The Duck Test vs Deep Learning

M. Andrecut

December 9, 2019

Calgary, Alberta, T3G 5Y8, Canada
mircea.andrecut@gmail.com

Abstract

The well known duck test is usually formulated as following: "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." By analogy with the duck test we formulate a simple algorithm for image classification. We show numerically that this simple algorithm outperforms more elaborated deep learning methods and their expensive software implementations. Moreover, we show that the duck test algorithm has several advantages over deep learning in general: (1) it is extremely robust; (2) it has no parameters, and therefore it cannot overfit the data; (3) it requires less training data; (4) it can be generally applied to any recognition or classification problem; (5) it is very simple to implement.

Keywords: probability and statistics, artificial intelligence, deep learning, duck test
PACS: 02.50.-r, 07.05.Mh

1 Introduction

Deep learning is the latest addition to machine learning, and implicitly to artificial intelligence [1]. The success of deep learning in data science competitions has reinvigorated the stagnant field of artificial intelligence, giving reason for new exciting developments in: image classification, speech recognition, intelligent assistants, machine translation, healthcare, finance forecasting, self-driving cars and drones, classified military applications etc. Despite of these undeniable success stories, the deep learning approach to artificial intelligence has been recently critically scrutinized, and several significant shortcomings have been pointed out, raising serious questions and doubts about the deep learning approach to artificial intelligence [2].

Here, we consider a completely different approach, which we hope it will sort out some of these controversies, or it will add few more. Our approach is based on an analogy with the well known "duck test": "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." In particular, we consider the problem of image classification, and we show that such a simple comparison with a set of features outperforms a more elaborated convolutional neural network, which is representative for deep learning.

The theoretical basis of the duck test is "abductive reasoning", which is a form of logical inference where a set of (incomplete) observations are used to find the simplest and most likely explanation (also known as the "the logic of Sherlock Holmes") [3]. We illustrate this approach with a simple algorithmic implementation applied to the MNIST data set, which is a well known benchmark frequently used in machine learning for image classification [4]. Surprisingly (or not), this very simple approach outperforms the convolutional neural network implementation provided by Tensorflow, which currently is the state of the art software library developed by Google for deep learning [5]. Moreover, with this example we show that the duck test approach has several significant advantages over deep learning: (1) it is extremely robust; (2) it has no parameters, and therefore it cannot overfit the data; (3) it requires less training data; (4) it can be generally applied to any recognition or classification problem; (5) it is very simple to implement.

2 The duck test

Here we formulate a very simple algorithmic implementation of the duck test, applied to the problem of image recognition. In order to illustrate our approach we consider the well known MNIST data set, which is a large database of handwritten digits (0,1,...,9), containing 60,000 training images and 10,000 testing images [4]. These are monochrome images with an intensity in the interval $[0, 255]$ and the size of 28×28 pixels. The MNIST data set is probably the most frequently used benchmark in image recognition.

The duck test consists in a set of comparisons of the unclassified object (image in this case), or parts of the object, with a set of features corresponding to different classes. Therefore, in order to formulate the duck test algorithm we first want to extract a set of features, which are characteristic for each class of digits $k = 0, ..., 9$. In order to do so, we iterate over all images from a class k, and we simply extract all the overlapping patches (sub-images) of a fixed size $\ell \times \ell$, where $\ell < L = 28$. Let us assume that x_n is an image from the class $c(x_n) = k$, then we extract all the patches $\xi_{nij} \subset x_n$, of size $\ell \times \ell$, where $i, j \in \{0, ..., L - \ell\}$:

$$x_{nij} = [x_{ni'j'}], \quad i' = i, ..., i + l - 1, j' = j, ..., j + l - 1. \tag{1}$$

We also normalize the patches as following:

$$\xi_{nij} \leftarrow \xi_{nij} - \langle \xi_{nij} \rangle, \tag{2}$$

$$\xi_{nij} \leftarrow \xi_{nij} / \|\xi_{nij}\|, \tag{3}$$

where $\langle , \rangle$ is the average, and $\| \cdot \|$ is the Euclidean norm of the vector ξ_{nij} obtained by concatenating the columns of the patch. Thus, from each image we extract $(L - \ell + 1) \times (L - \ell + 1)$ patches. These patches are then used to define the "most common set of features" corresponding to this class. This is done by employing a very "primitive" form of the well known k-means algorithm. For each class we consider Q centroids μ_q, $q = 0, ..., Q - 1$, randomly initialized and normalized such that: $\|\mu_q\| = 1$. For every patch ξ_{nij} we search the closest centroid μ_{q^*} satisfying:

$$q^* = \arg\max_q \xi_{nij}^T \mu_q. \tag{4}$$

After every patch was assigned to a centroid we update the centroids as following:

$$\mu_q \leftarrow \sum_n \sum_i \sum_j \xi_{nij} f(\xi_{nij}, \mu_q), \tag{5}$$

$$\mu_q \leftarrow \mu_q / \|\mu_q\|, \tag{6}$$

where $f(\xi_{nij}, \mu_q) = 1$ if x_{nij} was assigned to μ_q, and $f(\xi_{nij}, \mu_q) = 0$ otherwise. Because this is a relatively costly procedure, we only apply three iteration steps, so we do not seek full convergence of the k-means algorithm. In fact, our numerical results have shown that the convergence of k-means is not important, and it doesn't influence the results. What seems to be important is the "weak" mixing of similar patches into a centroid (common feature), which can be achieved using a simple three steps iteration. We repeat this "weak learning" process for each class, such that in total we have maximum $(K+1)Q$ centroids ($K = 9$ in this case), each of them corresponding to a different class: $c(\mu_q) = k$ if μ_q was "extracted" from the class k. It is important to note that not all the centroids will be selected and updated during this weak learning process (some of them may not be selected by any patches, and cannot form a cluster), and therefore the final number of centroids will be $Q' \leq (K+1)Q$. Anyway, what is important is that we have extracted a "weak" set of common features (the centroids), and each of these features corresponds to a certain class.

Once the features (observations) have been extracted we can use them for classification. So, let us assume that y_m is a test image. We extract all the patches $\gamma_{mij} \subset y_m$ of size $\ell \times \ell$, and we normalize them:

$$\gamma_{mij} \leftarrow \gamma_{mij} - \langle \gamma_{mij} \rangle, \tag{7}$$

$$\gamma_{mij} \leftarrow \gamma_{mij} / \|\gamma_{mij}\|. \tag{8}$$

Then for each patch γ_{mij} we find the most "similar" centroid (feature), and the corresponding class of that centroid:

$$q^* = \arg\max_q \gamma_{mij}^T \mu_q, \tag{9}$$

$$k^* = c(\mu_{q^*}). \tag{10}$$

In the end each patch from the image y_m "votes" with the corresponding class of the selected centroid, and the class with the most votes is attributed to the test image y_m. This is in fact analogous to applying the duck test. So, if the majority of features of an images are coming from a certain class, then the image will be associated with that class.

3 Numerical results

The Python implementation of the duck test algorithm is given in the Appendix. We have applied the duck test to the MNIST data set, and we have assumed the following parameters: $Q = 3,000$ and $\ell = 18$. To simplify the computation we only used patches extracted from the sub-image $[2, L-2] \times [2, L-2]$, since the 2 pixels border of the images don't seem to contain information. On a very basic computer with 4 CPU cores running at 3.2GHz, the whole process takes around 20 minutes. The algorithm extracted $Q' = 22,796$ features (centroids), providing a recognition accuracy of 98.39%.

In contrast, the architecture of the Tensorflow convolutional neural network is quite complicated, and it has several layers [6]:

- Convolutional Layer 1: Applies 32 5x5 filters (extracting 5x5-pixel subregions), with ReLU activation function;
- Pooling Layer 1: Performs max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap);
- Convolutional Layer 2: Applies 64 5x5 filters, with ReLU activation function;
- Pooling Layer 2: Again, performs max pooling with a 2x2 filter and stride of 2;
- Dense Layer 1: 1,024 neurons, with dropout regularization rate of 0.4 (probability of 0.4 that any given element will be dropped during training);
- Dense Layer 2 (Logits Layer): 10 neurons, one for each digit target class (0-9).

On the same basic computer it took around 158 minutes for training the network using the mini-batch implementation (100 samples at each step) of the stochastic gradient descent algorithm, with 20,000 steps. However, despite of these architectural complications and the elaborated training algorithm, the results were worse than for the simple and fast duck test algorithm, with a recognition accuracy of only 97.32%.

The above results have been obtained using the "standard" setting with 60,000 images for training, and respectively 10,000 images for testing. In a second experiment we have decided to see what is happening if we revert the situation, such that the 10,000 test images are used for training and the 60,000 training images are used for testing. In this case the simple duck test algorithm outperforms the Tensorflow convolutional neural network implementation by a much larger margin. The recognition accuracy of the duck test was 97.33% (with $Q' = 15,778$ features extracted, initially allowed $Q = 3,000$ per class), while the recognition accuracy of the convolutional neural network significantly dropped to 95.86%. In the third experiment we used for training only the first 1,000 images from the original test set, such that the rest of 69,000 images are used for testing. In this case the duck test still achieved 93.26% recognition accuracy (with only $Q' = 3,044$ features extracted, initially allowed $Q = 500$ per class), while the Tensorflow convolutional neural network implementation is left way behind with an accuracy of only 90.87%.

Another interesting property of the duck test is that we can drop a large fraction of the features (forget them, or delete them) and we still get a good recognition accuracy, while in the case of deep learning if we alter only a little bit the network weights we are guaranteed to get very bad results.

We should note that several other authors have reported higher recognition rates using convolutional neural networks and capsule networks (see [7] and [8] for a summary). These results have been reported with very complicated ensembles of convolutional neural networks, and by significantly expanding the training data set with (elastically) distorted versions of the images from the training data set. Of course that the duck test can be also applied to an augmented data set or it can be integrated into an ensemble of duck tests, and we may expect that the recognition accuracy will increase, however here we limit our investigation to the non-distorted, non-augmented and the single network case, because we want to have a one-on-one comparison between a single duck test and a single convolutional neural network.

We conclude that the duck test algorithm is extremely robust and requires less training data than the convolutional neural network method, also the duck test cannot be suspected of overfitting since the model doesn't have parameters analogue to the neural network weights, and it is not based on an optimization method.

Conclusion

In this paper we have formulated a very simple algorithmic implementation of the well known duck test: "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck", and we have shown numerically that this simple algorithm outperforms more elaborated deep learning methods, and their expensive software implementations. Moreover, we have shown that the duck test has several advantages over deep learning in general: (1) it is extremely robust; (2) it has no parameters, and therefore it cannot overfit the data; (3) it requires less training data; (4) it can be generally applied to any recognition or classification problem; (5) it is very simple to implement.

Appendix

Here we provide the duck_test.py, which is a Python3 implementation (requires Numpy) of the duck test algorithm applied to MNIST data downloaded from http://yann.lecun.com/exdb/mnist/.

```
import numpy as np

def read_data(imagefile, labelfile, N, M):
    x = np.zeros((N,M), dtype='uint8')
    images = open(imagefile, 'rb')
    images.read(16)  # skip the magic_number
    for n in range(N):
        x[n,:] = np.fromstring(images.read(M), dtype='uint8')
    images.close()
    labels = open(labelfile, 'rb')
    labels.read(8)  # skip the magic_number
    xl = np.fromstring(labels.read(N), dtype='uint8')
    labels.close()
    print(imagefile,", ", labelfile)
    return (x, xl)

def get_one_image(x, L, l):
    z = x.reshape((L,L))
    y = np.zeros(((L-l-4)*(L-l-4), l*l))
    n = 0
    for i in range(2,L-l-2):
        for j in range(2,L-l-2):
            y[n,:] = z[i:i+l,j:j+l].flatten()
            y[n,:] = y[n,:] - np.mean(y[n,:])
            y[n,:] = y[n,:]/np.linalg.norm(y[n,:])
            n = n + 1
    return y

def features(x, xl, K, L, l, Q, tm):
    (N, M), LL, ll = x.shape, (L-l-4)*(L-l-4), l*l
    u = np.random.rand(K*Q, l*l) # centroids
    ul = np.zeros(K*Q, dtype='uint8') # centroids labels
    t = 0
    for k in range(K):
        for q in range(Q):
            u[t,:] = u[t,:] - np.mean(u[t,:])
            u[t,:] = u[t,:]/np.linalg.norm(u[t,:])
            ul[t] = k
            t = t + 1
    s = np.zeros(K*Q, dtype='int')
    for k in range(K):
        kQ, kkQ = k*Q, (k+1)*Q
        for t in range(tm):
            v, s[kQ:kkQ] = np.zeros((Q, ll)), np.zeros(Q)
            for n in range(N):
                if xl[n] == k:
                    y = get_one_image(x[n,:], L, l)
                    r = np.dot(u[kQ:kkQ,:], y.T)
                    for i in range(LL):
                        q = np.argmax(r[:,i])
                        v[q,:] = v[q,:] + y[i,:]
                        s[q+kQ] = s[q+kQ] + 1
            for q in range(Q):
                if s[q+kQ] > 0:
                    u[q+kQ,:] = v[q,:]/np.linalg.norm(v[q,:])
            print(round((k*tm+t+1)*100.0/(K*tm),2), '%', end='\r', flush=True)
    print()
    QQ = np.sum(s>0)
    v, vl, n = np.zeros((QQ,ll)), np.zeros(QQ, dtype='uint8'), 0
    for q in range(K*Q):
        if s[q] > 0:
            v[n,:], vl[n] = u[q,:], ul[q]
            n = n + 1
    print("number of features=", len(vl))
    return (v, vl)

def duck_test(y, yl, u, ul, K, L, l):
    (J, M), acc, LL = y.shape, 0, (L-l-4)*(L-l-4)
    for n in range(J):
        z = get_one_image(y[n,:], L, l)
        r = np.dot(u, z.T)
        h = np.zeros(K).astype(int)
        for i in range(LL):
            q = np.argmax(r[:,i])
            h[ul[q]] = h[ul[q]] + 1
        k = np.argmax(h)
        if k == yl[n]:
            acc = acc + 1
        if (n+1) % 100 == 0:
            print(round((n+1)*100.0/J,2), '%', end='\r', flush=True)
    print()
    return acc*100.0/J

if __name__ == '__main__':
    np.random.seed(1234) # for reproducibility
    print("Read data:")
    switch = False # Switch training data with test data
    if not switch:
        (x, xl) = read_data("train-images.idx3-ubyte",
                            "train-labels.idx1-ubyte", 60000, 784)
        (y, yl) = read_data("t10k-images.idx3-ubyte",
                            "t10k-labels.idx1-ubyte", 10000, 784)
    else:
        (y, yl) = read_data("train-images.idx3-ubyte",
                            "train-labels.idx1-ubyte", 60000, 784)
        (x, xl) = read_data("t10k-images.idx3-ubyte",
                            "t10k-labels.idx1-ubyte", 10000, 784)
    print("Extract features:")
    (u, ul) = features(x, xl, 10, 28, 18, 3000, 3)
    print("Apply the duck test:")
    acc = duck_test(y, yl, u, ul, 10, 28, 18)
    print("Accuracy =", round(acc, 3), "%")
```

To run the program on a Linux computer use:

```
time python3 duck_test.py
```

The output will look like this:

```
Read data:
train-images.idx3-ubyte, train-labels.idx1-ubyte
t10k-images.idx3-ubyte, t10k-labels.idx1-ubyte
Extract features:
100.0 %
number of features= 22796
Apply the duck test:
100.0 %
Accuracy = 98.39 %

real 20m9.185s
user 78m6.823s
sys 1m35.762s
```

References

[1] Y. LeCun, Y. Bengio, G. Hinton, *Deep learning*, Nature, 521, 436 (2015).

[2] G. Marcus, *Deep Learning: A Critical Appraisal*, arXiv:1801.00631 (2018).

[3] *Abductive Inference: Computation, Philosophy, Technology* , Edited by J. R. Josephson and S. G. Josephson, Cambridge University Press, New York (1994).

[4] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE 86(11), 2278 (1998).

[5] M. Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, arXiv:1603.04467 (2016).

[6] Tensorflow tutorials: *A Guide to TF Layers: Building a Convolutional Neural Network*, https://www.tensorflow.org/tutorials/layers.

[7] MNIST database: https://en.wikipedia.org/wiki/MNIST_database

[8] S. Sabour, N. Frosst, G. E Hinton, *Dynamic Routing Between Capsules*, arXiv:1710.09829 (2017).