation
- **transform**: apply the transformation
- **fit\_transform**: learn the parameters and apply the transformation

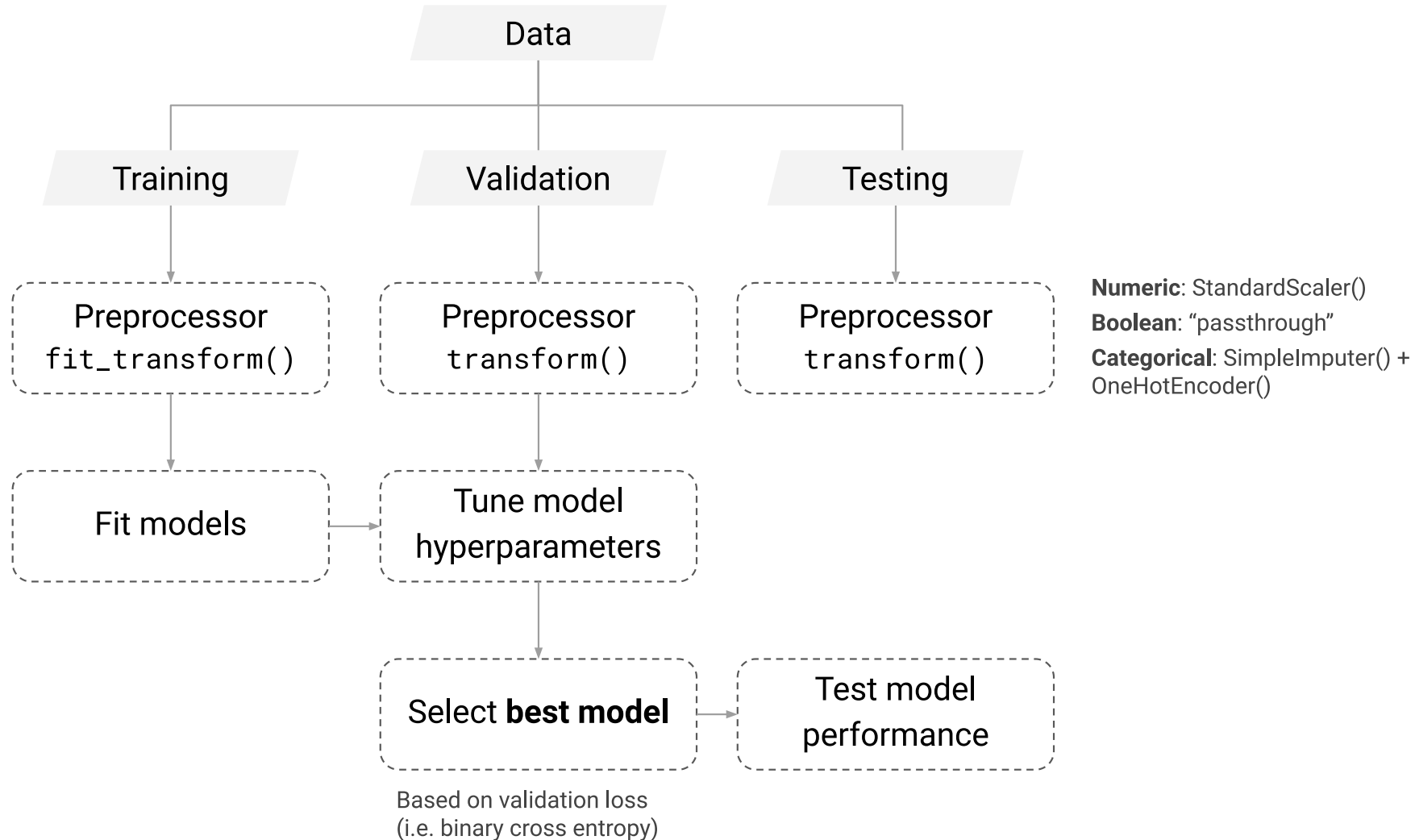
fit	fit_transform
-----	---------------

```
1 scaler = StandardScaler()
2 scaler.fit(X_train)
3 X_train_sc = scaler.transform(X_train)
4 X_val_sc = scaler.transform(X_val)
5 X_test_sc = scaler.transform(X_test)
6
7 print(X_train_sc.mean(axis=0))
8 print(X_train_sc.std(axis=0))
9 print(X_val_sc.mean(axis=0))
10 print(X_val_sc.std(axis=0))
```

```
[ 2.97e-17 -2.18e-17  1.98e-17 -5.65e-17]
[1.  1.  1.  1.]
[-0.34  0.07 -0.27 -0.82]
[1.01 0.66 1.26 0.89]
```



# Summary of the splitting



Source: Melantha Wang (2022), ACTL3143 Project.



# Dataframes & arrays

```
1 X_test.head(3)
```

	X1	X2	
83	0.075805	-0.677162	0.9751
53	0.954002	0.651391	-0.315
70	0.113517	0.662131	1.5860

```
1 X_test_sc
```

```
array([[ 0.13, -0.64,  0.89, -0.4 ],
       [ 1.15,  0.67, -0.44,  0.62],
       [ 0.18,  0.68,  1.52, -1.62],
       [ 0.77, -0.82, -1.22,  0.31],
       [ 0.06,  1.46, -0.39,  2.83],
       [ 2.21,  0.49, -1.34,  0.51],
       [-0.57,  0.53, -0.02,  0.86],
       [ 0.16,  0.61, -0.96,  2.12],
       [ 0.9 ,  0.2 , -0.23, -0.57],
       [ 0.62, -0.11,  0.55,  1.48],
       [ 0. ,  1.57, -2.81,  0.69],
       [ 0.96, -0.87,  1.33, -1.81],
       [-0.64,  0.87,  0.25, -1.01],
       [-1.19,  0.49, -1.06,  1.51],
       [ 0.65,  1.54, -0.23,  0.22],
       [-1.13,  0.34, -1.05, -1.82],
       [ 0.02,  0.14,  1.2 , -0.9 ],
       [ 0.68, -0.17, -0.34,  1. ],
       [ 0.44, -1.72,  0.22, -0.66],
```

## Note

By default, when you pass `sklearn` a DataFrame it returns a `numpy` array.



# Keep as a DataFrame

From **scikit-learn 1.2**:

```
1 from sklearn import set_config
2 set_config(transform_output="pandas")
3
4 imp = SimpleImputer()
5 imp.fit(X_train)
6 X_train_imp = imp.fit_transform(X_train)
7 X_val_imp = imp.transform(X_val)
8 X_test_imp = imp.transform(X_test)
```

```
1 X_test_imp
```

	X1	X2	
83	0.075805	-0.677162	0
53	0.954002	0.651391	-
...	...	...	..
42	-0.245388	-0.753736	-
69	0.199060	-0.600217	0

25 rows × 4 columns



# Lecture Outline

- Preprocessing
- **French Motor Claims & Poisson Regression**
- Ordinal Variables
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure



# French motor dataset

Download the dataset if we don't have it already.

```
1 from pathlib import Path
2 from sklearn.datasets import fetch_openml
3
4 if not Path("french-motor.csv").exists():
5     freq = fetch_openml(data_id=41214, as_frame=True).frame
6     freq.to_csv("french-motor.csv", index=False)
7 else:
8     freq = pd.read_csv("french-motor.csv")
9
10 freq
```



# French motor dataset

	IDpol	ClaimNb	Exposure	Area	VehPower	VehAge
0	1.0	1.0	0.10000	D	5.0	0.0
1	3.0	1.0	0.77000	D	5.0	0.0
2	5.0	1.0	0.75000	B	6.0	2.0
...	...	...	...	...	...	...
678010	6114328.0	0.0	0.00274	D	6.0	2.0
678011	6114329.0	0.0	0.00274	B	4.0	0.0
678012	6114330.0	0.0	0.00274	B	7.0	6.0

678013 rows × 12 columns

# Data dictionary

- **IDpol**: policy number (unique identifier)
- **ClaimNb**: number of claims on the given policy
- **Exposure**: total exposure in yearly units
- **Area**: area code (categorical, ordinal)
- **VehPower**: power of the car (categorical, ordinal)
- **VehAge**: age of the car in years
- **DrivAge**: age of the (most common) driver in years
- **BonusMalus**: bonus-malus level between 50 and 230 (with reference level 100)
- **VehBrand**: car brand (categorical, nominal)
- **VehGas**: diesel or regular fuel car (binary)
- **Density**: density of inhabitants per km<sup>2</sup> in the city of the living place of the driver
- **Region**: regions in France (prior to 2016)



# The model

Have  $\{(\mathbf{x}_i, y_i)\}_{i=1, \dots, n}$  for  $\mathbf{x}_i \in \mathbb{R}^{47}$  and  $y_i \in \mathbb{N}_0$ .

Assume the distribution

$$Y_i \sim \text{Poisson}(\lambda(\mathbf{x}_i))$$

We have  $\mathbb{E}Y_i = \lambda(\mathbf{x}_i)$ . The NN takes  $\mathbf{x}_i$  & predicts  $\mathbb{E}Y_i$ .

## Note

For insurance, *this is a bit weird*. The exposures are different for each policy.

$\lambda(\mathbf{x}_i)$  is the expected number of claims for the duration of policy  $i$ 's contract.

Normally,  $\text{Exposure}_i \notin \mathbf{x}_i$ , and  $\lambda(\mathbf{x}_i)$  is the expected rate *per year*, then

$$Y_i \sim \text{Poisson}(\text{Exposure}_i \times \lambda(\mathbf{x}_i)).$$

# Where are things defined?

In Keras, string options are used for convenience to reference specific functions or settings.

```
1 model = Sequential([
2     Dense(30, activation="relu"),
3     Dense(1, activation="exponential")
4 ])
```

is the same as

```
1 from keras.activations import relu, exponential
2
3 model = Sequential([
4     Dense(30, activation=relu),
5     Dense(1, activation=exponential)
6 ])
```

```
1 x = [-1.0, 0.0, 1.0]
2 print(relu(x))
3 print(exponential(x))
```

```
tf.Tensor([0. 0. 1.], shape=(3,), dtype=float32)
tf.Tensor([0.37 1.    2.72], shape=(3,), dtype=float32)
```



# String arguments to `.compile`

When we run

```
1 model.compile(optimizer="adam", loss="poisson")
```

it is equivalent to

```
1 from keras.losses import poisson
2 from keras.optimizers import Adam
3
4 model.compile(optimizer=Adam(), loss=poisson)
```

Why do this manually? To adjust the object:

```
1 optimizer = Adam(learning_rate=0.01)
2 model.compile(optimizer=optimizer, loss="poisson")
```

or to get help.



# Keras' “poisson” loss

```
1 help(keras.losses.poisson)
```

Help on function poisson in module keras.src.losses.losses:

```
poisson(y_true, y_pred)
```

Computes the Poisson loss between y\_true and y\_pred.

Formula:

```
```python
loss = y_pred - y_true * log(y_pred)
```
```

Args:

y\_true: Ground truth values. shape = `[batch\_size, d0, .. dN]`.

y\_pred: The predicted values. shape = `[batch\_size, d0, .. dN]`.

Returns:

Poisson loss values with shape = `[batch\_size, d0, .. dN-1]`.

Example:





# Lecture Outline

- Preprocessing
- French Motor Claims & Poisson Regression
- **Ordinal Variables**
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure



# Subsample and split

```
1 freq = freq.drop("IDpol", axis=1).head(25_000)
2
3 X_train, X_test, y_train, y_test = train_test_split(
4     freq.drop("ClaimNb", axis=1), freq["ClaimNb"], 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)
```



# What values do we see in the data?

```
1 X_train["Area"].value_counts()
2 X_train["VehBrand"].value_counts()
3 X_train["VehGas"].value_counts()
4 X_train["Region"].value_counts()
```

Area

```
C    5507
D    4113
A    3527
E    2769
B    2359
F     475
```

Name: count, dtype: int64

VehGas

```
Regular    10773
Diesel      7977
```

Name: count, dtype: int64

VehBrand

```
B1    5069
B2    4838
B12   3708
...
B13    336
B11    284
B14    136
```

Name: count, Length: 11, dtype: int64

Region

```
R24    6498
R82    2119
R11    1909
```

```
...
R21     90
R42     55
R43     26
```

Name: count, Length: 22, dtype: int64



# Ordinal & binary categories are easy

```
1 from sklearn.preprocessing import OrdinalEncoder
2 oe = OrdinalEncoder()
3 oe.fit(X_train[["Area", "VehGas"]])
4 oe.categories_
```

```
[array(['A', 'B', 'C', 'D', 'E', 'F'], dtype=object),
 array(['Diesel', 'Regular'], dtype=object)]
```

```
1 for i, area in enumerate(oe.categories_[0]):
2     print(f"The Area value {area} gets turned into {i}.")
```

The Area value A gets turned into 0.  
 The Area value B gets turned into 1.  
 The Area value C gets turned into 2.  
 The Area value D gets turned into 3.  
 The Area value E gets turned into 4.  
 The Area value F gets turned into 5.

```
1 for i, gas in enumerate(oe.categories_[1]):
2     print(f"The VehGas value {gas} gets turned into {i}.")
```

The VehGas value Diesel gets turned into 0.  
 The VehGas value Regular gets turned into 1.



# Ordinal encoded values

```
1 X_train_ord = oe.transform(X_train[["Area", "VehGas"]])
2 X_test_ord = oe.transform(X_test[["Area", "VehGas"]])
```

```
1 X_train[["Area", "VehGas"]].head()
```

|   | Area | VehGas  |
|---|------|---------|
| 0 | C    | Diesel  |
| 1 | C    | Regular |
| 2 | E    | Regular |
| 3 | D    | Diesel  |
| 4 | A    | Regular |

```
1 X_train_ord.head()
```

|   | Area | VehGas |
|---|------|--------|
| 0 | 2.0  | 0.0    |
| 1 | 2.0  | 1.0    |
| 2 | 4.0  | 1.0    |
| 3 | 3.0  | 0.0    |
| 4 | 0.0  | 1.0    |



# Train on ordinal encoded values

```
1 random.seed(12)
2 model = Sequential([
3     Dense(1, activation="exponential")
4 ])
5
6 model.compile(optimizer="adam", loss="poisson")
7
8 es = EarlyStopping(verbose=True)
9 hist = model.fit(X_train_ord, y_train, epochs=100, verbose=0,
10                 validation_split=0.2, callbacks=[es])
11 hist.history["val_loss"][-1]
```

Epoch 22: early stopping

0.7821308970451355

What about adding the continuous variables back in? Use a sklearn *column transformer* for that.



# Preprocess ordinal & continuous

```

1 from sklearn.compose import make_column_transformer
2
3 ct = make_column_transformer(
4     (OrdinalEncoder(), ["Area", "VehGas"]),
5     ("drop", ["VehBrand", "Region"]),
6     remainder=StandardScaler()
7 )
8
9 X_train_ct = ct.fit_transform(X_train)

```

```
1 X_train.head(3)
```

|   | Exposure | Area | VehPower |
|---|----------|------|----------|
| 0 | 1.00     | C    | 6.0      |
| 1 | 0.36     | C    | 4.0      |
| 2 | 0.02     | E    | 12.0     |

```
1 X_train_ct.head(3)
```

|   | ordinalencoder__Area | ord |
|---|----------------------|-----|
| 0 | 2.0                  | 0.0 |
| 1 | 2.0                  | 1.0 |
| 2 | 4.0                  | 1.0 |

# Preprocess ordinal & continuous II

```

1 from sklearn.compose import make_column_transformer
2
3 ct = make_column_transformer(
4     (OrdinalEncoder(), ["Area", "VehGas"]),
5     ("drop", ["VehBrand", "Region"]),
6     remainder=StandardScaler(),
7     verbose_feature_names_out=False
8 )
9 X_train_ct = ct.fit_transform(X_train)

```

```
1 X_train.head(3)
```

|   | Exposure | Area | VehPower |
|---|----------|------|----------|
| 0 | 1.00     | C    | 6.0      |
| 1 | 0.36     | C    | 4.0      |
| 2 | 0.02     | E    | 12.0     |

```
1 X_train_ct.head(3)
```

|   | Area | VehGas | Exposure  |
|---|------|--------|-----------|
| 0 | 2.0  | 0.0    | 1.126979  |
| 1 | 2.0  | 1.0    | -0.590896 |
| 2 | 4.0  | 1.0    | -1.503517 |



# Lecture Outline

- Preprocessing
- French Motor Claims & Poisson Regression
- Ordinal Variables
- **Categorical Variables & Entity Embeddings**
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure



# Region column



## French Administrative Regions

Source: Nell et al. (2020), [Case Study: French Motor Third-Party Liability Claims](#), SSRN.



# One-hot encoding

```
1 oe = OneHotEncoder(sparse_output=False)
2 X_train_oh = oe.fit_transform(X_train[["Region"]])
3 X_test_oh = oe.transform(X_test[["Region"]])
4 print(list(X_train["Region"][:5]))
5 X_train_oh.head()
```

['R24', 'R93', 'R11', 'R42', 'R24']

|   | Region_R11 | Region_R21 | Region_R22 | Region_R23 | Region |
|---|------------|------------|------------|------------|--------|
| 0 | 0.0        | 0.0        | 0.0        | 0.0        | 1.0    |
| 1 | 0.0        | 0.0        | 0.0        | 0.0        | 0.0    |
| 2 | 1.0        | 0.0        | 0.0        | 0.0        | 0.0    |
| 3 | 0.0        | 0.0        | 0.0        | 0.0        | 0.0    |
| 4 | 0.0        | 0.0        | 0.0        | 0.0        | 1.0    |

5 rows × 22 columns



# Train on one-hot inputs

```
1 num_regions = len(oe.categories_[0])
2
3 random.seed(12)
4 model = Sequential([
5     Dense(2, input_dim=num_regions),
6     Dense(1, activation="exponential")
7 ])
8
9 model.compile(optimizer="adam", loss="poisson")
10
11 es = EarlyStopping(verbose=True)
12 hist = model.fit(X_train_oh, y_train, epochs=100, verbose=0,
13                 validation_split=0.2, callbacks=[es])
14 hist.history["val_loss"][-1]
```

Epoch 12: early stopping

0.7526934146881104



# Consider the first layer

```
1 every_category = pd.DataFrame(np.eye(num_regions), columns=oe.categories_[0])
2 every_category.head(3)
```

|   | R11 | R21 | R22 | R23 | R24 | R25 | R26 | R31 | R41 | R42 | ... | R53 | R54 | R72 | R73 | R74 | R8  |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

3 rows  $\times$  22 columns

```
1 # Put this through the first layer of the model
2 X = every_category.to_numpy()
3 model.layers[0](X)
```

```
<tf.Tensor: shape=(22, 2), dtype=float32, numpy=
array([[ -0.21, -0.14],
       [ 0.21, -0.17],
       [-0.22,  0.1 ],
       [-0.83,  0.1 ],
       [-0.01, -0.66],
       [-0.65, -0.13],
       [-0.36, -0.41],
       [ 0.21, -0.03],
       [-0.93, -0.57],
       [ 0.2 , -0.41],
       [-0.43, -0.21],
       [-1.13, -0.33],
       [ 0.17, -0.68],
       [-0.88, -0.55],
       [-0.13,  0.05],
       [ 0.11,  0.  ],
       [-0.46, -0.38],
       [-0.62, -0.37],
       [-0.19, -0.28],
       [-0.22,  0.15],
       [ 0.2 ,  0.16],
       [ 0.2 ,  0.16]])
```





# Just a look-up operation

```
1 display(list(oe.categories_[0]))
```

```
['R11',
 'R21',
 'R22',
 'R23',
 'R24',
 'R25',
 'R26',
 'R31',
 'R41',
 'R42',
 'R43',
 'R52',
 'R53',
 'R54',
 'R72',
 'R73',
 'R74',
 'R82',
 'R83',
 ...]
```

```
1 W + b
```

```
array([[ -0.21, -0.14],
       [ 0.21, -0.17],
       [-0.22,  0.1 ],
       [-0.83,  0.1 ],
       [-0.01, -0.66],
       [-0.65, -0.13],
       [-0.36, -0.41],
       [ 0.21, -0.03],
       [-0.93, -0.57],
       [ 0.2 , -0.41],
       [-0.43, -0.21],
       [-1.13, -0.33],
       [ 0.17, -0.68],
       [-0.88, -0.55],
       [-0.13,  0.05],
       [ 0.11,  0.   ],
       [-0.46, -0.38],
       [-0.62, -0.37],
       [-0.19, -0.28],
       ...])
```



# Turn the region into an index

```
1 oe = OrdinalEncoder()  
2 X_train_reg = oe.fit_transform(X_train[["Region"]])  
3 X_test_reg = oe.transform(X_test[["Region"]])  
4  
5 for i, reg in enumerate(oe.categories_[0][:3]):  
6     print(f"The Region value {reg} gets turned into {i}.")
```

The Region value R11 gets turned into 0.

The Region value R21 gets turned into 1.

The Region value R22 gets turned into 2.





# Embedding

```
1 from keras.layers import Embedding
2 num_regions = len(np.unique(X_train[["Region"]]))
3
4 random.seed(12)
5 model = Sequential([
6     Embedding(input_dim=num_regions, output_dim=2),
7     Dense(1, activation="exponential")
8 ])
9
10 model.compile(optimizer="adam", loss="poisson")
```



# Fitting that model

```
1 es = EarlyStopping(verbose=True)
2 hist = model.fit(X_train_reg, y_train, epochs=100, verbose=0,
3                 validation_split=0.2, callbacks=[es])
4 hist.history["val_loss"][-1]
```

Epoch 5: early stopping

0.7526668906211853

```
1 model.layers
```

```
[<Embedding name=embedding, built=True>, <Dense name=dense_8, built=True>]
```



# Keras' Embedding Layer

```
1 model.layers[0].get_weights()[0]
```

```
array([[ -0.12, -0.11],
       [ 0.03, -0.   ],
       [-0.02,  0.01],
       [-0.25, -0.14],
       [-0.28, -0.32],
       [-0.3 , -0.22],
       [-0.31, -0.28],
       [ 0.1 ,  0.07],
       [-0.61, -0.51],
       [-0.06, -0.12],
       [-0.17, -0.14],
       [-0.6 , -0.46],
       [-0.22, -0.27],
       [-0.59, -0.5 ],
       [-0.   ,  0.02],
       [ 0.07,  0.06],
       [-0.31, -0.28],
       [-0.4 , -0.34],
       [-0.16, -0.15],
       ...])
```

```
1 X_train["Region"].head(4)
```

```
0    R24
1    R93
2    R11
3    R42
Name: Region, dtype: object
```

```
1 X_sample = X_train_reg[:4].to_numpy()
2 X_sample
```

```
array([[ 4.],
       [20.],
       [ 0.],
       [ 9.]])
```

```
1 enc_tensor = model.layers[0](X_sample)
2 keras.ops.convert_to_numpy(enc_tensor).
```

```
array([[ -0.28, -0.32],
       [ 0.08,  0.03],
       [-0.12, -0.11],
       [-0.06, -0.12]], dtype=float32)
```

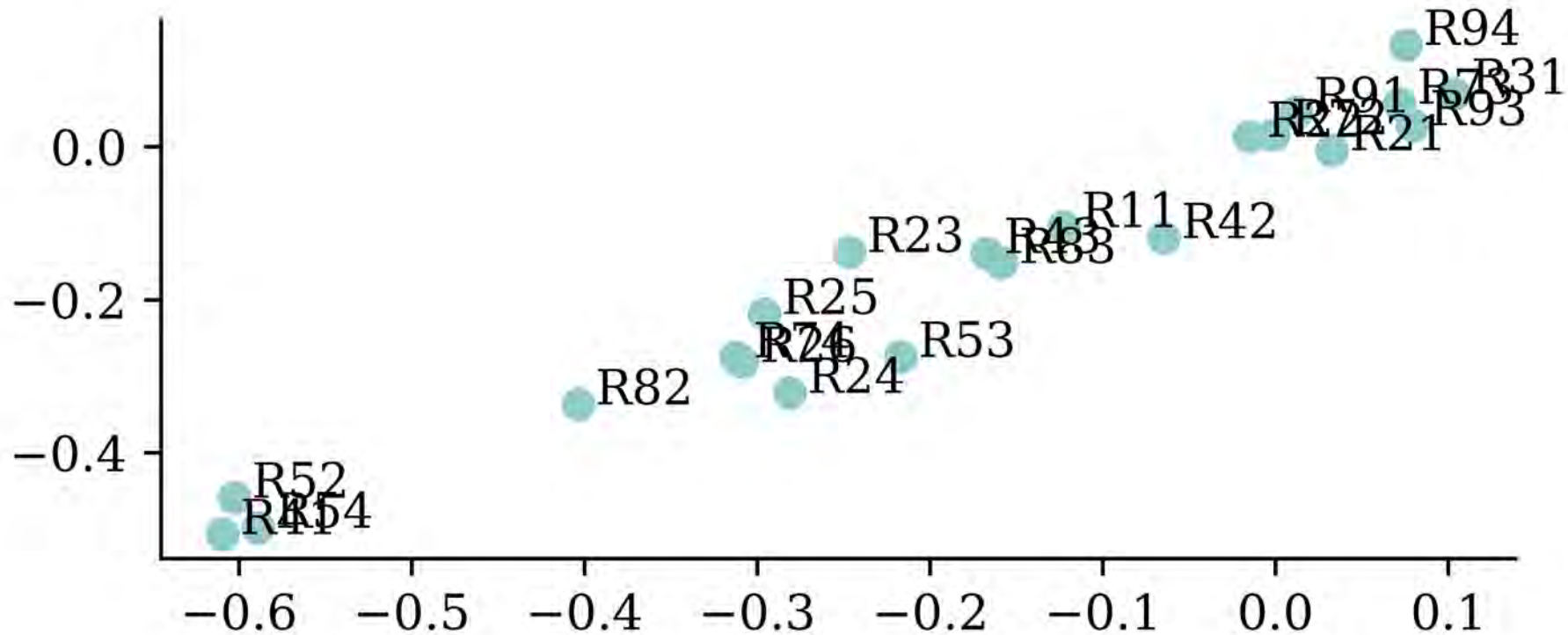


# The learned embeddings

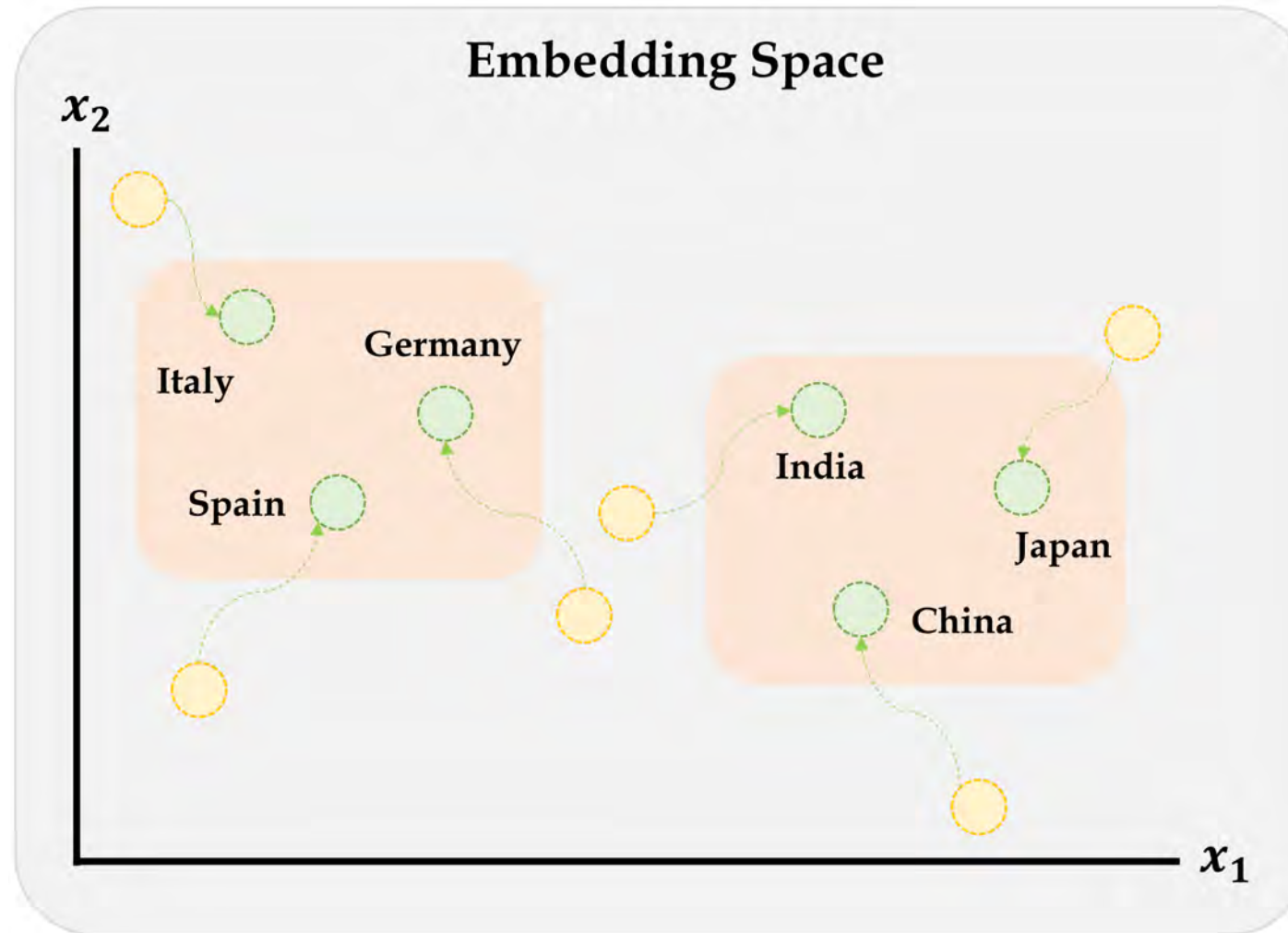
```

1 points = model.layers[0].get_weights()[0]
2 plt.scatter(points[:,0], points[:,1])
3 for i in range(num_regions):
4     plt.text(points[i,0]+0.01, points[i,1] , s=oe.categories_[0][i])

```



# Entity embeddings



Embeddings will gradually improve during training.

Source: Marcus Lautier (2022).



# Embeddings & other inputs

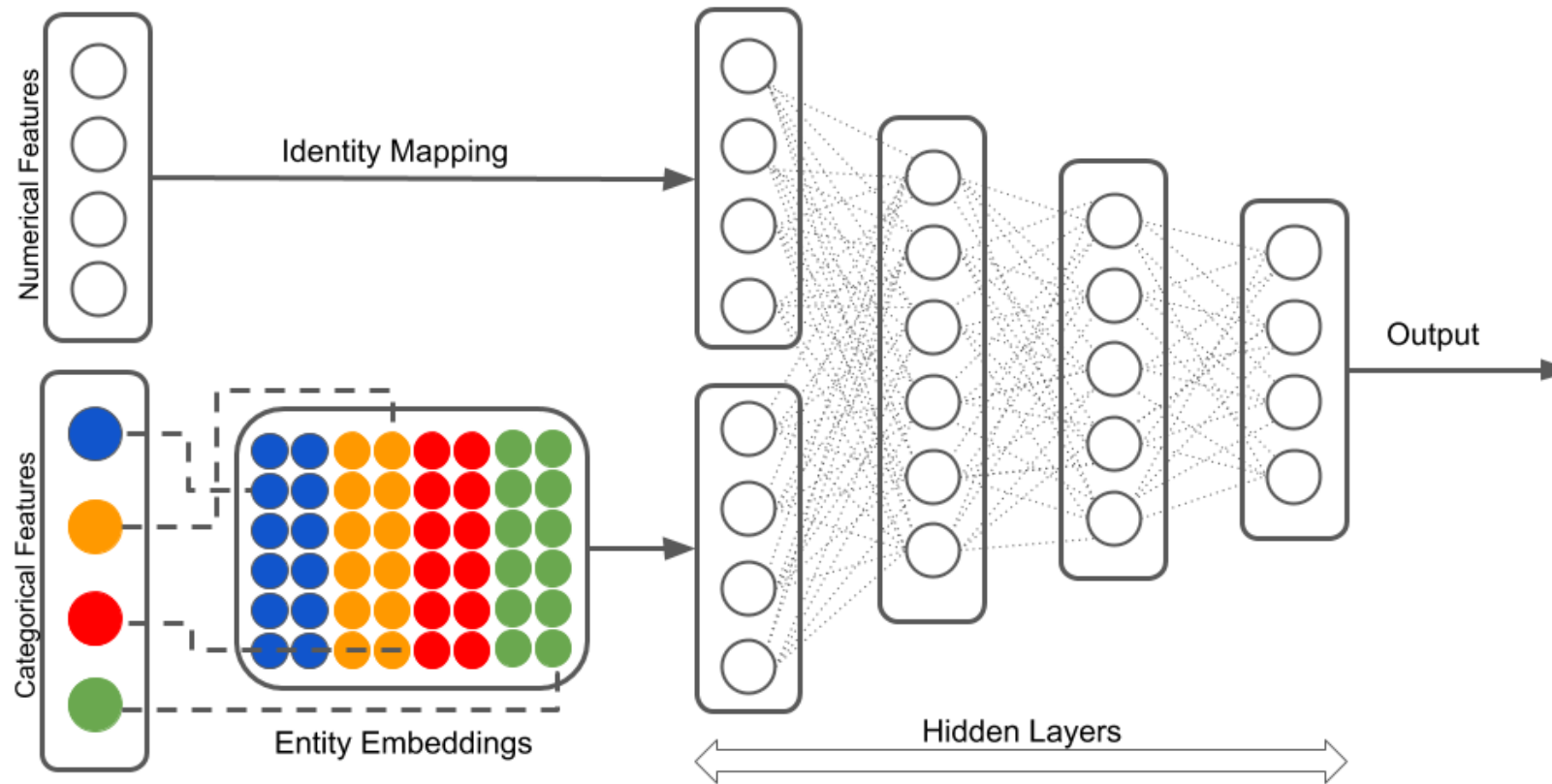


Illustration of a neural network with both continuous and categorical inputs.

We can't do this with Sequential models...

Source: LotusLabs Blog, [Accurate insurance claims prediction with Deep Learning](#).



# Lecture Outline

- Preprocessing
- French Motor Claims & Poisson Regression
- Ordinal Variables
- Categorical Variables & Entity Embeddings
- **Keras' Functional API**
- French Motor Dataset with Embeddings
- Scale By Exposure



# Converting Sequential models

```
1 from keras.models import Model
2 from keras.layers import Input
```

```
1 random.seed(12)
2
3 model = Sequential([
4     Dense(30, "relu"),
5     Dense(1, "exponential")
6 ])
7
8 model.compile(
9     optimizer="adam",
10    loss="poisson")
11
12 hist = model.fit(
13     X_train_ord, y_train,
14     epochs=1, verbose=0,
15     validation_split=0.2)
16 hist.history["val_loss"][-1]
```

0.7844388484954834

```
1 random.seed(12)
2
3 inputs = Input(shape=(2,))
4 x = Dense(30, "relu")(inputs)
5 out = Dense(1, "exponential")(x)
6 model = Model(inputs, out)
7
8 model.compile(
9     optimizer="adam",
10    loss="poisson")
11
12 hist = model.fit(
13     X_train_ord, y_train,
14     epochs=1, verbose=0,
15     validation_split=0.2)
16 hist.history["val_loss"][-1]
```

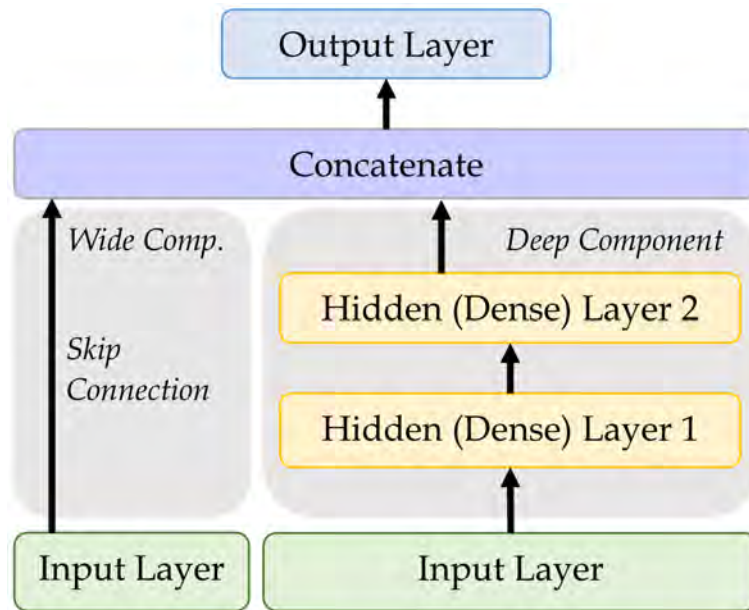
0.7844388484954834

Cf. **one-length tuples.**





# Wide & Deep network



An illustration of the wide & deep network architecture.

Add a *skip connection* from input to output layers.

```

1 from keras.layers \
2     import Concatenate
3
4 inp = Input(shape=X_train.shape[1:])
5 hidden1 = Dense(30, "relu")(inp)
6 hidden2 = Dense(30, "relu")(hidden1)
7 concat = Concatenate()(
8     [inp, hidden2])
9 output = Dense(1)(concat)
10 model = Model(
11     inputs=[inp],
12     outputs=[output])
  
```

# Naming the layers

For complex networks, it is often useful to give meaningful names to the layers.

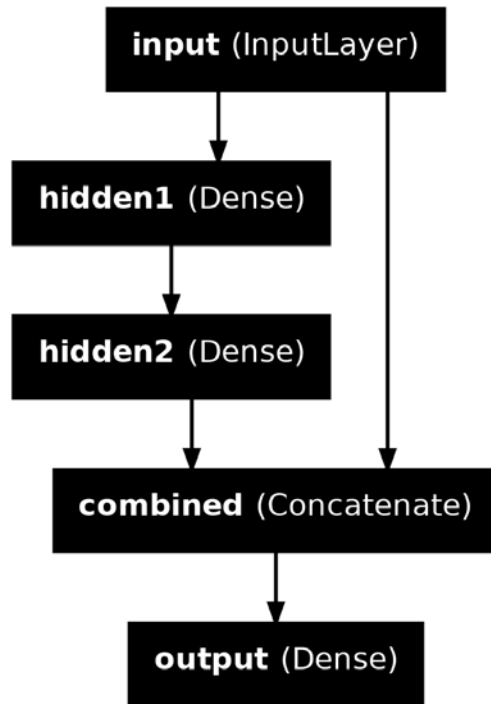
```
1 input_ = Input(shape=X_train.shape[1:], name="input")
2 hidden1 = Dense(30, activation="relu", name="hidden1")(input_)
3 hidden2 = Dense(30, activation="relu", name="hidden2")(hidden1)
4 concat = Concatenate(name="combined")([input_, hidden2])
5 output = Dense(1, name="output")(concat)
6 model = Model(inputs=[input_], outputs=[output])
```



# Inspecting a complex model

```
1 from keras.utils import plot_model
```

```
1 plot_model(model, show_shapes=True)
```



```
1 model.summary(line_length=75)
```

Model: "functional\_10"

| Layer (type)           | Output Shape | Param # | Connected to           |
|------------------------|--------------|---------|------------------------|
| input (InputLayer)     | (None, 10)   | 0       | -                      |
| hidden1 (Dense)        | (None, 30)   | 330     | input[0]               |
| hidden2 (Dense)        | (None, 30)   | 930     | hidden1[0]             |
| combined (Concatenate) | (None, 40)   | 0       | input[0]<br>hidden2[0] |
| output (Dense)         | (None, 1)    | 41      | combined[0]            |

Total params: 1,301 (5.08 KB)  
 Trainable params: 1,301 (5.08 KB)  
 Non-trainable params: 0 (0.00 B)

# Lecture Outline

- Preprocessing
- French Motor Claims & Poisson Regression
- Ordinal Variables
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- **French Motor Dataset with Embeddings**
- Scale By Exposure



# The desired architecture

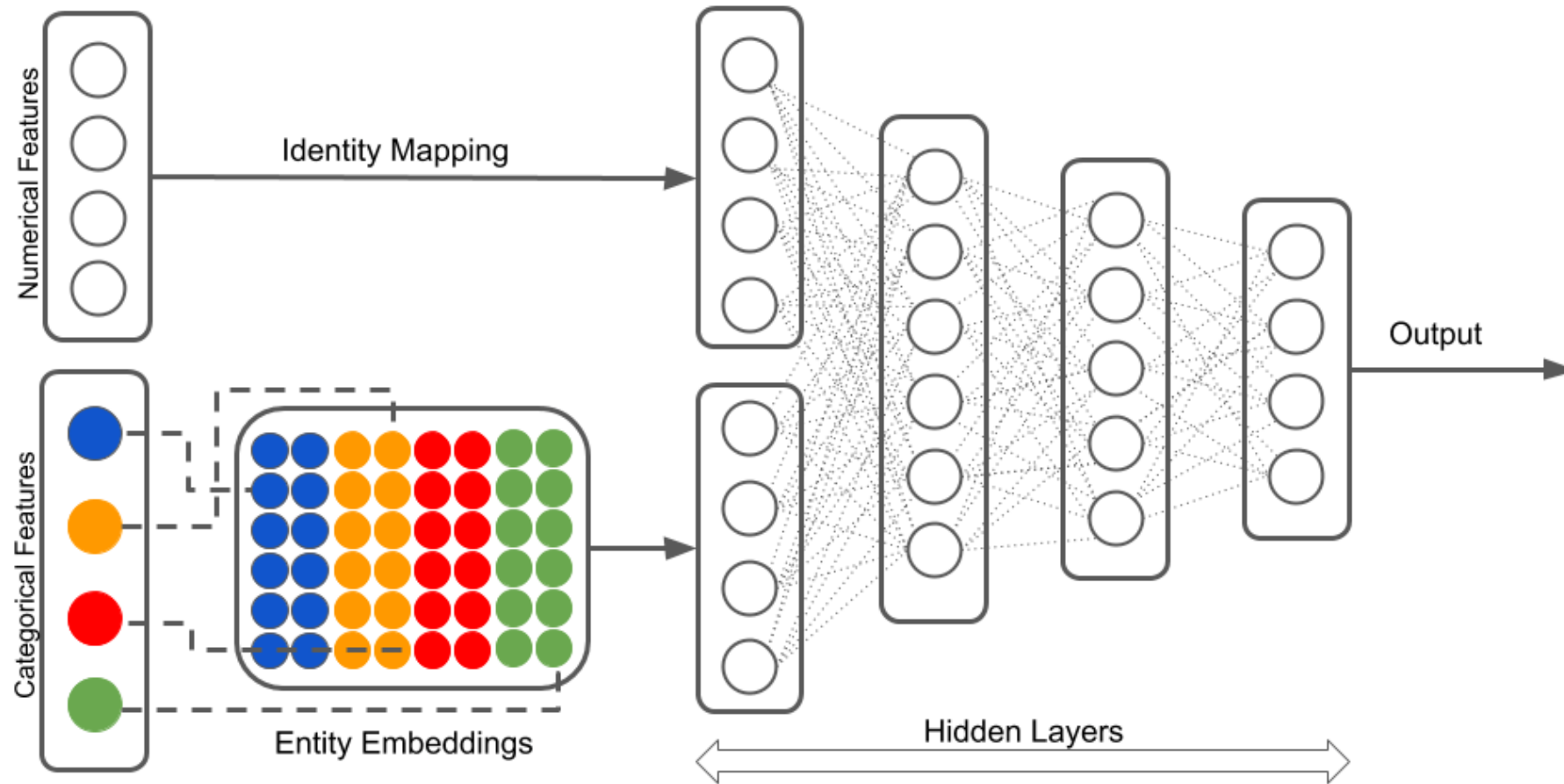


Illustration of a neural network with both continuous and categorical inputs.

Source: LotusLabs Blog, [Accurate insurance claims prediction with Deep Learning](#).



# Preprocess all French motor inputs

Transform the categorical variables to integers:

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

Split the brand and region data apart from the rest:

```
1 X_train_brand = X_train_ct["VehBrand"]; X_test_brand = X_test_ct["VehBrand"]
2 X_train_region = X_train_ct["Region"]; X_test_region = X_test_ct["Region"]
3 X_train_rest = X_train_ct.drop(["VehBrand", "Region"], axis=1)
4 X_test_rest = X_test_ct.drop(["VehBrand", "Region"], axis=1)
```



# Organise the inputs

Make a Keras **Input** for: vehicle brand, region, & others.

```
1 veh_brand = Input(shape=(1,), name="vehBrand")
2 region = Input(shape=(1,), name="region")
3 other_inputs = Input(shape=X_train_rest.shape[1:], name="otherInputs")
```

Create embeddings and join them with the other inputs.

```
1 from keras.layers import Reshape
2
3 random.seed(1337)
4 veh_brand_ee = Embedding(input_dim=num_brands, output_dim=2,
5     name="vehBrandEE")(veh_brand)
6 veh_brand_ee = Reshape(target_shape=(2,))(veh_brand_ee)
7
8 region_ee = Embedding(input_dim=num_regions, output_dim=2,
9     name="regionEE")(region)
10 region_ee = Reshape(target_shape=(2,))(region_ee)
11
12 x = Concatenate(name="combined")([veh_brand_ee, region_ee, other_inputs])
```



# Complete the model and fit it

Feed the combined embeddings & continuous inputs to some normal dense layers.

```
1 x = Dense(30, "relu", name="hidden")(x)
2 out = Dense(1, "exponential", name="out")(x)
3
4 model = Model([veh_brand, region, other_inputs], out)
5 model.compile(optimizer="adam", loss="poisson")
6
7 hist = model.fit((X_train_brand, X_train_region, X_train_rest),
8                 y_train, epochs=100, verbose=0,
9                 callbacks=[EarlyStopping(patience=5)], validation_split=0.2)
10 np.min(hist.history["val_loss"])
```

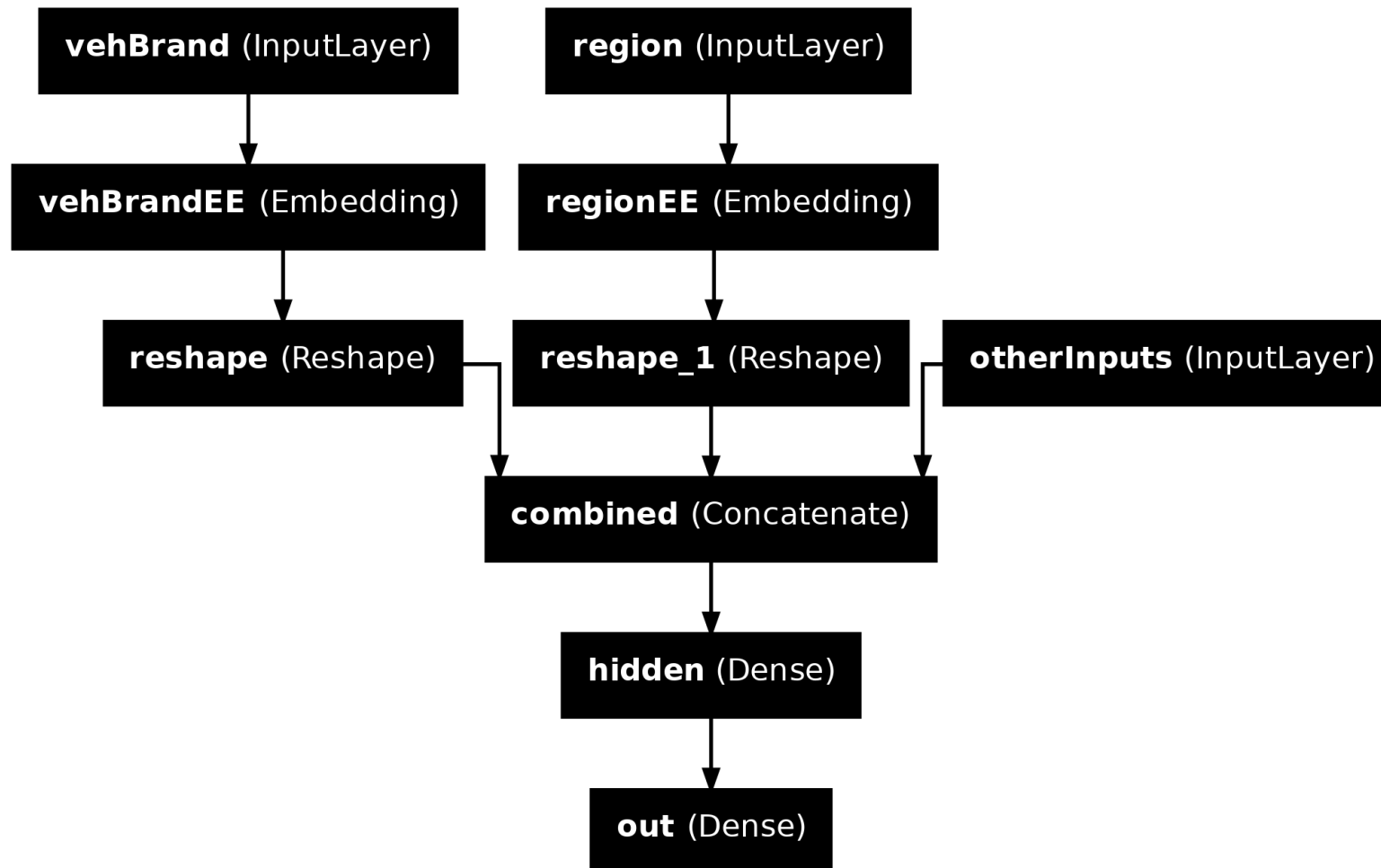
0.6692155599594116





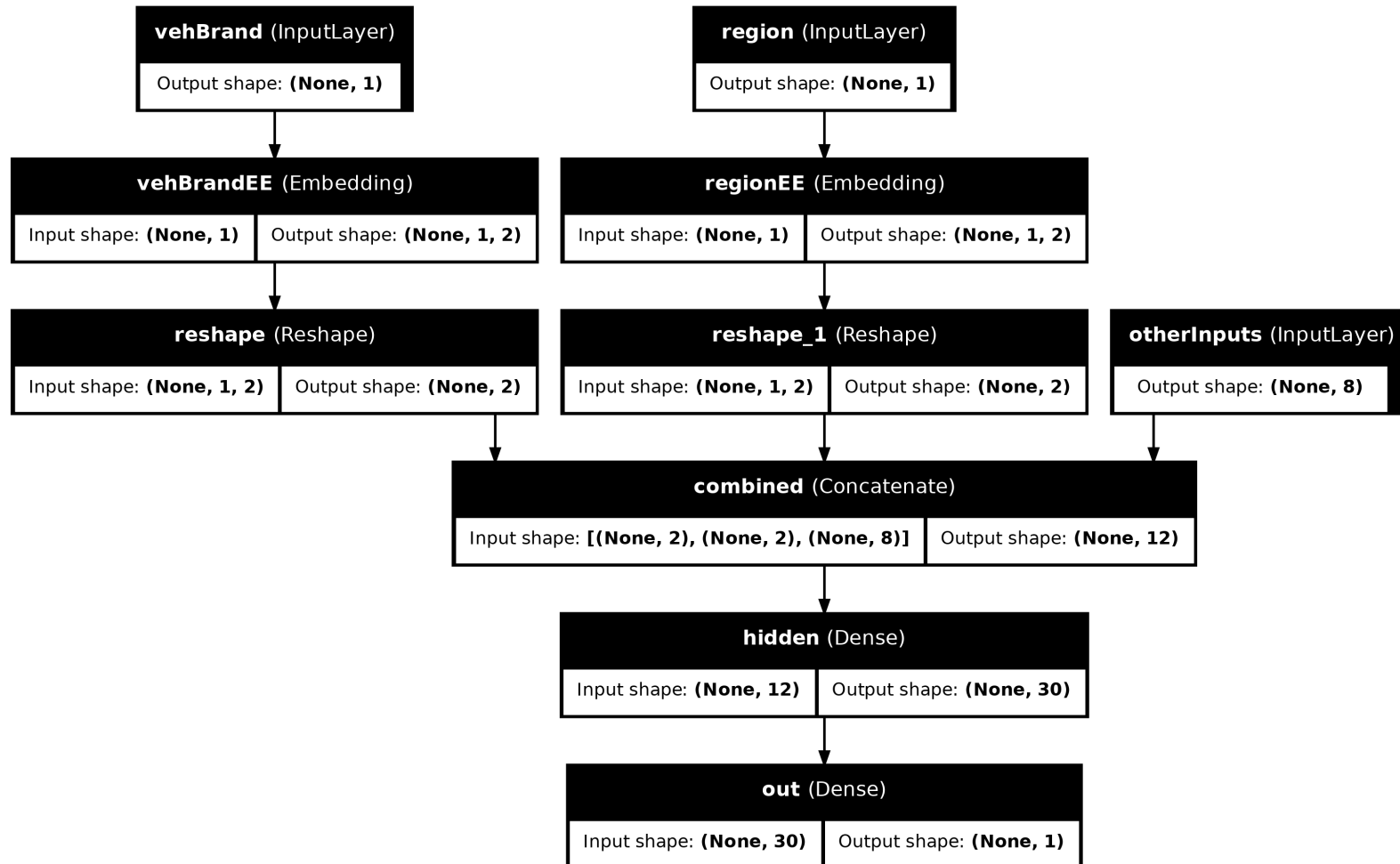
# Plotting this model

```
1 plot_model(model, show_layer_names=True)
```



# Why we need to reshape

```
1 plot_model(model, show_layer_names=True, show_shapes=True)
```



# Lecture Outline

- Preprocessing
- French Motor Claims & Poisson Regression
- Ordinal Variables
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- **Scale By Exposure**



# Two different models

Have  $\{(\mathbf{x}_i, y_i)\}_{i=1, \dots, n}$  for  $\mathbf{x}_i \in \mathbb{R}^{47}$  and  $y_i \in \mathbb{N}_0$ .

**Model 1:** Say  $Y_i \sim \text{Poisson}(\lambda(\mathbf{x}_i))$ .

But, the exposures are different for each policy.  $\lambda(\mathbf{x}_i)$  is the expected number of claims for the duration of policy  $i$ 's contract.

**Model 2:** Say  $Y_i \sim \text{Poisson}(\text{Exposure}_i \times \lambda(\mathbf{x}_i))$ .

Now,  $\text{Exposure}_i \notin \mathbf{x}_i$ , and  $\lambda(\mathbf{x}_i)$  is the rate *per year*.



# Just take continuous variables

```

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

```

Split exposure apart from the rest:

```

1 X_train_exp = X_train_ct["Exposure"]; X_test_exp = X_test_ct["Exposure"]
2 X_train_rest = X_train_ct.drop("Exposure", axis=1)
3 X_test_rest = X_test_ct.drop("Exposure", axis=1)

```

Organise the inputs:

```

1 exposure = Input(shape=(1,), name="exposure")
2 other_inputs = Input(shape=X_train_rest.shape[1:], name="otherInputs")

```



# Make & fit the model

Feed the continuous inputs to some normal dense layers.

```
1 random.seed(1337)
2 x = Dense(30, "relu", name="hidden1")(other_inputs)
3 x = Dense(30, "relu", name="hidden2")(x)
4 lambda_ = Dense(1, "exponential", name="lambda")(x)
```

```
1 from keras.layers import Multiply
2
3 out = Multiply(name="out")([lambda_, exposure])
4 model = Model([exposure, other_inputs], out)
5 model.compile(optimizer="adam", loss="poisson")
6
7 es = EarlyStopping(patience=10, restore_best_weights=True, verbose=1)
8 hist = model.fit((X_train_exp, X_train_rest),
9                 y_train, epochs=100, verbose=0,
10                 callbacks=[es], validation_split=0.2)
11 np.min(hist.history["val_loss"])
```

Epoch 40: early stopping

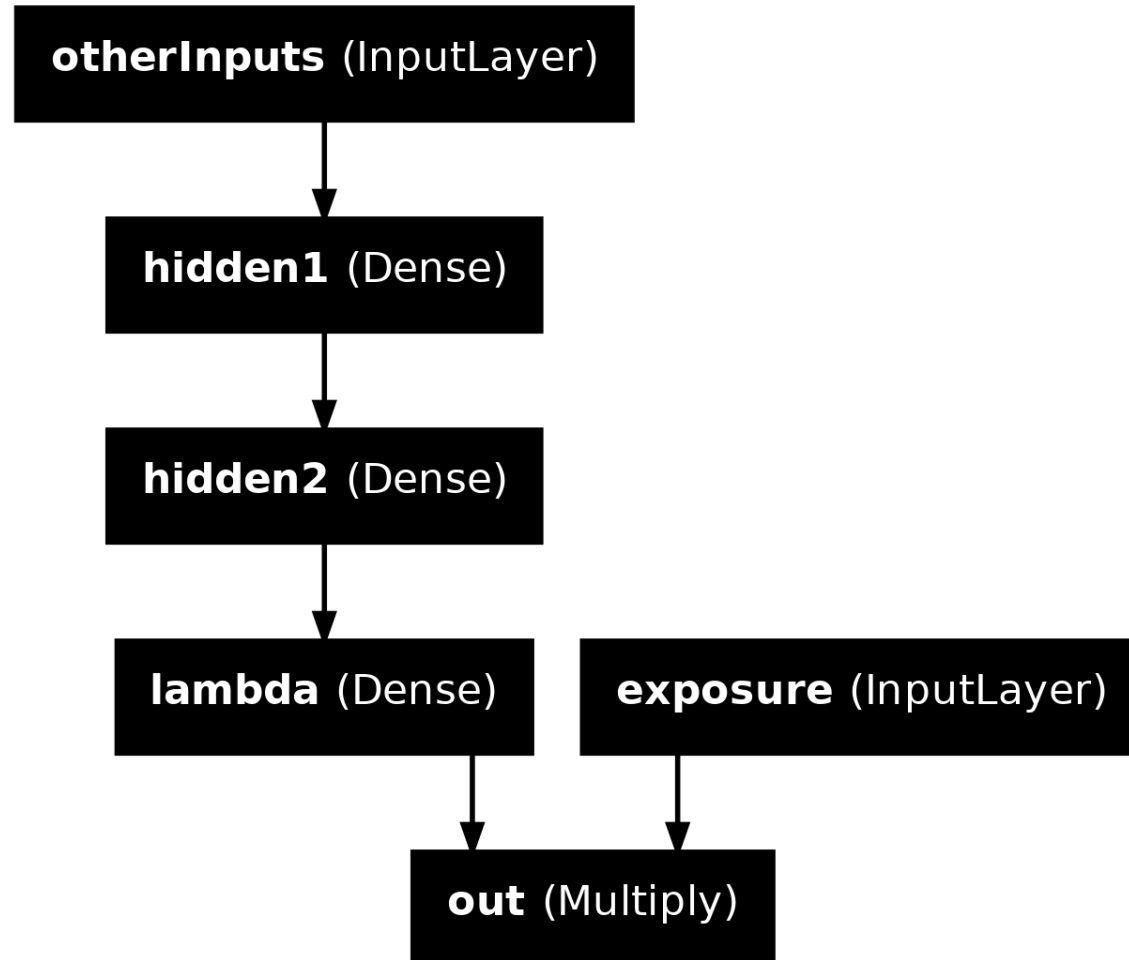
Restoring model weights from the end of the best epoch: 30.

0.8829042911529541



# Plot the model

```
1 plot_model(model, show_layer_names=True)
```



# 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      : 2.0.1
tensorflow: 2.16.1
tf_keras   : 2.16.0
```





# Glossary

- entity embeddings
- Input layer
- Keras functional API
- Reshape layer
- skip connection
- wide & deep network structure

