Fundamentals of Artificial Intelligence (57205HT16): Assignment 2 - Happy, Sad, Mischievous or Mad?

Marc Coquand
id14mcd
mcoquand@gmail.com

Linus Lagerhjelm
id14llm
id14llm@cs.umu.se

Supervisor: Thomas Johansson
Alexander Sutherland
Thomas Hellström

October 10, 2016

Contents

1 Introduction

This report contains the documentation for a perceptron based classification system that guesses the emotional state of faces presented as input. The inputs are 20 x 20 pixel maps with 32 grey levels from white to black.

1.1 Running the program

The implementation is written in python 2.7 which means that a python 2.7 shell is required to run the implementation. Required program arguments are a search path to a `training.txt`, search path to an answer file `training-facit.txt` and a test file `test-file.txt`. The format of the training file should be the id of the image followed by the image on a new line. The answer file should be formatted as the image id follow by the correct answer (1,2,3 or 4).

Should the user fail to provide the required arguments, the program will immediately terminate with a massage explaining that not enough input arguments were provided.

Example usage of the program might look like:

```
python faces.py ../data/FaceTest/training-file.txt
../data/FaceTest/training-facit.txt test-file.txt
```

2 Problem description

The program works as such that it starts of by reading the *images*. An image consists of it's pixels (value from 0 - white to 31 - black) and it's ID (sad, mischievous, happy, mad). The program then filters the noise and then sends the images in the *Tutor* class. The tutor uses Perceptrons to learn what images look like (algorithm explained below). A perceptron is an algorithm for learning a binary classifier. Afterwards the Examiner class makes a guess and predicts which image belongs to which on a new set of image. The program will then output the percentage of the guesses that were correct.

2.1 Perceptron Learning Algorithm

The program implements an algorithm for supervised learning called *Perceptron Learning*. *Supervised Learning* is a technuque for machine learning where the Artificial neural network is exposed to input and produces a guess based on the knowledge aquired from previous guesses. Since the desired output from the ANN, in supervised learning, is known for every input, the ANN is slightly modified with each iteration to eventually produce good outputs for every set of inputs.

The ANN in the produced program is composed by four different Perceptrons where each one represents one of the different facial expressions that are to be identified in the assignment.

Each perceptron is implemented in *Python* as a class composed by a list of *401* weights that mapps to the *400* pixel values that makes up an image. The additional weight is provided as a bias for the perceptron. A bias is used in order to shift the result of the activation function (more on that later) towards either the left or the right. I.e. to make the output more extreme so that it will produce a more distinct answer.

Training in the network occurs in the *Tutor* class where the provided set of training images is repetedly shown for each perceptron. The output from a perceptron, also known as *activation*, is computed using the formula:

$$a_i = act(\sum_j x_j * w_{j,i})$$

where x is the input, w is the weight and act is the activation function, which is the sigmoid function:

$$sigmoid(x) = 1/(1 + e^{-x})$$

The difference between the expected output and the actual output is the error e. The error e is then multiplied with the learning rate α and the pixel matrix to produce the gradient ∇w. This gradient is then added to every weight in the internal weight matrix for each perceptron. This course of events can be described using the bellow formulas:

$$
\begin{aligned}
&e_i = y_i - a_i \\
&\nabla w_{j,i} = \alpha e_i x_j \\
&w_{j,i} \leftarrow w_{j,i} + \nabla w_{j,i}
\end{aligned}
$$

This process is repeated for the entire set of training images, however, only performing this process once would not produce very good results and therefore, when the network has used all the training data, the images are shuffled and the entire process described above is repeated.

The main problem encountered while training an ANN is finding the fine line right before the network gets overfitted. When the ANN gets overfitted, it becomes too specialized on the training data which will render it useless when exposed to previously unseen data. It is, however, still crucial to train the ANN as close to this limit as possible in order in order for it to be able to make generalizations to the data and produce decent output.

One way of determine when the network has reached the desired level of training is, for each iteration of the training data set, use a formula to compute the sum of the output errors for the network and end training when the value from this function gets below a predefined level. The function used in the current implementation is:

$$E = \frac{1}{2} \sum_{i=1}^{p} ||y^{(i)} - d^{(i)}||^2$$

Where p is the number of input/output vector pairs in the training set.

When the threshold for the error is reached, the perceptrons is deemed ready to passed on to the *Examiner* class where the network is exposed to the set of test images. Each image is shown to every perceptron in the network and the activation for each perceptron is recorded. The largest activation is chosen as the answer for the network and is recorded into our list of answers. When every image in the test set has been classified the answers are printed to *stdout*.

3 Concluding discussion

The biggest difficulty was choosing which programming language to use. It was difficult choosing between Haskell and Python. Ultimately we chose Python as our knowledge in Haskell was not sufficient to write this program. It would be interesting to implement more AI and more specifically in machine learning programs in a functional programming language as it's easier to deal with concurrency. If you wanted to scale this program and have a larger neural network it might be necessary to add concurrency.

```python
import sys
import re
from Enum import Enum
from Perceptron import Perceptron
from Tutor import Tutor
from Utils import flatten
from Examiner import Examiner
from Image import Image

# Program arguments: training-set, answer-set, examination-set (from spec)
EXPECTED_ARGS = 3
Types = Enum(["HAPPY", "SAD", "MISCHIEVOUS", "MAD"])

# Optimal values (derived from experiments):
# learning_rate = 0.01
# threshold = 1
LEARNING_RATE = 0.1
THRESHOLD = 1


def parse_ans(ans_file):
    """
    Opens the provided answer file and extracts the answers which are
    returned as a list of integers. NON PURE FUNCTION!
    :param ans_file: path to answer file
    :return: list of integers
    """
    l2 = filter(lambda l: len(l) > 0, map(get_answer, open(ans_file, 'r')))
    return flatten(l2)


def parse_img_file(image_file):
    """
    Opens the provided image file and extracts its content into a
    list of image objects that gets returned. NON PURE FUNCTION!
    :return: list of image objects
```

```python
    """
    f = open(image_file, 'r')
    return reduce_img_file(f.readlines(), [])


def reduce_img_file(lines, images):
    """
    Consumes the lines of the image file in order to extract all
    the images and their ids. In the end it will return a list
    of image objects.
    :return: list om image objects
    """
    if not lines: return images
    if re.search('Image', lines[0]) is not None:
        img = Image(extract_img_data(20, lines[1:]))
        return reduce_img_file(lines[20:], images + [img.set_id(
    lines[0])])

    return reduce_img_file(lines[1:], images)


def extract_img_data(acc, lines):
    """
    Sub consumes all the lines that contains an image row and
    returns a 2D array containing list of lists of integers
    :param acc: int representing how many more lines to read
    :param lines: the lines of the file
    :return: 2D array of integers
    """
    if acc == 1: return [format_img_row(lines[0])[0]]
    return [format_img_row(lines[acc-1])[0]] + extract_img_data(
    acc - 1, lines)


def format_img_row(row):
    """
    format_img_row receives a row from the image file
    representing a pixel
    row in an image and returns it as a list of floats in [0,1)
    :param row: image pixel row
    :return: list of integers
    """
    return map(lambda l: map(lambda i: (float(i)/31), l), map(
    lambda i: i.rstrip().split(' '), [row]))

```

```
def get_answer(line):
    """
    get_answer retrieves the value for the answer from a line in the
    answer file
    :param line: line from answer file
    :return: Integer representing the answer for the line
    """
    return map(lambda s: int(s), re.findall(r'\b\d+\b', line))


def validate_arguments():
    """
    makes sure that the user provided the right amount of arguments to the
    program, else: fail gracefully. NON PURE FUNCTION!
    :return:
    """
    # Since program name is provided as additional first argument
    if len(sys.argv) < EXPECTED_ARGS + 1:
        print "This program expects exactly %d args. %d was provided" % \
              (EXPECTED_ARGS, len(sys.argv) - 1)
        sys.exit(0)


def print_results(res_arr):
    """
    Expects a list with formatted answer strings that gets printed to stdout
    """
    if not res_arr: return
    print res_arr[0]
    print_results(res_arr[1:])


def main():
    validate_arguments()
    perceptrons = (Perceptron(Types.HAPPY), Perceptron(Types.SAD),
                   Perceptron(Types.MISCHIEVOUS), Perceptron(Types.MAD))

```

```
    # Get content of the data files
    img_list = parse_img_file(sys.argv[1])
    ans_list = parse_ans(sys.argv[2])
    test_set = parse_img_file(sys.argv[3])

    # Link image and answer
    images = map(lambda ans: img_list.pop(0).set_ans(ans),
   ans_list)

    trained_p = Tutor(perceptrons, images, LEARNING_RATE,
  THRESHOLD).train()
    print_results(Examiner(trained_p, test_set).examine())

if __name__ == "__main__":
    main()
```

../src/faces.py

```
class Enum(set):
    """ Enum is a python 2.7 implementation of a Enum
        provided by http://stackoverflow.com/a/2182437/4689625
    """
    def __getattr__(self, name):
        if name in self:
            return name
        raise AttributeError
```

../src/Enum.py

```
class Examiner:
    """
    Examiner is a class that, provided a set of trained neurons
   and
    test images, examines the training results of the network
    Attributes:
        perceptrons: tuple of trained perceptrons
```

```
        test_set: set of test images

    """

    def __init__(self, perceptrons, test_set):
        self.perceptrons = perceptrons
        self.test_set = test_set

    def examine(self):
        """
        Tests the provided perceptrons on the provided set of
    images
        :return: A list of formatted answer strings
        """
        return map(self._make_guess, self.test_set)

    def _make_guess(self, image):
        """
        Exposes the provided image to the network and returns a
    string
        that is compliant with the specification of what a guess
     should
        look like
        :return:
        """
        guess = self._expose_perceptrons(image, {"type": 0, "
    activation": 0}, self.perceptrons)
        return self._format_answer(image, guess["type"])

    def _expose_perceptrons(self, image, current_guess,
    perceptrons):
        """
        Shows an image to all of the perceptrons and returns the
     largest
        activation from the network
        :param image: image to show
        :param current_guess: The previous largest activation in
     the network
        :param perceptrons: consumed list of perceptrons
        :return:
        """
        if not perceptrons: return current_guess
        activation = perceptrons[0].process(image)
        if activation > current_guess["activation"]:
            current_guess = self._update_guess(activation,
```

```
perceptrons[0])

        return self._expose_perceptrons(image, current_guess,
perceptrons[1:])

    def _update_guess(self, activation, perceptron):
        """
        Returns a new dictionary containing the activation for
the
        perceptron type
        """
        return {"type": perceptron.get_type_id(), "activation":
activation}

    def _format_answer(self, img, guess):
        """
        Returns properly formatted string based on the networks
guess of
        the provided image
        """
        return "" + img.get_id() + " " + str(guess)
```

../src/Examiner.py

```
import Utils
FILTER_VAL = float(3)/float(32)  # Derived from experiments


class Image:
    """
    Image represents an image that holds a list of pixels, id,
and what
    facial expression the image represents.
    Attributes:
        img: list of pixel values
        id: the id of the image on format ImageXX
        ans: integer 1-4 representing what facial expression the
    img represents
    """
    def __init__(self, img):
        self.img = self._preprocess_img(img)
```

```
        self.id = None
        self.ans = None

    def _preprocess_img(self, img):
        """
        Performs preprocessing operations in order to sanitize
        the input data so that it contains what we expects.
        """
        i1 = Utils.flatten(img)
        i1.append(1)
        return map(self._filter_noise, i1)

    def _filter_noise(self, pix):
        """
        Since the images contains a lot of noise, we're filtering
        away those pixels that wont't impact the result other than
        to make it unambiguous.
        """
        return pix if pix > FILTER_VAL else float(0)

    def get_img(self):
        """
        Returns the pixel data for the image as a list of integers
        """
        return self.img

    def set_ans(self, ans):
        """
        Updates the answer for the image. Should be a single integer
        between 1-4
        """
        self.ans = ans
        return self

    def get_ans(self):
        """
        Returns the answer as an integer between 1-4
        """
        return self.ans

    def set_id(self, id):
```

```
        """
        Updates the id of the image
        """
        self.id = str(id).rstrip()
        return self

    def get_id(self):
        """
        Returns the id of the image
        """
        return self.id
```

../src/Image.py

```
import random
import math
from Utils import *


class Perceptron:
    """
    Perceptron represents a perceptron in the network. A
   perceptron
    represent one of the four feelings specified in the TYPE
   field.
    It reacts to an input image with a response based on it's
   training.
    Attributes:
        TYPES: mapping of the perceptron type and type id
        weights: a list of weights that maps to image pixels
        type_id: the type id of the perceptron
    """

    def __init__(self, percept_type):
        self.TYPES = {"HAPPY": 1, "SAD": 2, "MISCHIEVOUS": 3, "
  MAD": 4}
        self.weights = self._generate_weights(401)
        self.type_id = self.TYPES[percept_type]

    def process(self, img):
        """
```

```
        Uses the formula sum(w[i], x[i]) from lecture notes to
    process an image
        :return: summed weight of inputs
        """
        l1 = apply(float.__mul__, img.get_img(), self.weights)
        return self._act(sum(l1))

    def _act(self, val):
        """
        Computes the activation for the perceptron based on the
    sum of the
        weighted input
        :param val: summed weight in input
        :return: activation of the perceptron
        """
        return 1 / (1 + math.exp(-val))

    def update_weight(self, dw):
        """
        adds the provided delta error for the last iteration to
    each
        value in the internal weight matrix according to lecture
     notes
        :return: None
        """
        self.weights = apply(float.__add__, flatten(dw), self.
    weights)

    def get_type_id(self):
        """
        Returns the type id for this perceptron. Consult the
    type
        type attribute for explanation of the meaning of the
    returned
        value.
        :return: type id as an integer [1, 4]
        """
        return self.type_id

    def _generate_weights(self, acc):
        """
        Returns a list of the specified length containing pseudo
        random values in interval [0.0, 1.0)
        :param acc: length of the list to generate
        :return: list of pseudo random floats
```

```
        """
        return [random.random()] if acc == 1 else self.
    _generate_weights(acc-1) + [random.random()]
```

../src/Perceptron.py

```
import random
from Utils import mmult, flatten


class Tutor:
    """
    Tutor represents a supervisor object that is responsible for
     training a
    set of perceptrons using a provided set of training images
    until a
    certain level of correctness is reached.
    Attributes:
        perceptrons: a tuple of perceptrons to train
        images: a list of image objects to train on
        learning_rate: value specifying how quickly the network
    should learn
        threshold: the value that indicates when to stop
    training
    """

    def __init__(self, perceptrons, images, learning_rate,
    threshold):
        self.perceptrons = perceptrons
        self.images = images
        self.learning_rate = learning_rate
        self.threshold = threshold

    def train(self):
        """
        Trains the perceptrons provided to this object with the
    provided
        using the provided training images until the desired
    threshold
        for the sum squared error of the network is reached. The
     trained
```

```
        perceptrons are returned as a tuple.
        """
        self._do_training([])
        return self.perceptrons

    def _do_training(self, errors):
        """
        This function is what actually does what's described in
the train
        method. This separation is done so that the user of this
 method is
        not bothered by the fact that the method is implemented
recursively.
        """
        if self._accurate(errors): return self
        random.shuffle(self.images)
        self._do_training(flatten(map(self._each_perceptron,
self.images)))

    def _each_perceptron(self, image):
        """
        Shows the provided image to all the perceptrons and
returns a list
        containing the error for every perceptron.
        """
        return map(lambda p: self._expose(image, p), self.
perceptrons)

    def get_perceptrons(self):
        """
        get_perceptrons returns a tuple of perceptrons. This
method returns
        the perceptrons in their current state. The train()
method should be
        called until desired MSE is achieved before a call to
this method
        makes sense.
        :return: perceptrons as a tuple
        """
        return self.perceptrons

    def _expose(self, image, perceptron):
        """
        Shows the provided image to the specific perceptron and
uses the
```

```
        formula for perceptron learning from lecture notes to update the
        internal weights in the perceptron based on the delta error
        for the perceptron on the image.
        :return:
        """
        err = self._error(self._desired(image, perceptron), perceptron.process(image))
        dw = self._delta_weight(image, err)
        perceptron.update_weight(dw)
        return err

    def _delta_weight(self, inp, err):
        """
        Uses the formula (error * learning rate * input) from lecture notes
        to produce the delta error for the activation of the perceptron
        which is returned.
        :param inp: the perceptron input (image)
        :param out: desired output, either 0 or 1
        :param act: the activation of the perceptron
        :return:
        """
        return mmult(inp.get_img(), self.learning_rate * err)

    def _error(self, out, act):
        """
        Computes the error between the desired output and the actual activation
        """
        return out - act

    def _accurate(self, err_list):
        """
        Determines the size of the error using the formula for computing
        the error of a network provided over at:
        http://lcn.epfl.ch/tutorial/english/perceptron/html/learning.html
        Uses the user provided threshold value to determine whether the
        network performs well enough.
        """
```

```python
        if not err_list: return False
        mse = (sum(map(lambda x: x**2, err_list)))/2
        return mse < self.threshold

    def _desired(self, img, percept):
        """
        Compares the provided image and perceptron to determine whether
        the neuron should be firing on the image. The decision is
        represented as either 0 or 1.
        :return: 1 or 0 if the neuron should fire
        """
        return int(img.get_ans() == percept.get_type_id())
```

../src/Tutor.py

```python


def mmult(m, c):
    """
    Receives a matrix and returns a new matrix
    where every element in the matrix is multiplied
    by the provided constant
    :param m: matrix
    :param c: constant
    :return: new matrix with the applied constant
    """
    return map(lambda x: x * c, m)


def msum(m):
    """
    Returns the sum of every element in the matrix
    """
    return sum(map(sum, m))


def apply(op, l1, l2):
    """
    Returns a list of the same length as l1 and l2 where each
```

```python
    index is the result of the provided operation on l1[i] & l2[i]
    NOTE: Shadows built in function apply but since it is deprecated
    since version 2.3 we don't care.
    Reference: https://docs.python.org/2/library/functions.html#apply
    """
    return map(lambda tup: op(float(tup[0]), float(tup[1])), zip(l1, l2))


def flatten(l):
    """
    Receives a nested list and returns all the values in a single list
        Example: [[1], [2, 3], 4] -> [1, 2, 3, 4]
    :return: flattened list
    """
    if not l: return l
    return flatten(l[0]) + flatten(l[1:]) if isinstance(l[0], list) else l[:1] + flatten(l[1:])
```

../src/Utils.py