Modelling of Cognitive Processes

Delta learning

Lesson 06
29/10/2019
Pieter Huycke

Overview

Theoretical

- Delta learning: quick recap

Practical

1. Florence + the machine: a Delta learning tutorial
2. Modelling the blocking effect
3. Flower recognition with `scikit-learn`

Theory

The delta rule

In the last theoretical lesson, we considered the mean squared error (MSE) function, which has the following form:

$$E = \frac{1}{n} \sum_{i=1}^{n} (t_i - y_i)^2$$

where t_i are the values provided by the supervisor.
Hence, this type of learning belongs to the category of **supervised learning**.
Recall that minimizing this function is straightforward: we have to minimize the difference between the predicted values y_i and the 'required' values t_i.

The delta rule

We apply gradient descent in weight space, resulting in the following equation:

$$\Delta w = -\beta \frac{\partial E}{\partial w_{ij}}$$

Mind that the notation $\frac{\partial f(x,y)}{\partial y}$ refers to the partial derivative of the function $f(x, y)$ with respect to variable y.

The delta rule

Working out this equation algebraically brings us to the following equation:

$$\Delta w_{ij} = \beta_j (t_i - y_i) \frac{\partial}{\partial in_i} f(in_i)$$

Which can be simplified if we use the linear activation function to:

$$\Delta w_{ij} = \beta_j (t_i - y_i)$$

Practical

1. Florence + the machine: a Delta learning tutorial

The unknown artist

Imagine you are listening to the radio, and suddenly a song comes up that you really like.
After the song, the radio host mentions the song 'Stand by me' by 'Florence + the machine'.

You decide to search them online, and you find the following information...

Florence + the machine

- English indie rock band
- Formed in London in 2007
- Lead singer: Florence Welch ⬇

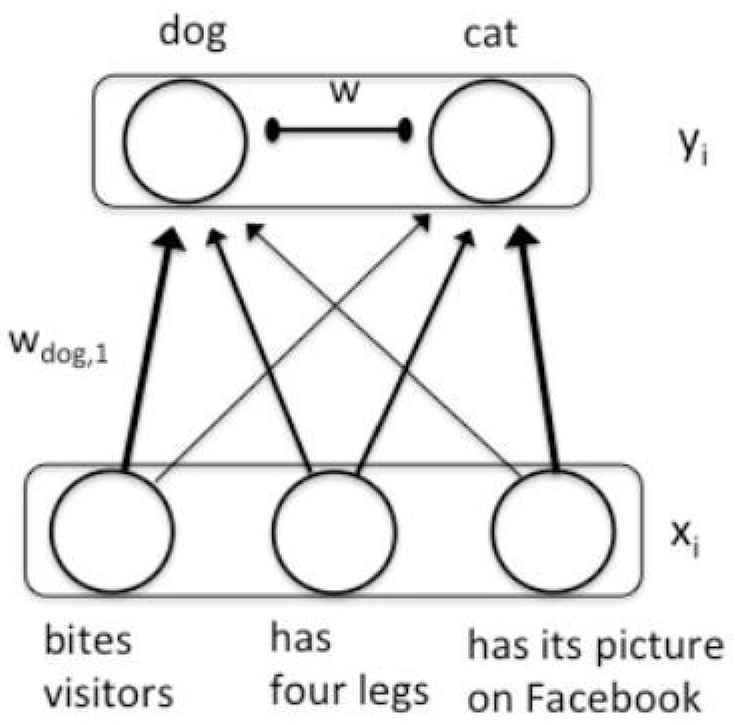
Florence: the modelling aproach

When you now hear *Stand by me* again, you will be able to conjure up Florence's picture in your mind.
In MCP terms: you learned an association between two items.
Please note that encountering one item (the song) will result in the second item (the mental picture of Florence you saw online).

We have already seen these dynamics in the cat-dog model...

Florence: comparison with the pet detector

Note that the pet detector also worked with specific features.



What do we expect here?

```
input = np.array([0, 1, 1])
```

Florence: comparison with the pet detector

Model input

- Unit 1 is **inactive**: does not bite visitors
- Unit 2 is **active**: has 4 legs
- Unit 3 is **active**: has a picture on Facebook

Model output

Florence: the modelling aproach

Mind what happened:

- First, the song was not associated with mental images
- After the Google search, we could picture the singer of this song

How?
Learning

Now, we will represent this learning process in Python 3.

Florence: the modelling aproach

Our action plan:

1. Open Spyder 🕸️
2. Open **'ch4_florence_delta_solution.py'**
3. Notice that "blocks" of code are separated by the `#%%` character
4. Run these blocks of code by clicking inside this block and pressing `shift + enter`
5. Look at the output
6. Sit back and listen to my explanation of each block!

In [1]:
```
# import modules
import ch0_delta_learning as delta_learning
import numpy              as np

# alter print options for numpy: suppress scientific printing
np.set_printoptions(suppress = True)

image_florence   = [.99, .01, .99, .01, .99, .01]     # represents image
song_stand_by_me = [.99, .99, .01, .01]               # represents song

# define a weight matrix exclusively filled with zeros
weight_matrix = delta_learning.initialise_weights(image_florence,
                                                  song_stand_by_me,
                                                  zeros      = True,
                                                  predefined = False,
                                                  verbose    = True)

# show me what you got
print('Our original weight matrix, for now filled with zeros:\n',
      weight_matrix)

# make a copy of the original weight matrix
original_weight_matrix = np.copy(weight_matrix)
```

```
Using zeros to fill the array...

Our original weight matrix, for now filled with zeros:
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

In [2]:
```
# activation associated with the all zero weight matrix
activation_original = delta_learning.internal_input(image_florence,
                                                     weight_matrix)[0]
print('\nActivation levels at output for the original weight matrix:\n',
      activation_original)
```

```
Activation levels at output for the original weight matrix:
 [0.5, 0.5, 0.5, 0.5]
```

```
In [3]: loops = 1000
        alpha = 1.5

        for loop_var in np.arange(1, loops + 1):
            weights_after_learning = delta_learning.weight_change(alpha,
                                                                  image_florence,
                                                                  song_stand_by_me,
                                                                  weight_matrix)
            weight_matrix = weights_after_learning


        print('\nOur altered weight matrix after {} trials of delta learning:\n'.format(loo
        ps),
              weight_matrix)
```

```
Our altered weight matrix after 1000 trials of delta learning:
 [[ 1.31    0.006   1.243   0.004   1.181   0.004]
 [ 1.31    0.006   1.243   0.004   1.181   0.004]
 [-1.31   -0.006  -1.243  -0.004  -1.181  -0.004]
 [-1.31   -0.006  -1.243  -0.004  -1.181  -0.004]]
```

```
In [4]: # activation associated with this altered weight matrix
        activation_after_learning = delta_learning.internal_input(image_florence,
                                                                   weight_matrix)[0]
        print('\nActivation levels at output after {} trials of delta learning:\n'.format(l
        oops),
              np.round(activation_after_learning, 3))
```

```
Activation levels at output after 1000 trials of delta learning:
 [0.976 0.976 0.024 0.024]
```

2. Modelling the blocking effect

Basic classical conditioning

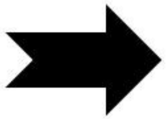
We consider the following situation

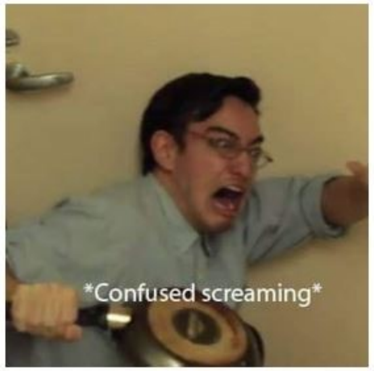
Confused screaming

Here, the played sound is the **first conditioned stimulus (CS1)**.
We pair the sound with an electrical shock, which is referred to as the **unconditioned stimulus (US)**.
The reaction our subject has to the shock is often referred to as the **unconditioned response (UR)**.

Basic classical conditioning

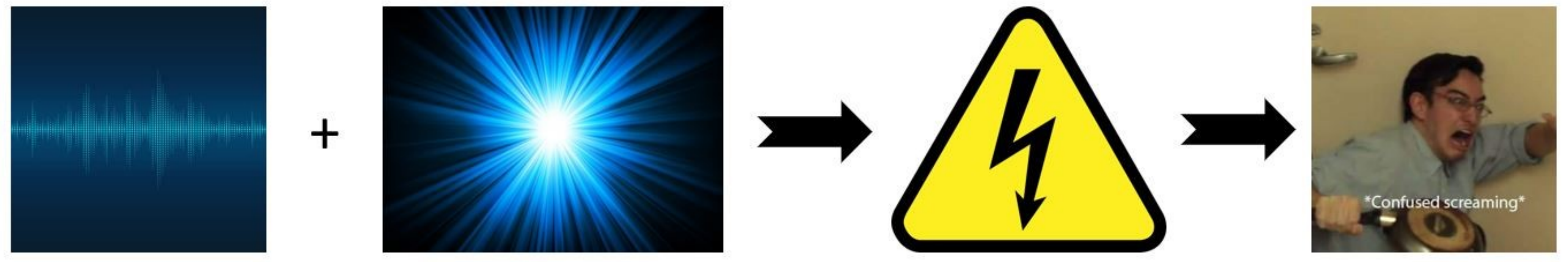
After pairing CS1 and US multiple times, the UR (confused screaming) will become the **conditioned response (CR)**.
Thus, the sound will elicit the screaming even though no shock was administered.





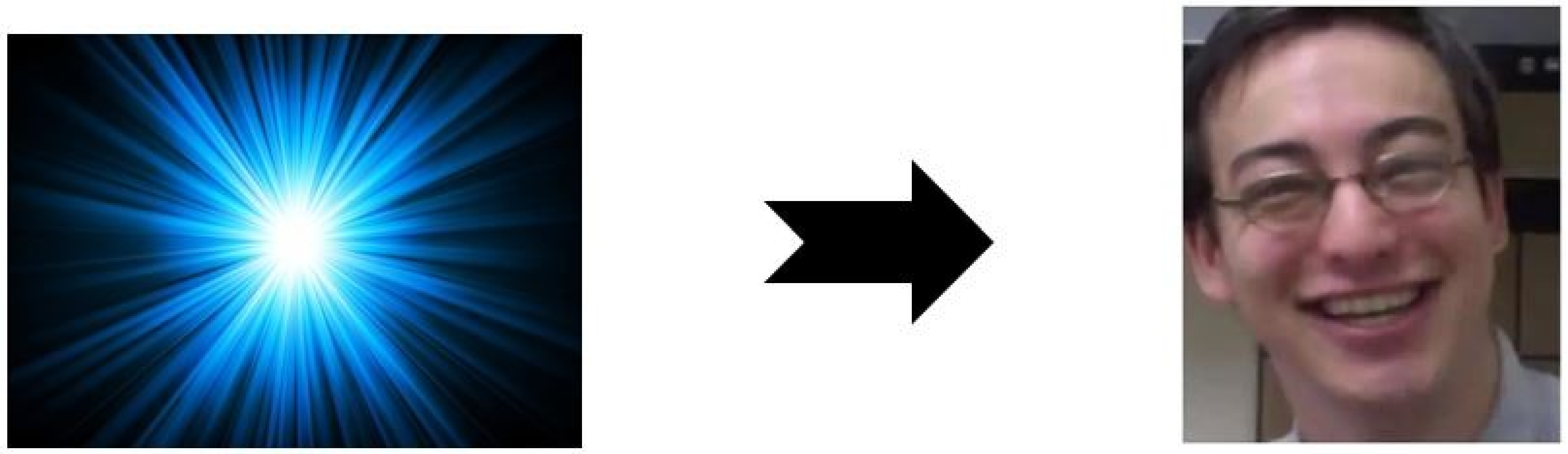
The blocking effect

Now we do extra conditiong, but we show CS1 together with a **second conditioned stimulus (CS2)** (e.g. a strong light).
After conditioning, CS1 + CS2 will also lead to confused screaming:



The blocking effect

Interestingly, when we show CS2 alone, this will not lead to the CR (Kamin, 1967) (https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19680014821.pdf).



It appears that a subject is not able to learn that the light also predicts the shock.
In other words: the learning of the CS2 - US association is *blocked* because the CS1 - US assocation already exists...

Modelling the blocking effect

Now, we ask you to prove the blocking effect using a model.
You will have to do this yourself, relying on the code provided for exercise 1.
The questions asked below might help you out.

- How many units does your model need?
    - Input layer
        - The model can encounter two different stimuli: **only sound** and **sound + light**
        - Sound and light can be seen as two different units: if the unit is switched of, the stimulus is not available
    - Output layer
        - Only two outputs: aversion or no aversion --> this is doable with one unit

Modelling the blocking effect

- Make a weight matrix to start with
- Use delta learning to learn the **CS1 - US association**
- Use delta learning to learn the **CS2 - US association**
    - Importantly, make sure that you use the weight matrix obtained from the previous step as a starting point
- Does the end result prove blocking? Why / why not?
    - How can you check investigate whether the blocking occured or not?

3. Delta learning: DIY with the iris dataset

Iris dataset?

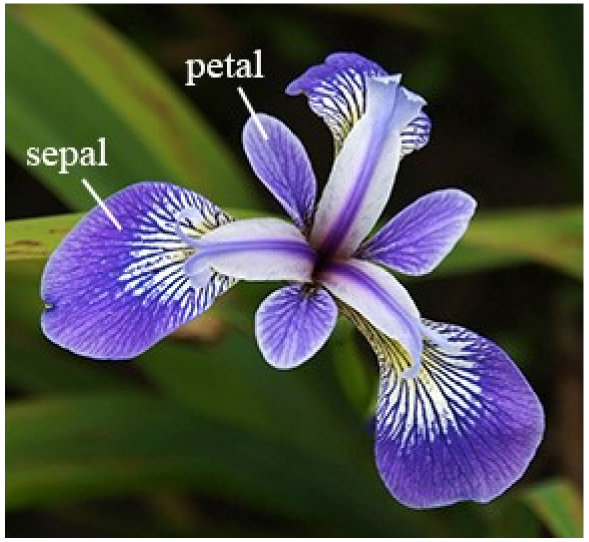
In this exercise, we will use the iris dataset (Fisher, 1936) (https://onlinelibrary.wiley.com/doi/epdf/10.1111/j.1469-1809.1936.tb02137.x).
More specifically, we will use this dataset to **predict the species** of the flower **based on the features** of the flower.

The dataset consists of 150 rows, where each row represents measurements of 150 different flowers.
Each flower is different, but they all belong to the same family: "iris".
There are 3 different species in the dataset, so we have 50 different flowers for each family.

The data available 🌹

- Features of the flower
  - Sepal width
  - Sepal length
  - Petal width
  - Petal length
- The name of the flower
  - Iris setosa
  - Iris virginica
  - Iris vericolor

An example of the provided features



Our question
What iris *type (setosa*, *virginica or versicolor?)* is this based on the provided measures?

In [6]:
```
# show me the way (first 10 rows)
print('First 5 observations:\n', iris_visual[:5])
print('\nLast 5 observations:\n',iris_visual[-5:])
```

```
First 5 observations:
    sep len  sep wid  pet len  pet wid  family
0      5.1      3.5      1.4      0.2     0.0
1      4.9      3.0      1.4      0.2     0.0
2      4.7      3.2      1.3      0.2     0.0
3      4.6      3.1      1.5      0.2     0.0
4      5.0      3.6      1.4      0.2     0.0

Last 5 observations:
      sep len  sep wid  pet len  pet wid  family
145      6.7      3.0      5.2      2.3     2.0
146      6.3      2.5      5.0      1.9     2.0
147      6.5      3.0      5.2      2.0     2.0
148      6.2      3.4      5.4      2.3     2.0
149      5.9      3.0      5.1      1.8     2.0
```

...?

Our goal is to predict the family based on the provided features.
So, if we see the following:

```
In [9]: X[10,:]
Out[9]: array([5.4, 3.7, 1.5, 0.2])

In [10]:y[10]
Out[10]: 0
```

We know that flower 11 has a sepal length of 5.4 cm, sepal width of 3.7 cm ... .
We also know that flower 11 belongs to family 0 (i.e. setosa).

Ideally, our model would be able to predict the family based on the features for every flower.
So, if we give the model the features for flower 62:

```
In [16]: X[61,:]
Out[16]: array([5.9, 3. , 4.2, 1.5])
```

we want to output of the model to be equal to 1 (i.e. versicolor), which is the observed family for flower 62.

Our action plan:

1. Open **'ch4_iris_delta_exercise.py'**
2. Use delta learning to let your model respond with the correct flower family based on the features
3. Take a look at the Python module `scikit-learn`, which allows us to implement a single layered model
    - i.e. there are no layers between the input- and the output layer
4. Split the data in a training set and a testing set, use the training set to optimize the model, use the test set to check the performance of your model
    - Google *split data train test scikit learn*, and see which Python functions might help you out 🌐
5. Use the `Perceptron` function to make a single layered model, and train this model on the training set
    - Stuck? Consider using `help(Perceptron)` to read the docs
6. Finally, check whether your trained model is able to predict the family of new observations (your test set serves as "new observations")

In [7]:
```python
# import: general and scikit-learn specific
import numpy                    as np

from sklearn                    import datasets
from sklearn.linear_model       import Perceptron
from sklearn.metrics            import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing      import StandardScaler
```

In [8]:
```
# import the Iris flower dataset
iris        = datasets.load_iris()
X           = iris.data
y           = iris.target
class_names = iris.target_names

# split data in training and testing set
X_train, X_test, y_train, y_test = train_test_split(X,
                                                     y,
                                                     random_state = 20)
```

In [9]:
```
# train the standard scaler with a part of the data
sc = StandardScaler()
sc.fit(X_train)

# apply scaler to all x
X_train_std = sc.transform(X_train)
X_test_std  = sc.transform(X_test)
```

In [10]:
```
# define classifier (Perceptron object from scikit-learn)
classification_algorithm = Perceptron(max_iter         = 100,
                                      tol              = 1e-3,
                                      verbose          = 0,
                                      random_state     = 20,
                                      n_iter_no_change = 5)

# fit ('train') classifier to the training data
classification_algorithm.fit(X_train_std, y_train)

# predict y based on x for the test data
y_pred = classification_algorithm.predict(X_test_std)
```

```
In [11]: # select wrong predictions (absolute vals) and print them
         compared       = np.array(y_pred == y_test)
         absolute_wrong = (compared == False).sum()
         print("Our classification was wrong for {0} out of the {1} cases.".format(absolute_
         wrong,
                                                                                  len(compa
         red)))


         # print accuracy using dedicated function
         print('Accuracy percentage: {0:.2f}'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Our classification was wrong for 2 out of the 38 cases.
Accuracy percentage: 94.74
```