Advanced Tabular Data

ACTL3143 & ACTL5111 Deep Learning for Actuaries

Patrick Laub

UNSW SYDNEY

Lecture Outline

- **Entity Embedding**
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure
- Dropout

Continuing on the French motor dataset example

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

```python
from pathlib import Path
from sklearn.datasets import fetch_openml

if not Path("french-motor.csv").exists():
    freq = fetch_openml(data_id=41214, as_frame=True).frame
    freq.to_csv("french-motor.csv", index=False)
else:
    freq = pd.read_csv("french-motor.csv")

freq
```

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

Continuing on the French motor dataset example

	IDpol	ClaimNb	Exposure	Area	VehPower	VehAg
0	1.0	1	0.10000	D	5	0
1	3.0	1	0.77000	D	5	0
…	…	…	…	…	…	…
678011	6114329.0	0	0.00274	B	4	0
678012	6114330.0	0	0.00274	B	7	6

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 km² in the city of the living place of the driver
- `Region`: regions in France (prior to 2016)

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

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

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

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, $\text{Exposure}_i \not\in \mathbf{x}_i$, and $\lambda(\mathbf{x}_i)$ is the expected rate *per year*, then

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

Where are things defined?

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

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

is the same as

```
from keras.activations import relu, exponential

model = Sequential([
    Dense(30, activation=relu),
    Dense(1, activation=exponential)
])
```

```
x = [-1.0, 0.0, 1.0]
print(relu(x))
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

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

it is equivalent to

```
from keras.losses import poisson
from keras.optimizers import Adam

model.compile(optimizer=Adam(), loss=poisson)
```

Why do this manually? To adjust the object:

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

or to get help.

Keras' "poisson" loss

```python
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:
```

Subsample and split

```
freq = freq.drop("IDpol", axis=1).head(25_000)

X_train, X_test, y_train, y_test = train_test_split(
  freq.drop("ClaimNb", axis=1), freq["ClaimNb"], random_state=2023)

# Reset each index to start at 0 again.
X_train = X_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
```

What values do we see in the data?

```python
X_train["Area"].value_counts()
X_train["VehBrand"].value_counts()
X_train["VehGas"].value_counts()
X_train["Region"].value_counts()
```

```
Area
C    5507
D    4113
     ...
B    2359
F     475
Name: count, Length: 6, dtype: int64
```

```
VehBrand
B1     5069
B2     4838
       ...
B11     284
B14     136
Name: count, Length: 11, dtype: int64
```

```
VehGas
'Regular'    10773
'Diesel'      7977
Name: count, dtype: int64
```

```
Region
R24    6498
R82    2119
       ...
R42      55
R43      26
Name: count, Length: 22, dtype: int64
```

Preprocess ordinal & continuous

```
from sklearn.compose import make_column_transformer

ct = make_column_transformer(
  (OrdinalEncoder(), ["Area", "VehGas"]),
  ("drop", ["VehBrand", "Region"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train_ct = ct.fit_transform(X_train)
```

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

	Exposure	Area	VehPower
0	1.00	C	6
1	0.36	C	4
2	0.02	E	12

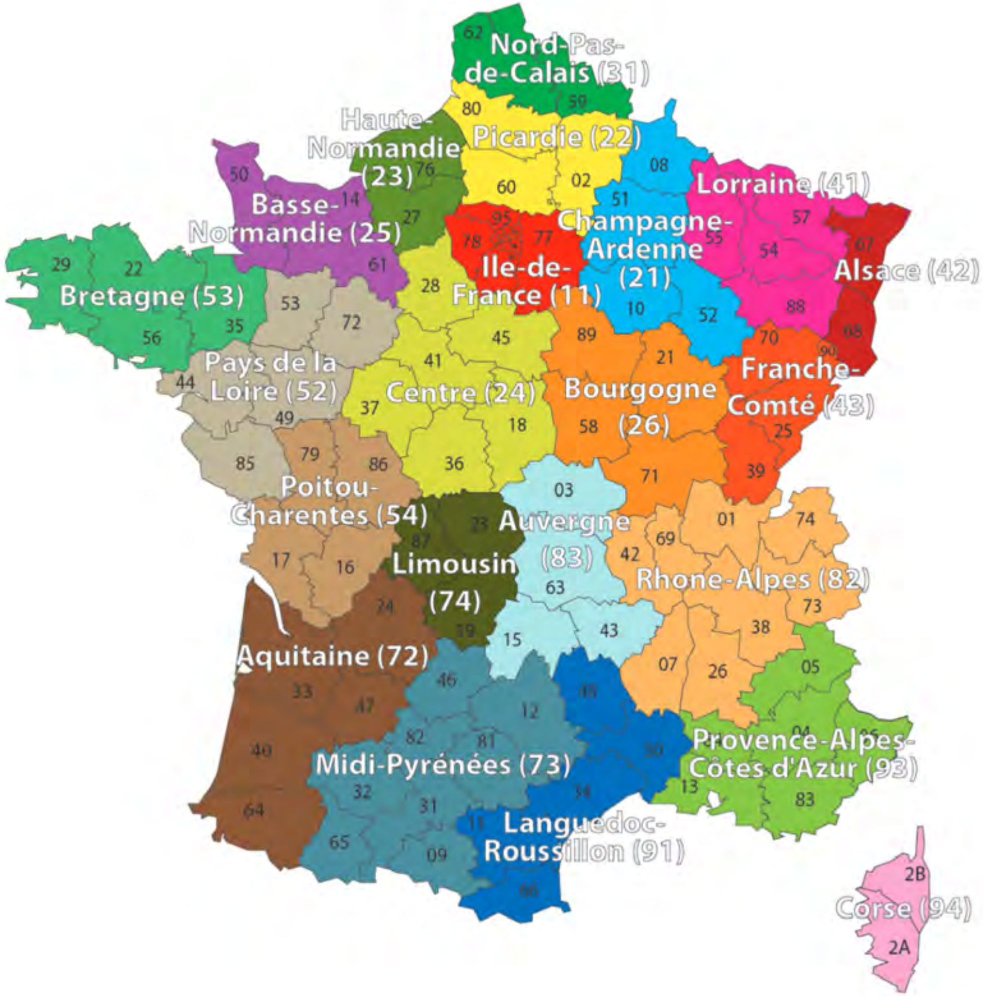
```
X_train_ct.head(3)
```

	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

- Entity Embedding
- **Categorical Variables & Entity Embeddings**
- Keras’ Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure
- Dropout

Region column



French Administrative Regions

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

One-hot encoding

```python
oe = OneHotEncoder(sparse_output=False)
X_train_oh = oe.fit_transform(X_train[["Region"]])
X_test_oh = oe.transform(X_test[["Region"]])
print(list(X_train["Region"][: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
…	…	…	…	…	…
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

```python
num_regions = len(oe.categories_[0])

random.seed(12)
model = Sequential([
  Dense(2, input_dim=num_regions),
  Dense(1, activation="exponential")
])

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

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

```
Epoch 12: early stopping
```

```
0.7526934146881104
```

Consider the first layer

```python
every_category = pd.DataFrame(np.eye(num_regions), columns=oe.categories_[0])
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.
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.
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.

3 rows × 22 columns

```python
# Put this through the first layer of the model
X = every_category.to_numpy()
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],
```

The first layer

```
layer = model.layers[0]
W, b = layer.get_weights()
X.shape, W.shape, b.shape
```

```
((22, 22), (22, 2), (2,))
```

```
X @ 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],
```

```
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],
```

Just a look-up operation

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

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

```
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

```python
oe = OrdinalEncoder()
X_train_reg = oe.fit_transform(X_train[["Region"]])
X_test_reg = oe.transform(X_test[["Region"]])

for i, reg in enumerate(oe.categories_[0][:3]):
  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

```python
from keras.layers import Embedding
num_regions = len(np.unique(X_train[["Region"]]))

random.seed(12)
model = Sequential([
  Embedding(input_dim=num_regions, output_dim=2),
  Dense(1, activation="exponential")
])

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

UNSW SYDNEY

Fitting that model

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

```
Epoch 5: early stopping
```

```
0.7526668906211853
```

```python
model.layers
```

```
[<Embedding name=embedding, built=True>, <Dense name=dense_6, built=True>]
```

Keras’ Embedding Layer

```
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],
```

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

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

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

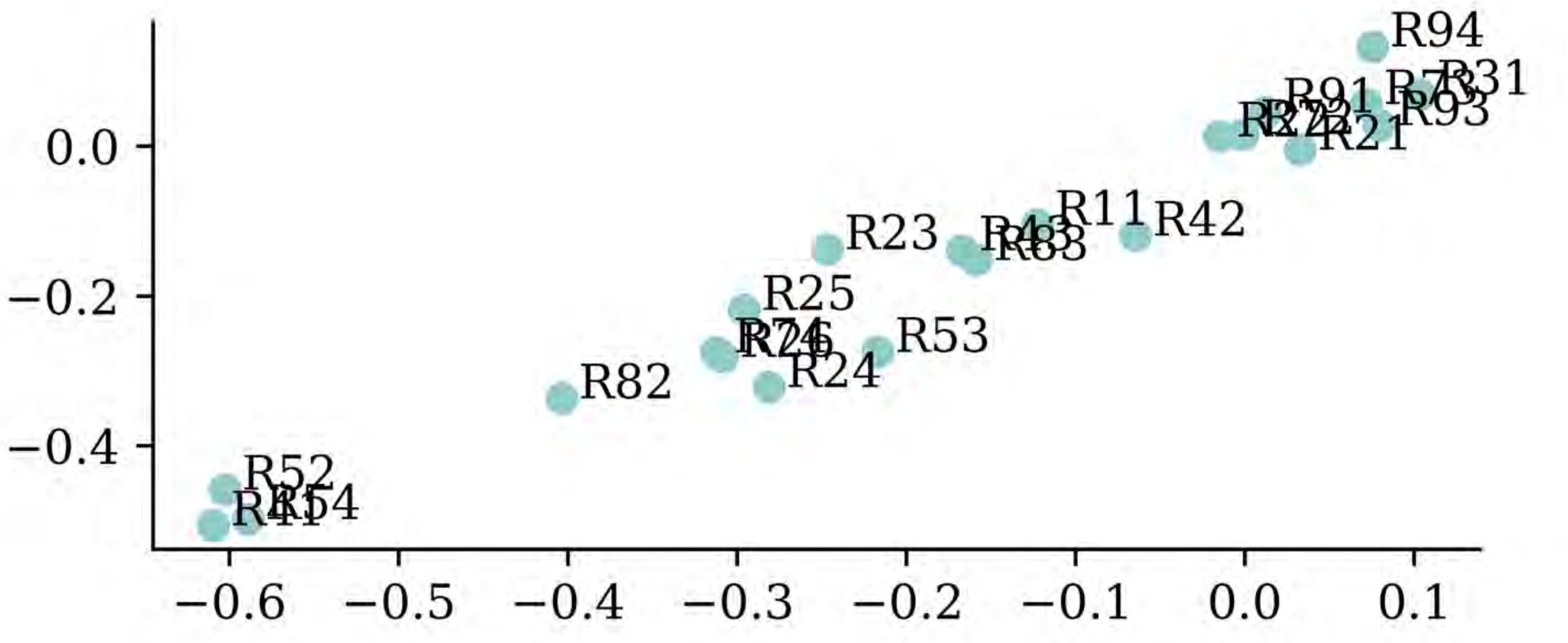
```
array([[ 4.],
       [20.],
       [ 0.],
       [ 9.]])
```

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

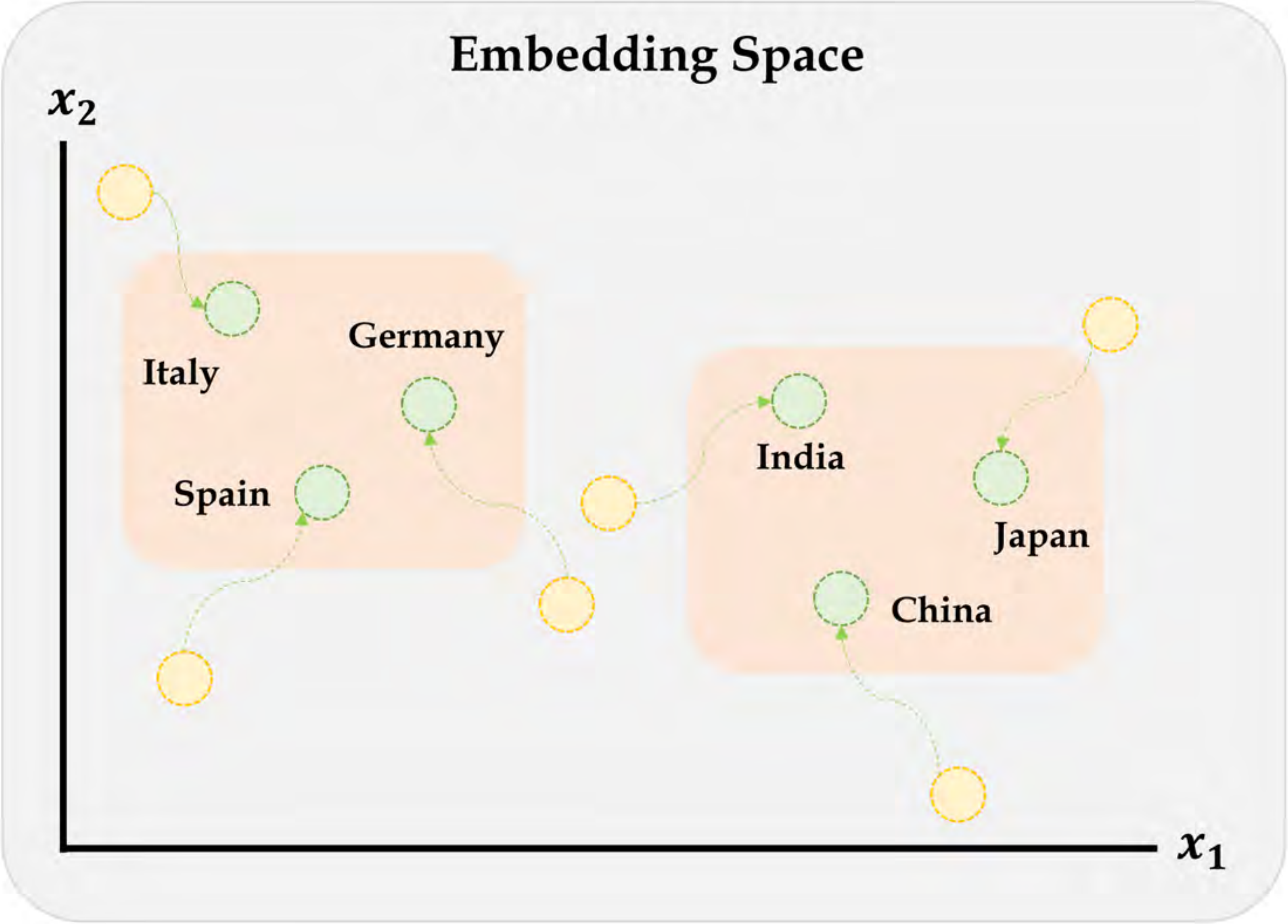
```
array([[-0.28, -0.32],
       [ 0.08,  0.03],
       [-0.12, -0.11],
       [-0.06, -0.12]], dtype=float32)
```

The learned embeddings

```
points = model.layers[0].get_weights()[0]
plt.scatter(points[:,0], points[:,1])
for i in range(num_regions):
  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

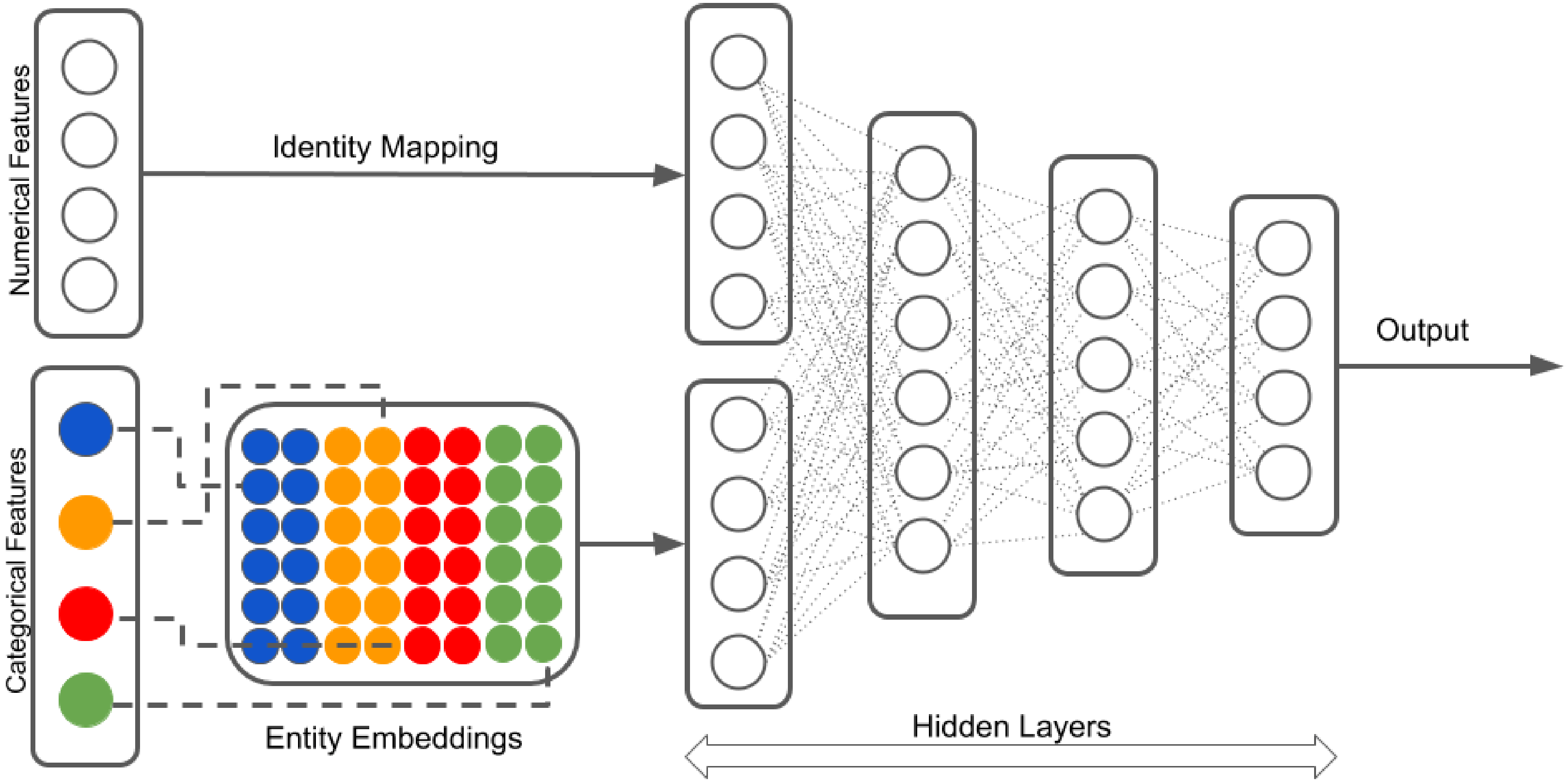


Illustration of a neural network with both continuous and categorical inputs.

We can't do this with Sequential models…

Source: LotusLabs Blog, Accurate insurance claims prediction with Deep Learning.

UNSW SYDNEY

Lecture Outline

- Entity Embedding
- Categorical Variables & Entity Embeddings
- **Keras' Functional API**
- French Motor Dataset with Embeddings
- Scale By Exposure
- Dropout

Converting Sequential models

```python
from keras.models import Model
from keras.layers import Input
```

```python
random.seed(12)

model = Sequential([
  Dense(30, "leaky_relu"),
  Dense(1, "exponential")
])

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

hist = model.fit(
  X_train_oh, y_train,
  epochs=1, verbose=0,
  validation_split=0.2)
hist.history["val_loss"][-1]
```

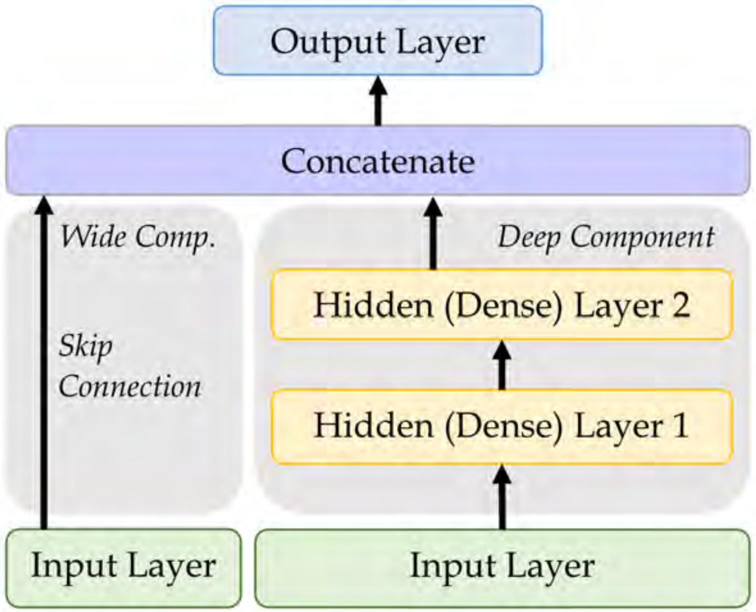
```
0.7535399198532104
```

```python
random.seed(12)

inputs = Input(shape=(X_train_oh.shape[
x = Dense(30, "leaky_relu")(inputs)
out = Dense(1, "exponential")(x)
model = Model(inputs, out)

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

hist = model.fit(
  X_train_oh, y_train,
  epochs=1, verbose=0,
  validation_split=0.2)
hist.history["val_loss"][-1]
```

```
0.7535399198532104
```

See one-length tuples.

Wide & Deep network



An illustration of the wide & deep network architecture.

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

```
from keras.layers \
    import Concatenate

inp = Input(shape=X_train.shape[1:])
hidden1 = Dense(30, "leaky_relu")(inp)
hidden2 = Dense(30, "leaky_relu")(hidden1)
concat = Concatenate()(
  [inp, hidden2])
output = Dense(1)(concat)
model = Model(
    inputs=[inp],
    outputs=[output])
```

Naming the layers

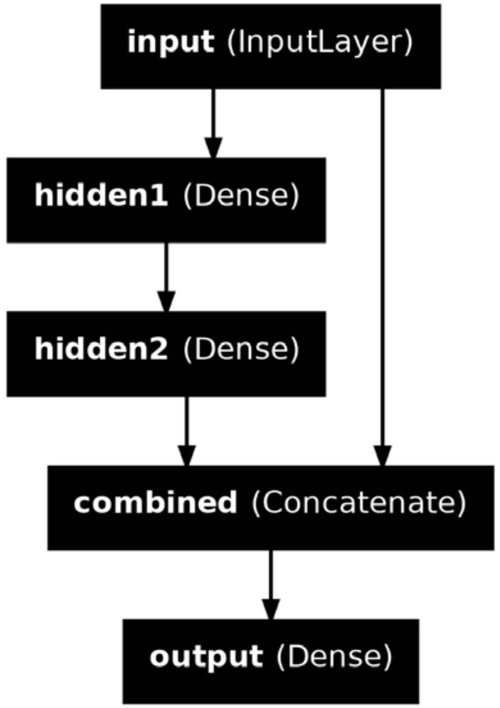
For complex networks, it is often useful to give meaningul names to the layers.

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

Inspecting a complex model

```python
from keras.utils import plot_model
```

```python
plot_model(model, sh
```




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

Model: "functional_8"

Layer (type)	Output Shape	Param #	Connecte
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][hidden2[0
output (Dense)	(None, 1)	41	combined[

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

Lecture Outline

- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- **French Motor Dataset with Embeddings**
- Scale By Exposure
- Dropout

The desired architecture

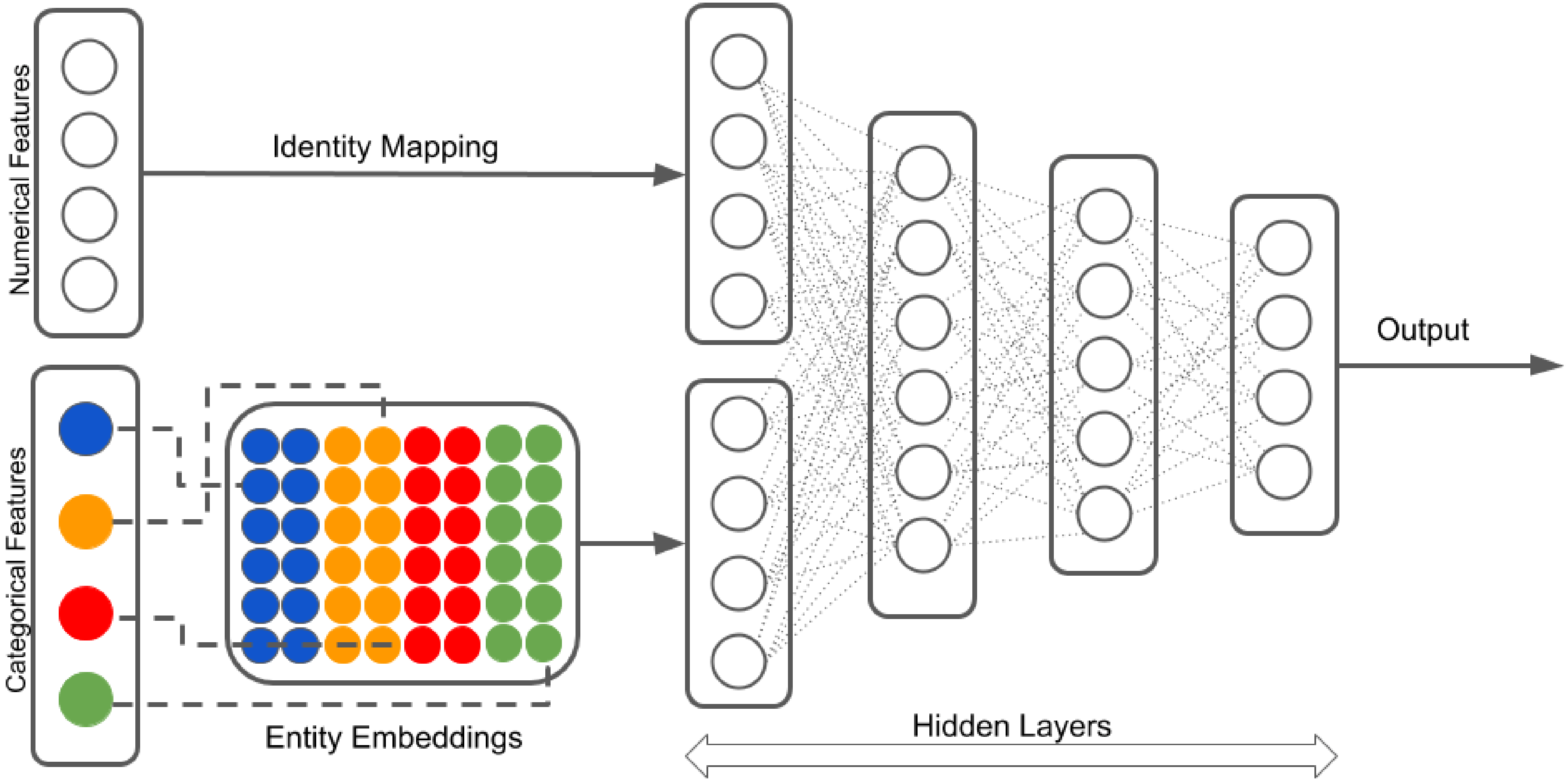


Illustration of a neural network with both continuous and categorical inputs.

Source: LotusLabs Blog, Accurate insurance claims prediction with Deep Learning.

Preprocess all French motor inputs

Transform the categorical variables to integers:

```
num_brands, num_regions = X_train.nunique()[["VehBrand", "Region"]]

ct = make_column_transformer(
  (OrdinalEncoder(), ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train_ct = ct.fit_transform(X_train)
X_test_ct = ct.transform(X_test)
```

Split the brand and region data apart from the rest:

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

UNSW SYDNEY

Organise the inputs

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

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

Create embeddings and join them with the other inputs.

```python
from keras.layers import Reshape

random.seed(1337)
veh_brand_ee = Embedding(input_dim=num_brands, output_dim=2,
    name="vehBrandEE")(veh_brand)
veh_brand_ee = Reshape(target_shape=(2,))(veh_brand_ee)

region_ee = Embedding(input_dim=num_regions, output_dim=2,
    name="regionEE")(region)
region_ee = Reshape(target_shape=(2,))(region_ee)

x = Concatenate(name="combined")([veh_brand_ee, region_ee, other_inputs])
```

Complete the model and fit it

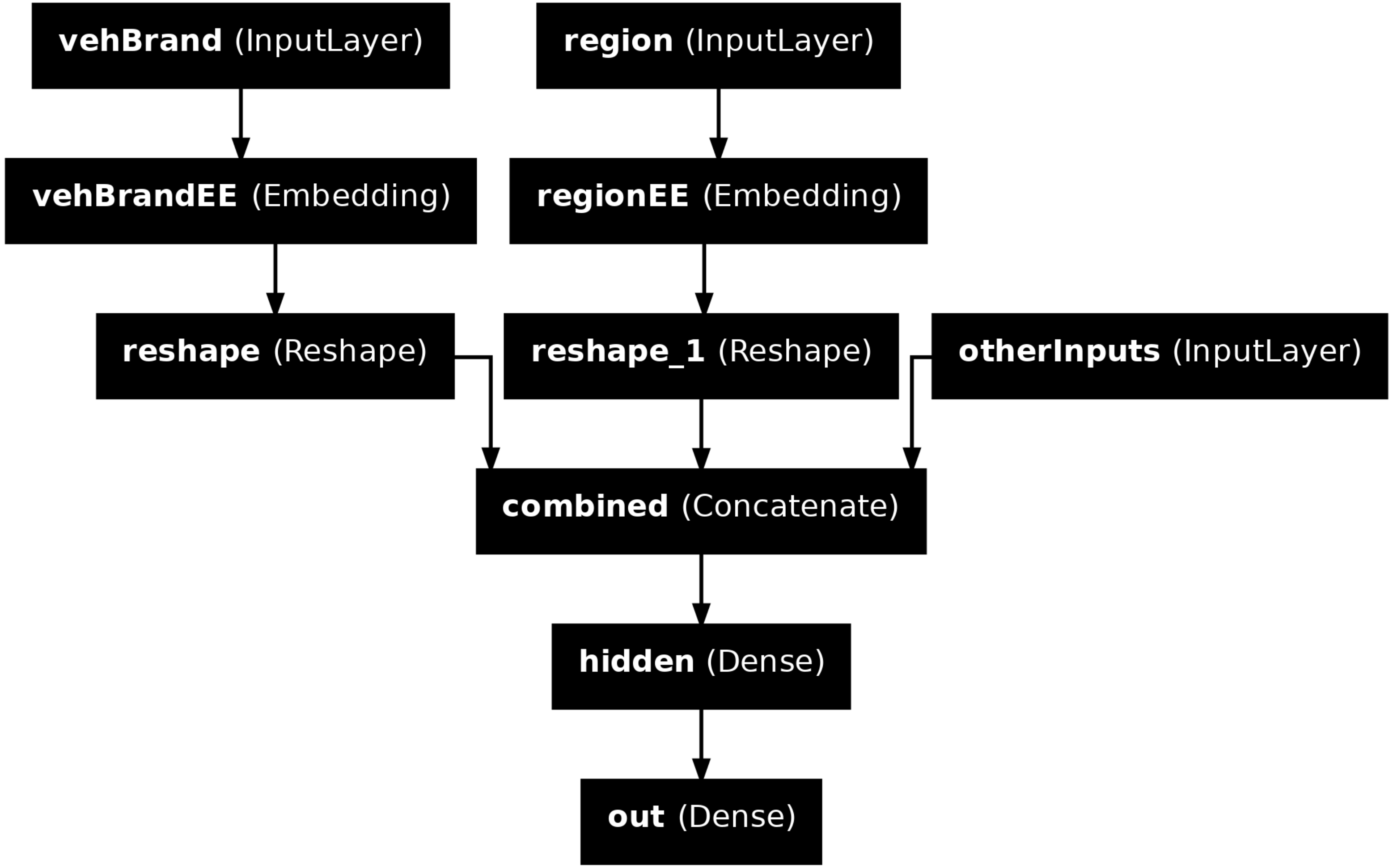
Feed the combined embeddings & continuous inputs to some normal dense layers.

```python
x = Dense(30, "relu", name="hidden")(x)
out = Dense(1, "exponential", name="out")(x)

model = Model([veh_brand, region, other_inputs], out)
model.compile(optimizer="adam", loss="poisson")

hist = model.fit((X_train_brand, X_train_region, X_train_rest),
    y_train, epochs=100, verbose=0,
    callbacks=[EarlyStopping(patience=5)], validation_split=0.2)
np.min(hist.history["val_loss"])
```

```
0.6692155599594116
```

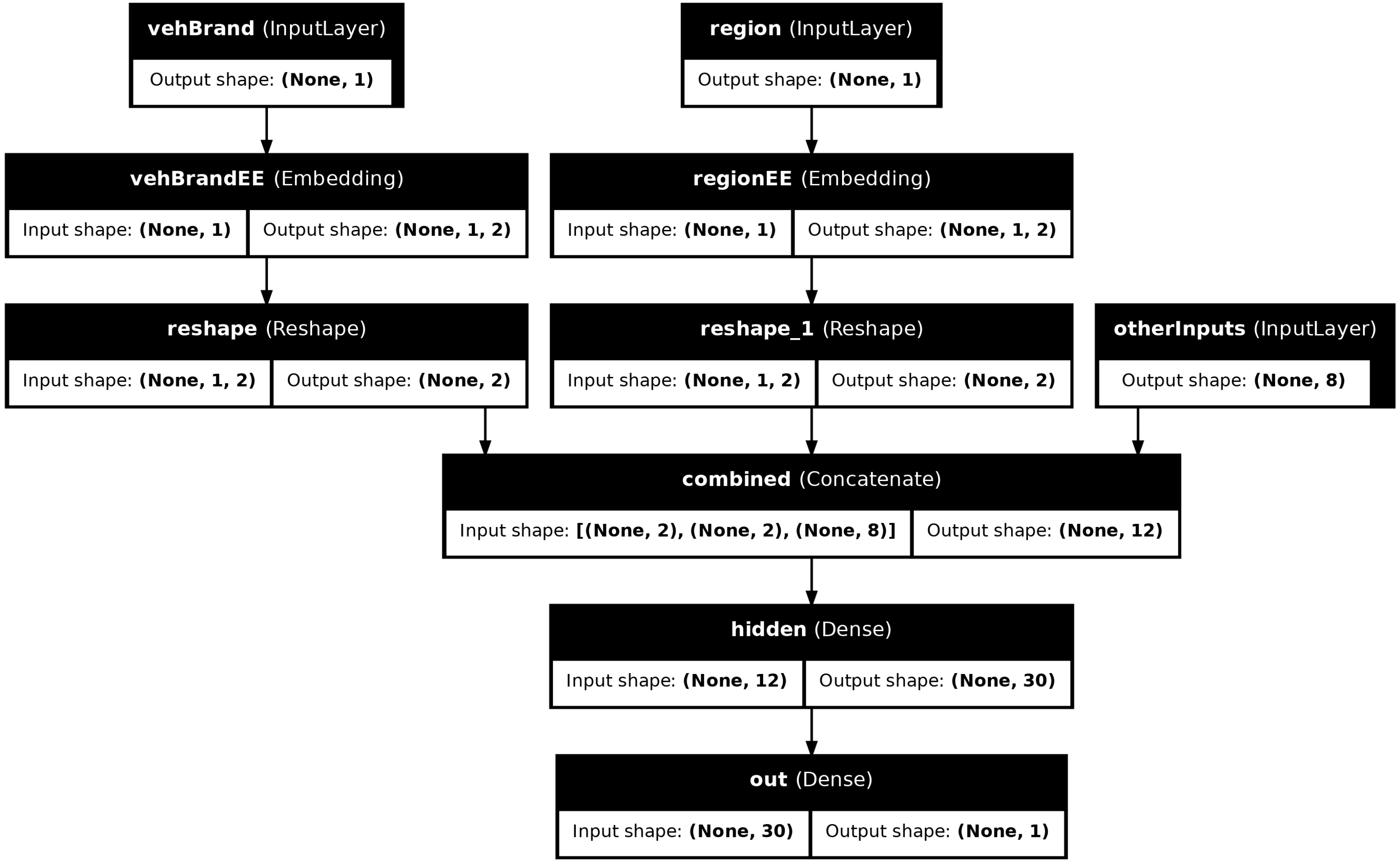
Plotting this model

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



Why we need to reshape

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



Lecture Outline

- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- **Scale By Exposure**
- Dropout

Two different models

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

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

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

Now, $\text{Exposure}_i \not\in \mathbf{x}_i$, and $\lambda(\mathbf{x}_i)$ is the rate *per year*.

Just take continuous variables

```python
ct = make_column_transformer(
  ("passthrough", ["Exposure"]),
  ("drop", ["VehBrand", "Region", "Area", "VehGas"]),
  remainder=StandardScaler(),
  verbose_feature_names_out=False
)
X_train_ct = ct.fit_transform(X_train)
X_test_ct = ct.transform(X_test)
```

Split exposure apart from the rest:

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

Organise the inputs:

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

Make & fit the model

Feed the continuous inputs to some normal dense layers.

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

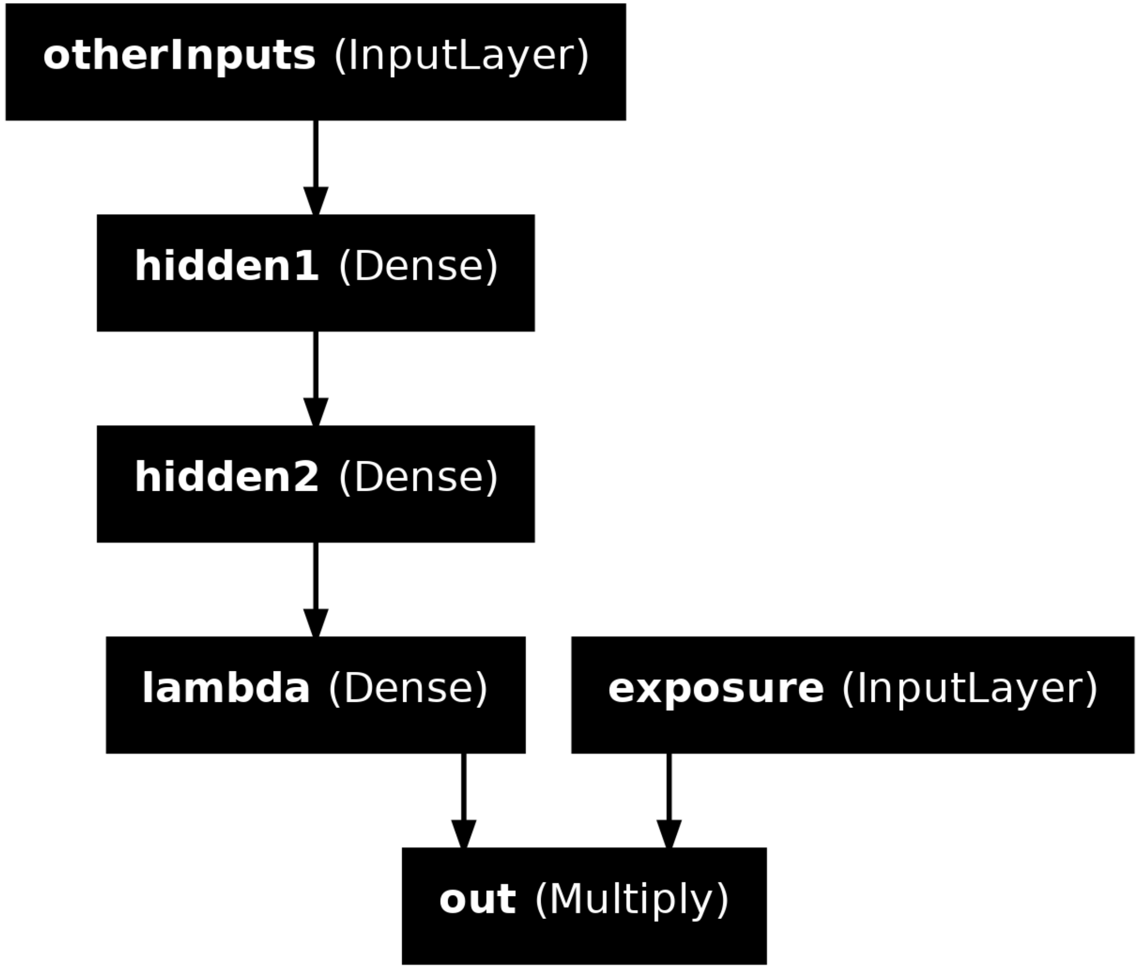
```python
from keras.layers import Multiply

out = Multiply(name="out")([lambda_, exposure])
model = Model([exposure, other_inputs], out)
model.compile(optimizer="adam", loss="poisson")

es = EarlyStopping(patience=10, restore_best_weights=True, verbose=1)
hist = model.fit((X_train_exp, X_train_rest),
    y_train, epochs=100, verbose=0,
    callbacks=[es], validation_split=0.2)
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

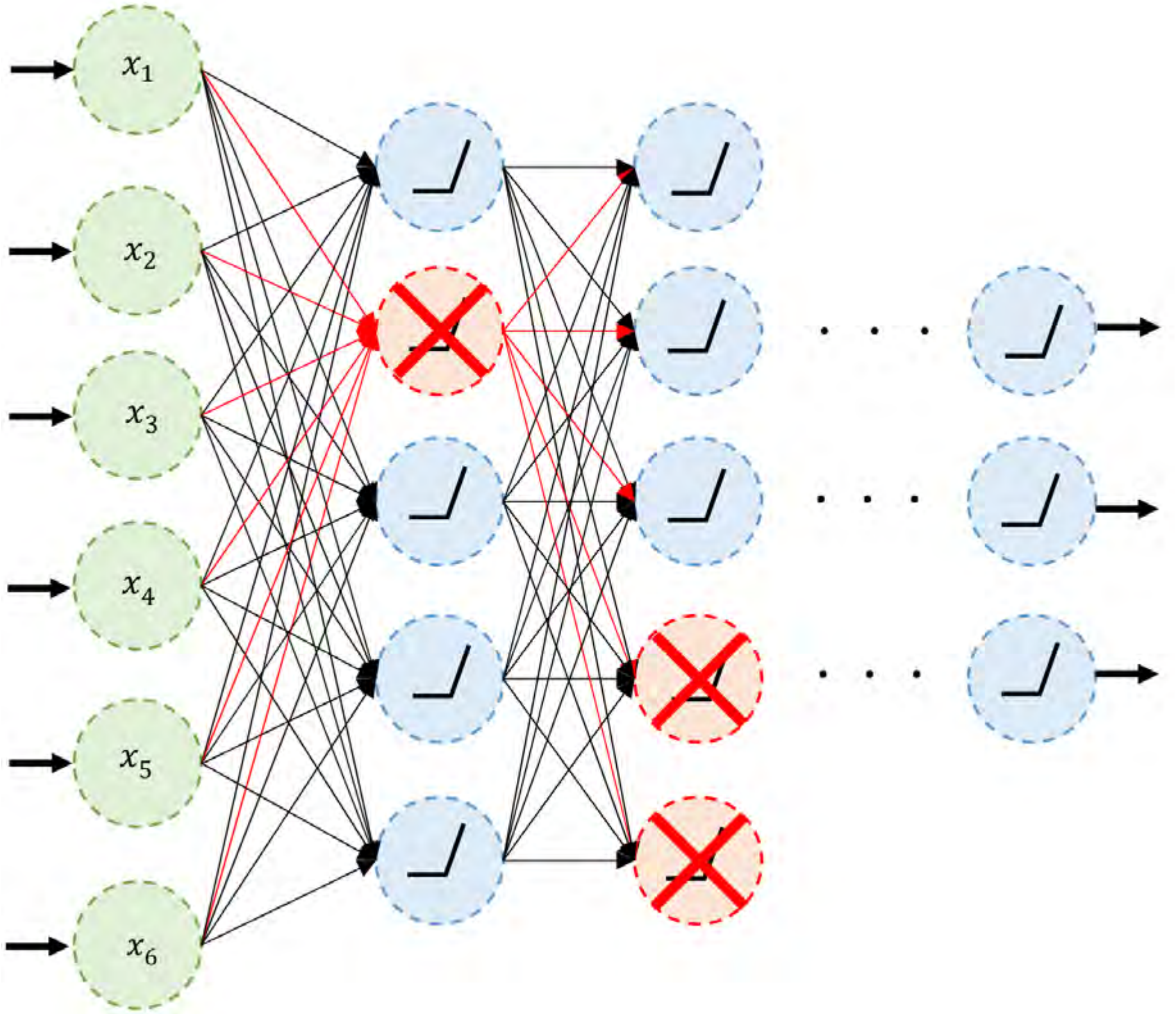
```python
plot_model(model, show_layer_names=True)
```



Lecture Outline

- Entity Embedding
- Categorical Variables & Entity Embeddings
- Keras' Functional API
- French Motor Dataset with Embeddings
- Scale By Exposure
- **Dropout**

Dropout



An example of neurons dropped during training.

Sources: Marcus Lautier (2022).

Dropout quote #1

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

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

Dropout quote #2

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

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

Code: Dropout

Dropout is just another layer in Keras.

```python
from keras.layers import Dropout

random.seed(2);

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

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

Code: Dropout after training

Making predictions is the same as any other model:

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

```
array([[0.64],
       [0.14],
       [0.46]], dtype=float32)
```

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

```
array([[0.64],
       [0.14],
       [0.46]], dtype=float32)
```

We can make the model think it is still training:

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

```
array([[0.64],
       [0.11],
       [0.48]], dtype=float32)
```

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

```
array([[0.68],
       [0.16],
       [0.55]], 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.

Deep Ensembles

It's simple to implement and requires very little hyperparameter tuning.

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

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

> **Note**
>
> Unfinished section… Please imagine some wildly interesting content here.

Package Versions

```python
from watermark import watermark
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
tf_keras  : 2.16.0
```

Glossary

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