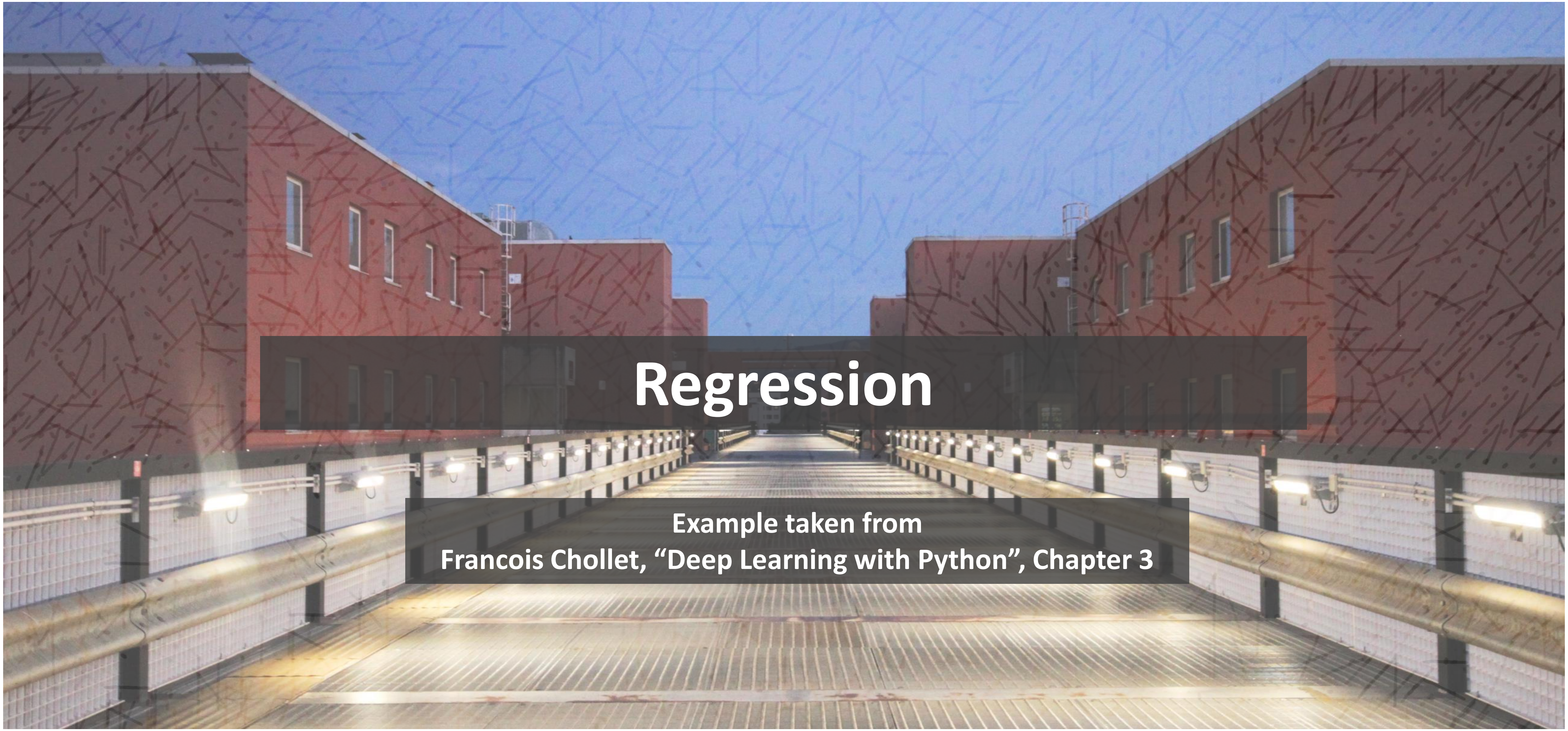
UNIVERSITÀ DELLA CALABRIA



Regression

Example taken from
Francois Chollet, “Deep Learning with Python”, Chapter 3

Boston Housing Price dataset

- Predict the median price of homes in a given Boston suburb in the mid-1970s, given a few data points about the suburb at the time:

1	crime	per capita crime rate by town.	8	dis	weighted mean of distances to five Boston employment centres.
2	zn	proportion of residential land zoned for lots over 25,000 sq.ft.	9	rad	index of accessibility to radial highways.
3	indus	proportion of non-retail business acres per town.	10	tax	full-value property-tax rate per $10,000.
4	chas	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).	11	ptratio	pupil-teacher ratio by town.
5	nox	nitrogen oxides concentration	12	black	1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town.
6	rm	average number of rooms per dwelling.	13	lstat	lower status of the population (percent).
7	age	proportion of owner-occupied units built prior to 1940.			

Main Issues with the Dataset

- Very few data points, only 506 in total
    - 404 training samples and 102 test samples
- Each "feature" in the input data (e.g. the crime rate is a feature) has a different scale

Loading the Dataset

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =
boston_housing.load_data()
```

- The targets are the median values of owner-occupied homes, in thousands of dollars:

```
train_targets
```

```
array([15.2, 42.3, 50. , 21.1, 17.7, 18.5,
11.3, 15.6, 15.6, 14.4, 12.1, 17.9, 23.1,
19.9, 15.7, 8.8, 50. , 22.5, 24.1, 27.5,
10.9, 30.8, 32.9, 24. , 18.5, 13.3, 22.9,
34.7, 16.6, 17.5, 22.3, 16.1, 14.9, 23.1,
…])
```

Prepare the Data: Normalization

- The feature is centered around 0 and has a unit standard deviation
- Note that the quantities (mean, std) used for normalizing the test data are computed using the training data!

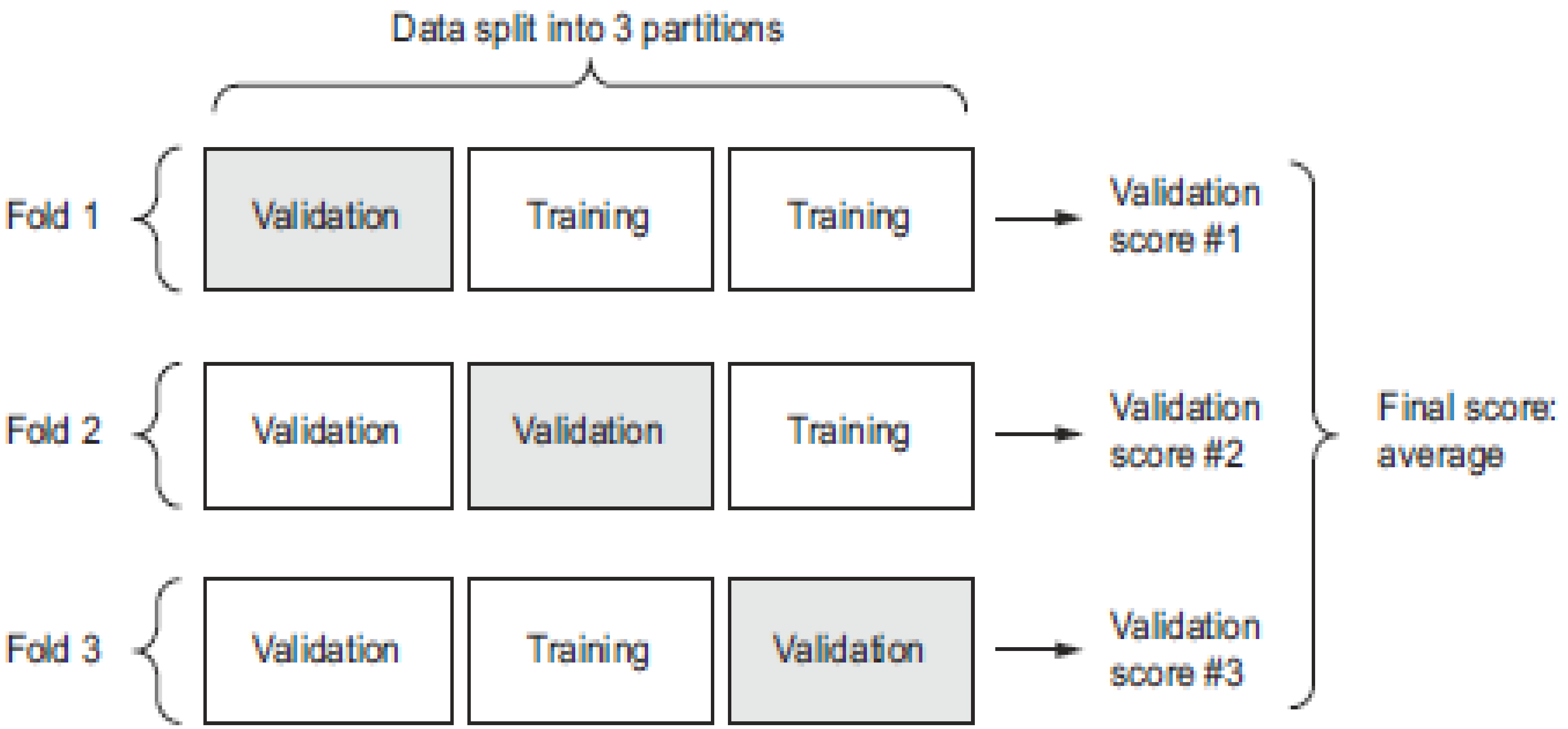
```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

Building the Network

- The network ends with a single unit and no activation
- Loss function: Mean-Squared Error (mse)
- Metrics: Mean Absolute Error (MAE)

```
from keras import models
from keras import layers

def build_model():
    # Because we will need to instantiate
    # the same model multiple times,
    # we use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Validation with Few Data Points



Implementation of K-fold Validation

```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    # Build the Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    # Evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Results

```
all_scores
```

```
[2.1265724118393248, 2.5035528570118517, 2.568339293546016, 2.6296658681170775]
```

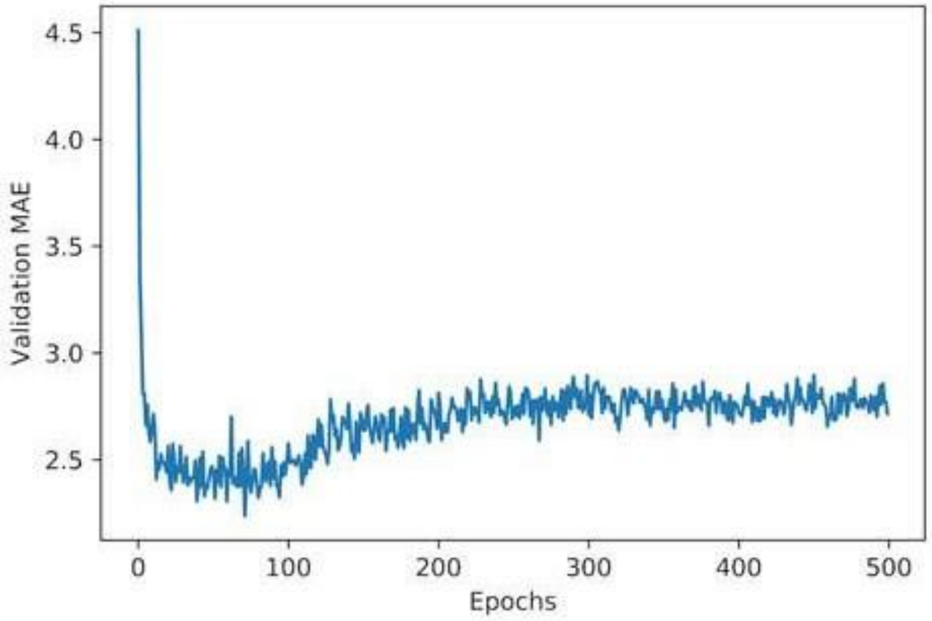
Save the History

```
all_mae_histories = []
for i in range(k):
    ...
    ...
    history = model.fit(partial_train_data, partial_train_targets,
                        epochs=num_epochs, batch_size=1, verbose=0)
    ...

    # For visualization
    mae_history = history.history['mean_absolute_error']
    all_mae_histories.append(mae_history)
```

Visualize the average MAE score

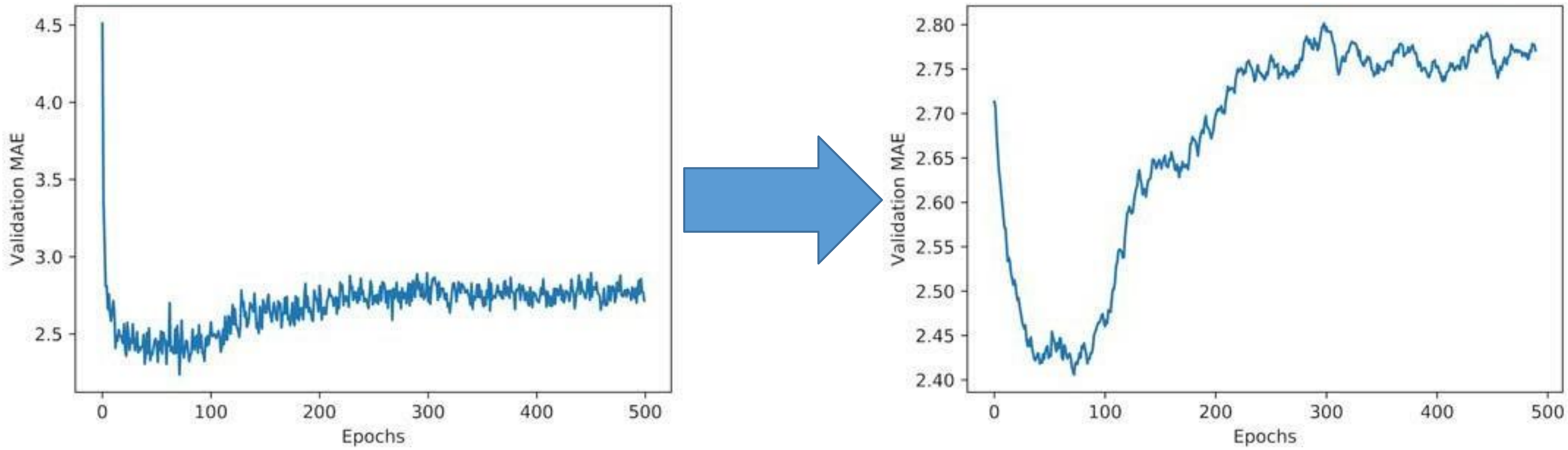
```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

```
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



A closer look

- Omit the first 10 data points

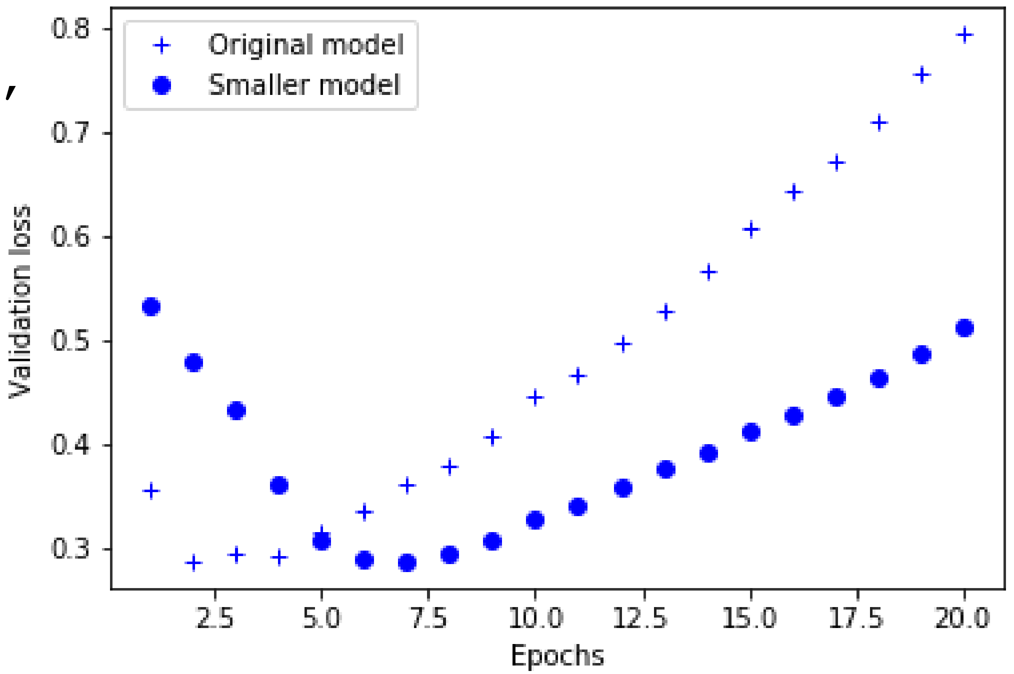


UNIVERSITÀ DELLA CALABRIA

Overfitting

Reduce the Network size

```
from keras import models
from keras import layers

original_model = models.Sequential()
original_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
original_model.add(layers.Dense(16, activation='relu'))
original_model.add(layers.Dense(1, activation='sigmoid'))

original_model.compile(optimizer='rmsprop',
                       loss='binary_crossentropy',
                       metrics=['acc'])
```



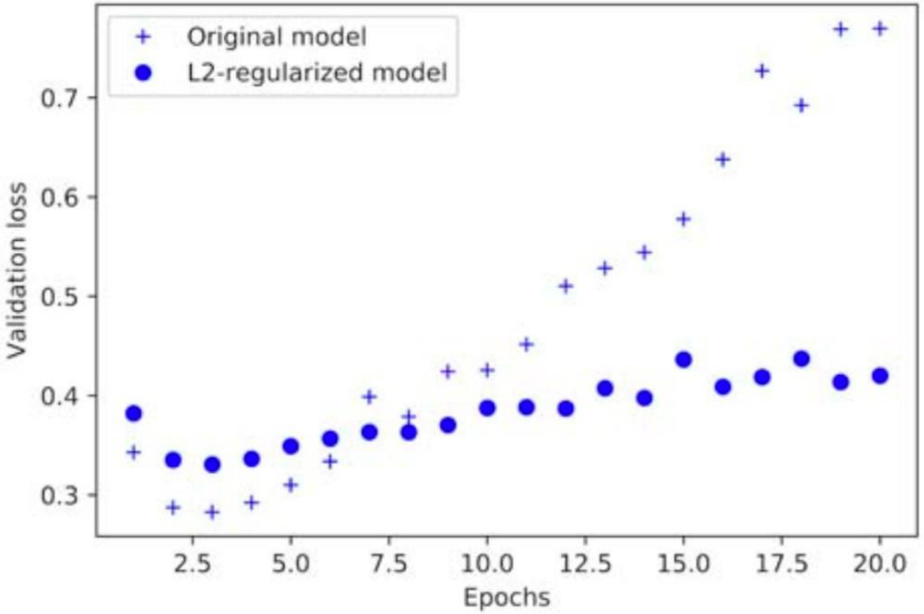
Weight regularization

- Put constraints on the complexity of a network by forcing its weights to only take small values, which makes the distribution of weight values more "regular“:
  - L1 regularization, where the cost added is proportional to the *absolute value of the weights coefficients* $\text{new loss function} = \text{old loss function} + \lambda \sum_i |w_i|$
  - 2 regularization, where the cost added is proportional to the *square of the value of the weights coefficients* $\text{new loss function} = \text{old loss function} + \lambda \sum_i w_i^2$

- A reason for weight regularization: large weight can make the model more sensitive to noise/variance in data.
  - L2 regularization: it tends to make all weights small.
  - L1 regularization: it tends to make weights sparser (namely, more 0s).

L2 Regularization

```
from keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(8, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(8, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))
```

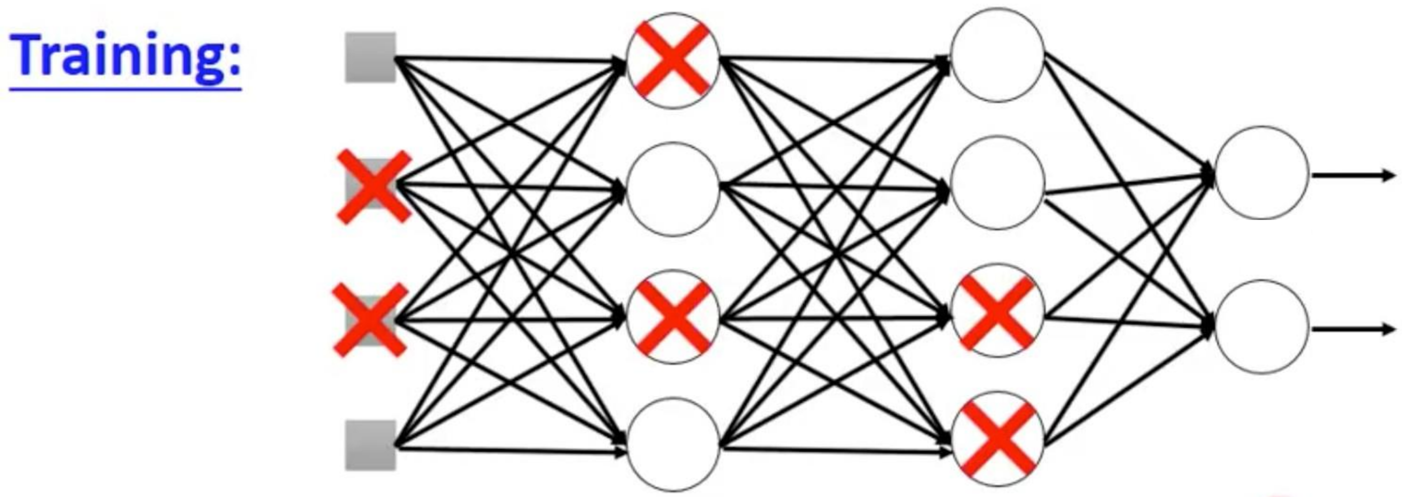


L1 regularization and Combination

```
from keras import regularizers

# L1 regularization
regularizers.l1(0.001)

# L1 and L2 regularization at the same time
regularizers.l1_l2(l1=0.001, l2=0.001)
```

Dropout

- Dropout, applied to a layer, consists of randomly "dropping out" (i.e. setting to zero) a number of output features of the layer during training

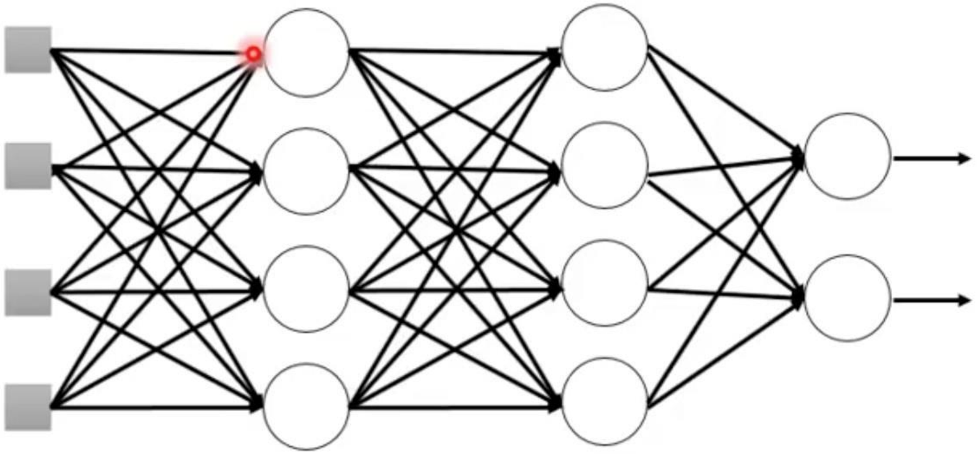


- **Each time before updating the parameters**
  - Each neuron has p% to dropout

Dropout

- At test time, no units are dropped out, and instead the layer's output values are scaled down by a factor equal to the dropout rate, so as to balance for the fact that more units are active than at training time.

Testing:



- No dropout
  - If the dropout rate at training is p%, all the weights times (1-p)%

Dropout

- Intuition: The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that are not significant (what Hinton refers to as "conspiracies"), which the network would start memorizing if no noise was present

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['acc'])
```

Dropout

- Effect on validation loss