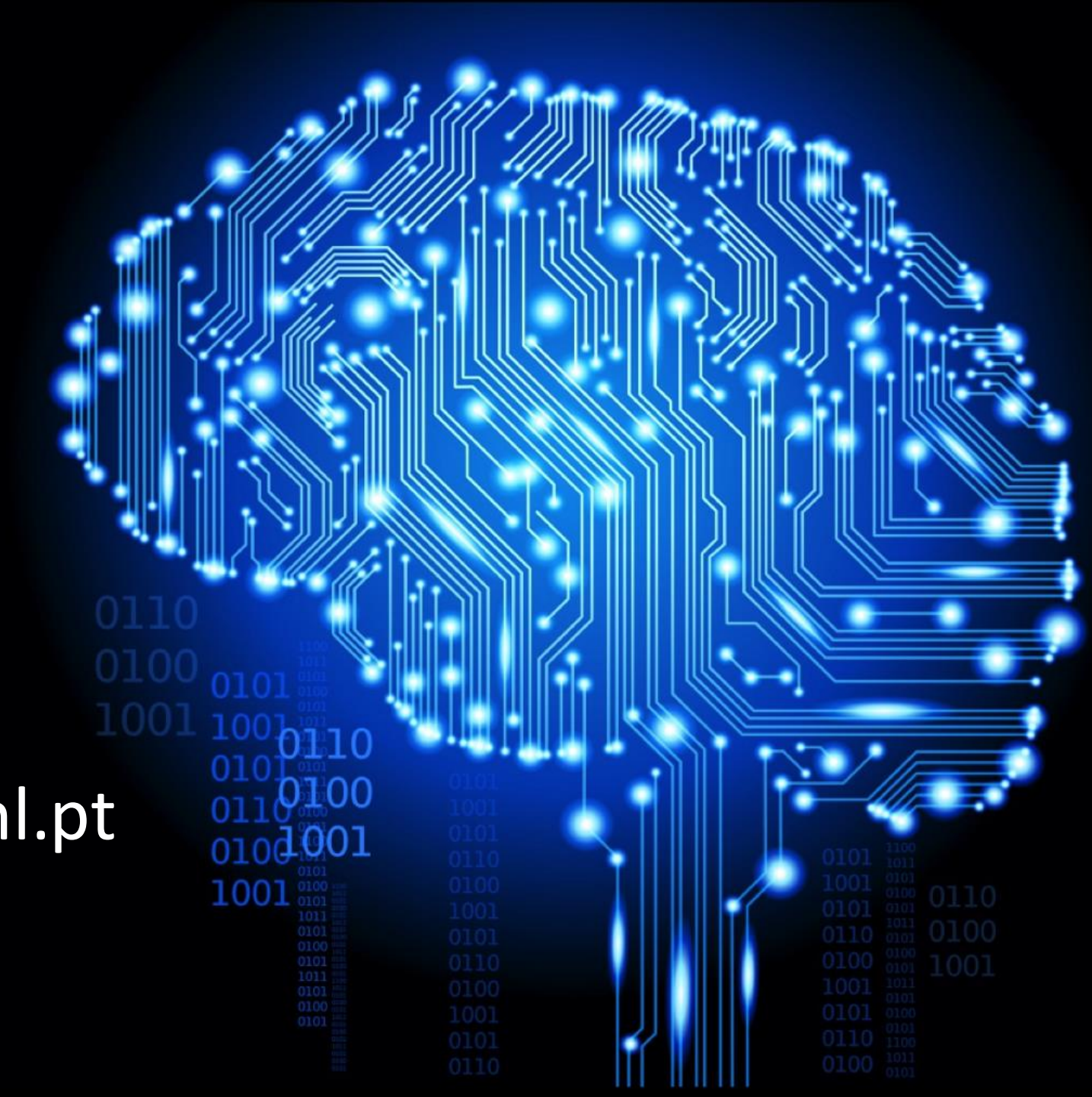
DEEP LEARNING

Spring 2021

MAURO CASTELLI
mcastelli@novaims.unl.pt



Regression

Regression

The two previous examples were considered classification problems, where the goal was to predict a single discrete label of an input data point.

Another common type of machine-learning problem is *regression*, which consists of predicting a continuous value instead of a discrete label.

Regression - Example

We'll attempt to predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on.

The dataset has relatively few data points: only 506, split between 404 training samples and 102 test samples.

Each *feature* in the input data (for example, the crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1; others take values between 1 and 12, others between 0 and 100, and so on.

Loading the Boston housing dataset

```
from keras.datasets import boston_housing

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

Let's look at the data:

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

Loading the Boston housing dataset

As you can see, you have 404 training samples and 102 test samples, each with 13 numerical features, such as per capita crime rate, average number of rooms per dwelling, accessibility to highways, and so on.

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

```
>>> train_targets
[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

The prices are typically between $10000 and $50000. If that sounds cheap, remember that this was the mid-1970s, and these prices aren't adjusted for inflation.

Data Preparation

It would be problematic to feed into a neural network values that all take wildly different ranges.

The network might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult.

A widespread best practice to deal with such data is to do feature-wise normalization:

Normalization

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

Note that the quantities used for normalizing the test data are computed using the training data. You should never use in your workflow any quantity computed on the test data, even for something as simple as data normalization.

Building the Network

Because so few samples are available, we'll use a very small network with two hidden layers, each with 64 units.

In general, the less training data you have, the worse overfitting will be, and using a small network is one way to mitigate overfitting.

```
from keras import models
from keras import layers

def build_model():
    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
```

Because you'll need to instantiate the same model multiple times, you use a function to construct it.

Building the Network

The network ends with a single unit and no activation (it will be a linear layer).

This is a typical setup for scalar regression (a regression where you're trying to predict a single continuous value).

Applying an activation function would constrain the range the output can take; for instance, if you applied a sigmoid activation function to the last layer, the network could only learn to predict values between 0 and 1.

Here, because the last layer is purely linear, the network is free to learn to predict values in any range.

Observations

- Note that you compile the network with the `mse` loss function—*mean squared error*, the square of the difference between the predictions and the targets.

- This is a widely used loss function for regression problems.

- We're also monitoring a new metric during training: *mean absolute error* (MAE). It's the absolute value of the difference between the predictions and the targets.

K-fold validation

To evaluate the network while you keep adjusting its parameters (such as the number of epochs used for training), you could split the data into a training set and a validation set, as we did in the previous examples.

But because we have so few data points, the validation set would end up being very small (for instance, about 100 examples).

As a consequence, the validation scores might change a lot depending on which data points you chose to use for validation and which you chose for training: the validation scores might have a high *variance* with regard to the validation split.

This would prevent you from reliably evaluating your model.

K-fold validation

The best practice in such situations is to use *K-fold* cross-validation.

It consists of splitting the available data into *K* partitions (typically *K* = 4 or 5), instantiating *K* identical models, and training each one on *K* – 1 partitions while evaluating on the remaining partition.

The validation score for the model used is then the average of the *K* validation scores obtained. In terms of Keras code, this is straightforward.

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)
    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]

    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)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

K-fold validation

Running this with num_epochs = 100 yields the following results:

```
>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984,
3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462
```

Let’s try training the network a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, you’ll modify the training loop to save the per-epoch validation score log.

Saving the validation logs at each fold

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    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

    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)

model = build_model()

history = model.fit(partial_train_data, partial_train_targets,
                    validation_data=(val_data, val_targets),
                    epochs=num_epochs, batch_size=1, verbose=0)
mae_history = history.history['val_mean_absolute_error']
all_mae_histories.append(mae_history)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

Building the history of successive mean K-fold validation scores

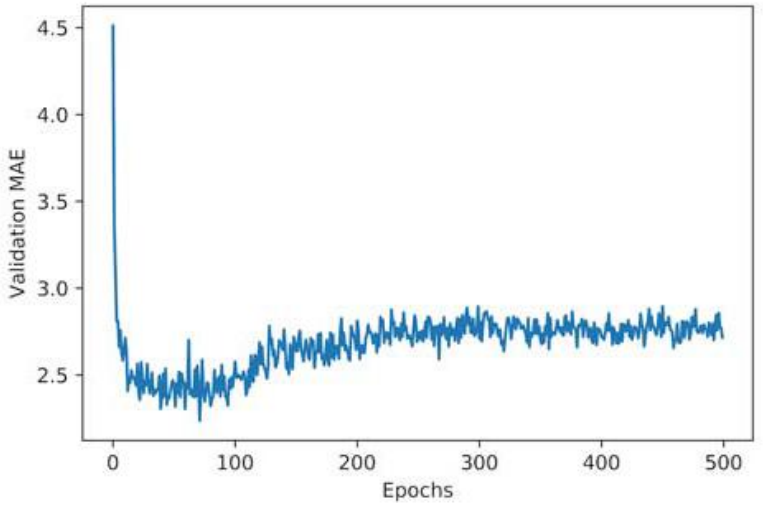
You can then compute the average of the per-epoch MAE scores for all folds.

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

Let’s plot this!

Plotting validation scores

```
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()
```

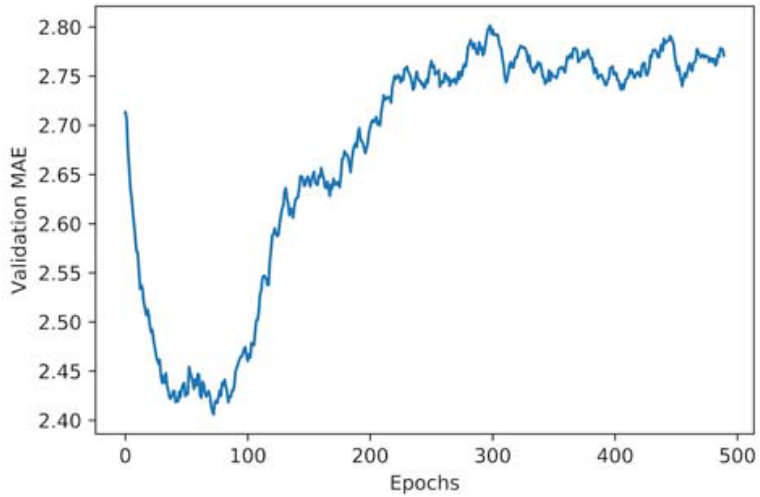


It may be a little difficult to see the plot, due to scaling issues and relatively high variance.

Let's do the following:

- Omit the first 10 data points, which are on a different scale than the rest of the curve.
- Replace each point with an exponential moving average of the previous points, to obtain a smooth curve.

Plotting validation scores, excluding the first 10 data points

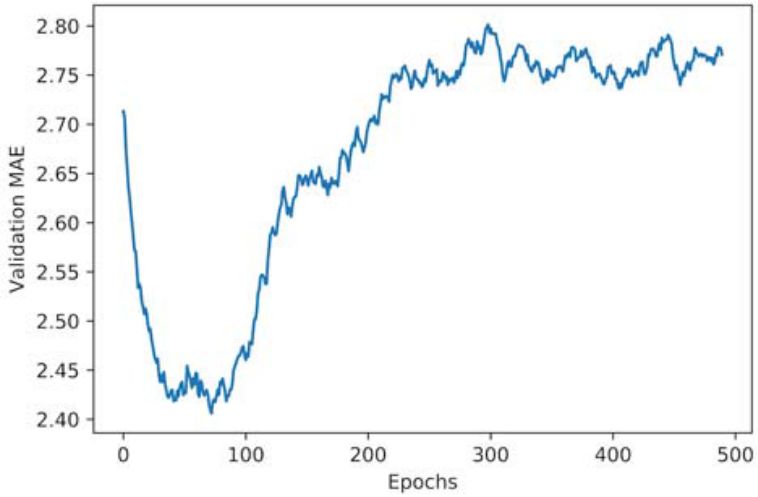



```
def smooth_curve(points, factor=0.9):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

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

Comments

According to this plot, validation MAE stops improving significantly after 80 epochs. Past that point, you start overfitting.

Once you're finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the hidden layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.



Training the final model

Gets a fresh, compiled model

Trains it on the entirety of the data

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Here's the final result:

```
>>> test_mae_score
2.5532484335057877
```

Take home messages

- ✓ Regression is done using different loss functions than what we used for classification.
- ✓ Mean squared error (MSE) is a loss function commonly used for regression.
- ✓ Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn't apply for regression. A common regression metric is mean absolute error (MAE).
- ✓ When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- ✓ When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- ✓ When little training data is available, it's preferable to use a small network with few hidden layers (typically only one or two), in order to avoid severe overfitting.

Summary

- ✓ We're now able to handle the most common kinds of machine-learning tasks on vector data: binary classification, multiclass classification, and regression.
- ✓ As training progresses, NNs eventually begin to overfit and obtain worse results on never-before-seen data.
- ✓ If you don't have much training data, use a small network with only one or two hidden layers, to avoid severe overfitting.
- ✓ If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.
- ✓ Regression uses different loss functions and different evaluation metrics than classification.
- ✓ When you're working with little data, K-fold validation can help evaluate your model.