

Introduction: This report is about making bleeding edge machine learning accessibility to developer. This field is moving faster. There are so many incredible discoveries. We are going to learn to apply some latest machine learning techniques to practical example that you can complete into your own apps.

Neural networks have been since 50s but we had not much data and computing power. These two are the key elements we can recognize it as Deep Learning. Deep learning network enough data and compute, it'll suddenly become self aware. The brain is a neural network but do we really learn the deep neural net.

Let recognize a image of banana. A human can recognize a banana with different shape or different colour. But what about a machine? Does a machine can? Human don't need thousand example. But a machine needs. Human Has richer representation than a machine.

Here is a paper named **Human level concept learning through probabilistic program induction** written by Brenden M. Lake, Ruslan Salakhutdinov, Joshua B. Tenenbaum has the concept about the neural network system.

People learning new concepts can often generalize successfully from just a single example, yet machine learning algorithms typically require tens or hundreds of examples to perform with similar accuracy. People can also use learned concepts in richer ways than conventional algorithms for action, imagination and explanation. We present a computational model that captures these human learning abilities for a large class of simple visual concepts: handwritten characters from the world's alphabets. The model represents concepts as simple programs that best explain observed examples under a Bayesian criterion. On a challenging one-shot classification task, the model achieves human-level performance while outperforming recent deep learning approaches. We also present several "visual Turing tests" probing the model's creative generalization abilities, which in many cases are indistinguishable from human behavior.

What is One-shot Learning?

One-shot learning is an object categorization problem in computer vision. Whereas most machine learning based object categorization algorithms require training on hundreds or thousands of images and very large datasets, one-shot learning aims to learn information about object categories from one, or only a few, training images.

The primary focus of this article will be on the solution to this problem presented by Fei-Fei Li, R. Fergus and P. Perona in IEEE Transactions on Pattern Analysis and Machine

Intelligence, Vol28(4), 2006, which uses a generative object category model and variational Bayesian framework for representation and learning of visual object categories from a handful of training examples. Another paper, presented at the International Conference on Computer Vision and Pattern Recognition (CVPR) 2000 by Erik Miller, Nicholas Matsakis, and Paul Viola will also be discussed.

Motivation of OSL

The ability to learn object categories from few examples, and at a rapid pace, has been demonstrated in humans and it is estimated that a child has learned almost all of the 10 ~ 30 thousand object categories in the world by the age of six.

Theory

The Bayesian one-shot learning algorithm represents the foreground and background of images as parametrized by a mixture of constellation models.

Bayesian framework:

Given the task of finding a particular object in a query image, the overall objective of the Bayesian One-Shot Learning algorithm is to compare the probability that that object is present in the image and the probability that only background clutter is present in the image. If the former probability is higher, the algorithm reports the object's presence in the image, and if the latter probability is higher, the algorithm reports the absence of that object in the image. In order to compute these probabilities, the object class must be modeled from a set of (1 ~ 5) training images containing examples of that object.

$$R = \frac{p(O_{fg}|I, I_t)}{p(O_{bg}|I, I_t)} = \frac{p(I|I_t, O_{fg})p(O_{fg})}{p(I|I_t, O_{bg})p(O_{bg})},$$

One Shot Learning with Memory Augmented Neural Network:

OK so does that mean BPL is the way to go? Well, it does have its flaws. It lacks explicit knowledge of certain things like parallel lines, symmetry, and connections between ends of strokes and other strokes. And the learning isn't really transferable to other things so it's not better than deep learning in every way.

A few months later though, DeepMind challenged the paper by releasing their own called '***One shot learning with memory augmented neural networks.***' The basic idea they had was that deep learning is very data intensive, but perhaps there's a way to build a deep neural net that only needs a few examples to learn. Deep Learning without the huge datasets. So they built what's called a Memory Augmented Neural Network or MANN. A MANN has two parts, a controller which is either a feed forward neural net or LSTM neural net and an external memory module. The controller interacts with the external memory module with a number of read/write heads. Its capable of long term storage via slow updates of weights and short term storage via the external memory module. They fed it a few examples of handwritten characters and continuously trained it thousands of times on just those examples. It performed meta learning, which means it learned to learn. And guess what? It outperformed humans as well! So they proved that one-shot learning can be accomplished by using a neural network .

Meta Learning:

Meta learning is a subfield of Machine learning where automatic learning algorithms are applied on meta-data about machine learning experiments. Although different researchers hold different views as to what the term exactly means (see below), the main goal is to use such meta-data to understand how automatic learning can become flexible in solving different kinds of learning problems, hence to improve the performance of existing learning algorithms.

Flexibility is very important because each learning algorithm is based on a set of assumptions about the data, its inductive bias. This means that it will only learn well if the bias matches the data in the learning problem. A learning algorithm may perform very well on one learning problem, but very badly on the next. From a non-expert point of view, this poses strong restrictions on the use of machine learning or data mining techniques, since the relationship between the learning problem (often some kind of database) and the effectiveness of different learning algorithms is not yet understood. By using different kinds of meta-data, like properties of the learning problem, algorithm properties (like performance measures), or patterns previously derived from the data, it

is possible to select, alter or combine different learning algorithms to effectively solve a given learning problem. Critiques of meta learning approaches bear a strong resemblance to the critique of metaheuristic, which can be said to be a related problem.

Neural Turing machine:

A Neural Turing machine (NTMs) is a recurrent neural network model published by Alex Graves et al. in 2014. NTMs combine the fuzzy pattern matching capabilities of neural networks with the algorithmic power of programmable computers. An NTM has a neural network controller coupled to external memory resources, which it interacts with through attentional mechanisms. The memory interactions are differentiable end-to-end, making it possible to optimize them using gradient descent. An NTM with a long short-term memory (LSTM) network controller can infer simple algorithms such as copying, sorting, and associative recall from input and output examples.

Long short-term memory:

Long short-term memory (LSTM) is an artificial neural network architecture that supports machine learning. It is recurrent (RNN), allowing data to flow both forwards and backwards within the network.

An LSTM is well-suited to learn from experience to classify, process and predict time series given time lags of unknown size and bound between important events. Relative insensitivity to gap length gives an advantage to LSTM over alternative RNNs, hidden Markov models and other sequence learning methods in numerous applications.

We're going to build a one-shot handwritten character classifier in Python using the scipy library.

We need to import our dependencies first. We're going to want 3 libraries, numpy, scipy, and copy.

```
#!/usr/bin/python
import os
import numpy as np
from scipy.ndimage import imread
from scipy.spatial.distance import cdist
```

Once we have those we can define two variables, the number of runs we want to complete and a reference var for where we store our class labels.

```
# Parameters
nrun = 20 # Number of classification runs
path_to_script_dir = os.path.dirname(os.path.realpath(__file__))
path_to_all_runs = os.path.join(path_to_script_dir, 'all_runs')
fname_label = 'class_labels.txt' # Where class labels are stored for each run
```

Then in our main method we can create an array of the size of runs which is 20. We'll use this array to store all of our classification error rates, one every run. Then we'll write a for loop to train our algorithm 20 times. For each run, we'll run a classification function which will attempt to classify a small sample set of images, and store the error rate in in the array. Then we'll print out the error rate to terminal and when we are done with all of our runs, we'll go ahead and get the mean error rate from the array and print it out as the last statement in terminal.

```
#Main function
if __name__ == "__main__":
    #
    # Running this demo should lead to a result of 38.8% average error rate.
    #
    # M.-P. Dubuisson, A. K. Jain (1994). A modified hausdorff distance for object
    matching.
    # International Conference on Pattern Recognition, pp. 566-568.
    #
    # ** Models should be trained on images in 'images_background' directory to
    # avoid using images and alphabets used in the one-shot evaluation **
    #
    print('One-shot classification demo with Modified Hausdorff Distance')
    perror = np.zeros(nrun)
    for r in range(nrun):
        perror[r] = classification_run('run{:02d}'.format(r + 1),
```

```

load_img_as_points,
modified_hausdorff_distance,
'cost')
print(' run {:02d} (error {:.1f}%)'.format(r, perror[r]))
total = np.mean(perror)
print('Average error {:.1f}%' .format(total))

```

To understand the `classification_run` function we need to understand two function. 1. `Load_img_as_points`, 2. `modified_hausdorff_distance`

The load image as points function loads an image file, in our case this will be a character image. It then converts the image to an array and finds all the nonzero values, that is all of the inked pixels and stores that in an array. then it creates an array of all the coordinates of those pixels and returns that.

```

def load_img_as_points(filename):
    # Load image file and return coordinates of black pixels in the binary image
    #
    # Input
    # filename : string, absolute path to image
    #
    # Output:
    # D : [n x 2] rows are coordinates
    #
    I = imread(filename, flatten=True)
    # Convert to boolean array and invert the pixel values
    I = ~np.array(I, dtype=np.bool)
    # Create a new array of all the non-zero element coordinates
    D = np.array(I.nonzero()).T
    return D - D.mean(axis=0)

```

The modified hausdorff distance is a metric that computes the similarity between two images by comparing their pixel coordinate arrays. That comes from the load image as points functions. It calculates the mean difference between both images and returns it.

```

def modified_hausdorff_distance(itemA, itemB):
    # Modified Hausdorff Distance
    #
    # Input
    # itemA : [n x 2] coordinates of black pixels
    # itemB : [m x 2] coordinates of black pixels
    #

```

```

# M.-P. Dubuisson, A. K. Jain (1994). A modified hausdorff distance for object
matching.
# International Conference on Pattern Recognition, pp. 566-568.
#
D = cdist(itemA, itemB)
mindist_A = D.min(axis=1)
mindist_B = D.min(axis=0)
mean_A = np.mean(mindist_A)
mean_B = np.mean(mindist_B)
return max(mean_A, mean_B)

```

The last parameter of the classification function, `'cost'`, just notifies the function that small values from the modified hausdorff distance mean more similar.

Now to the classification function itself. In this function, we retrieve both our training and testing images and load their image point matrices into memory. Then we compute the cost matrix using the modified hausdorff distance. After that, we compute the error rate and return it. That's all!

```

def classification_run(folder, f_load, f_cost, ftype='cost'):
    # Compute error rate for one run of one-shot classification
    #
    # Input
    # folder : contains images for a run of one-shot classification
    # f_load : itemA = f_load('file.png') should read in the image file and
    #          process it
    # f_cost : f_cost(itemA,itemB) should compute similarity between two
    #          images, using output of f_load
    # ftype : 'cost' if small values from f_cost mean more similar,
    #          or 'score' if large values are more similar
    #
    # Output
    # perror : percent errors (0 to 100% error)
    #
    assert ftype in {'cost', 'score'}

    with open(os.path.join(path_to_all_runs, folder, fname_label)) as f:
        pairs = [line.split() for line in f.readlines()]
    # Unzip the pairs into two sets of tuples
    test_files, train_files = zip(*pairs)

    answers_files = list(train_files) # Copy the training file list
    test_files = sorted(test_files)
    train_files = sorted(train_files)
    n_train = len(train_files)
    n_test = len(test_files)

```

```

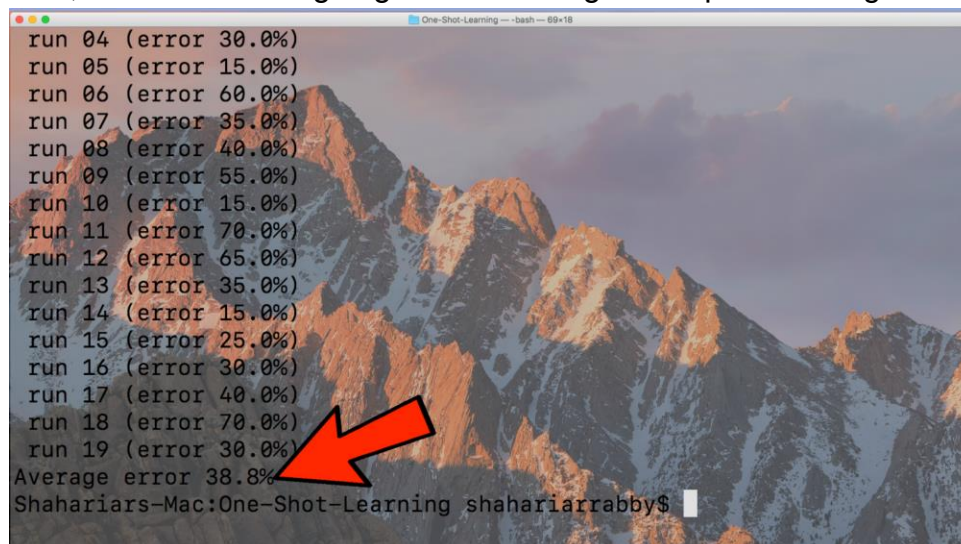
# Load the images (and, if needed, extract features)
train_items = [f_load(os.path.join(path_to_all_runs, f))
                for f in train_files]
test_items = [f_load(os.path.join(path_to_all_runs, f))
               for f in test_files]

# Compute cost matrix
costM = np.zeros((n_test, n_train))
for i, test_i in enumerate(test_items):
    for j, train_j in enumerate(train_items):
        costM[i, j] = f_cost(test_i, train_j)
if ftype == 'cost':
    y_hats = np.argmin(costM, axis=1)
elif ftype == 'score':
    y_hats = np.argmax(costM, axis=1)
else:
    # This should never be reached due to the assert above
    raise ValueError('Unexpected ftype: {}'.format(ftype))

# compute the error rate by counting the number of correct predictions
correct = len([1 for y_hat, answer in zip(y_hats, answers_files)
               if train_files[y_hat] == answer])
pcorrect = correct / float(n_test) # Python 2.x ensure float division
perror = 1.0 - pcorrect
return perror * 100

```

We can see that when we run this code, I'll return an average error rate of around a third. Which isn't state-of-the-art like DeepMind or BPL but it does make for a good baseline demo of one-shot learning. One shot learning will only get more popular over time, and soon we're going to start seeing lots of production grade code using this.

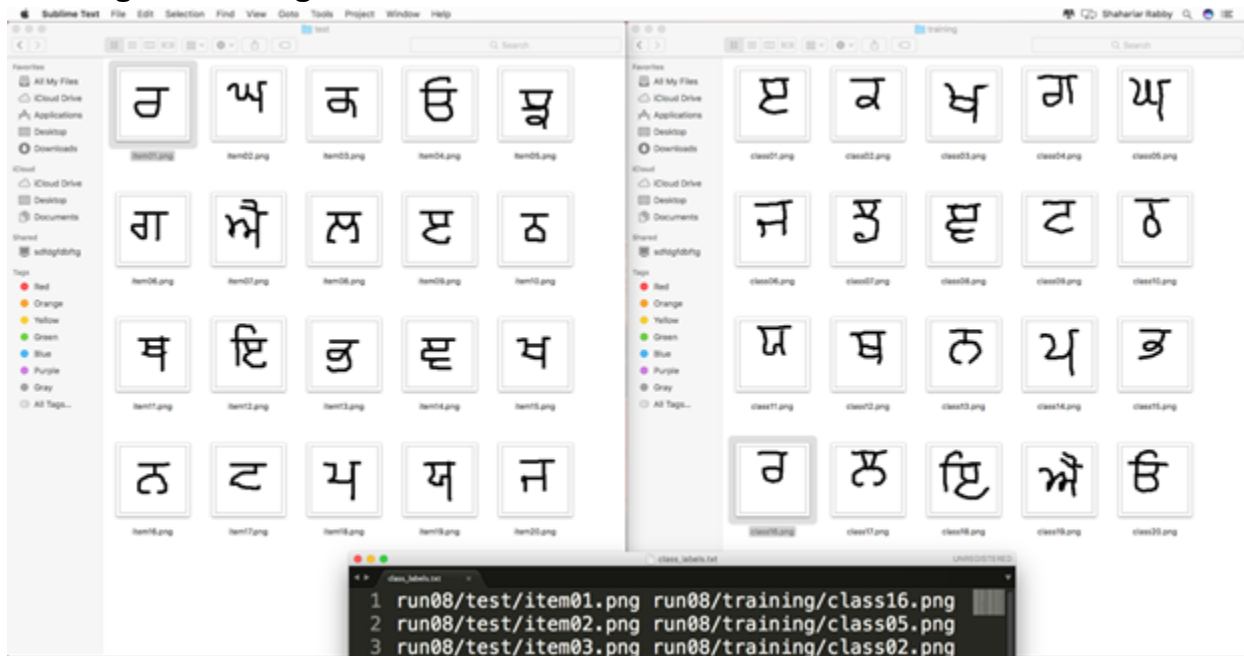


```

run 04 (error 30.0%)
run 05 (error 15.0%)
run 06 (error 60.0%)
run 07 (error 35.0%)
run 08 (error 40.0%)
run 09 (error 55.0%)
run 10 (error 15.0%)
run 11 (error 70.0%)
run 12 (error 65.0%)
run 13 (error 35.0%)
run 14 (error 15.0%)
run 15 (error 25.0%)
run 16 (error 30.0%)
run 17 (error 40.0%)
run 18 (error 70.0%)
run 19 (error 30.0%)
Average error 38.8%
Shahariars-Mac:One-Shot-Learning shahariarrabby$

```


Training and testing set:



Source:

1. Human Level Concept Learning through Probabilistic Program Induction: <http://web.mit.edu/cocosci/Papers/Science-2015-Lake-1332-8.pdf>
2. One-Shot Learning with Memory Augmented Neural Networks: <https://arxiv.org/pdf/1605.06065v1.pdf>
3. For Code & Papers: <https://github.com/shahariarrabby/One-Shot>