Implementation of DEN using CORe50 dataset

The training images are followed by the test images.

's10/o6/C_10_06_298.png' is an example of the format the image paths are stored in the paths.pkl file where the name of the image is in the form "C_[session_num]_[obj_num]_[frame_seq_id].png".

The number of tasks for MNIST are 10 but what will be the task number for CORe50?

Also what will the different tasks be?

Training, validation and test sets

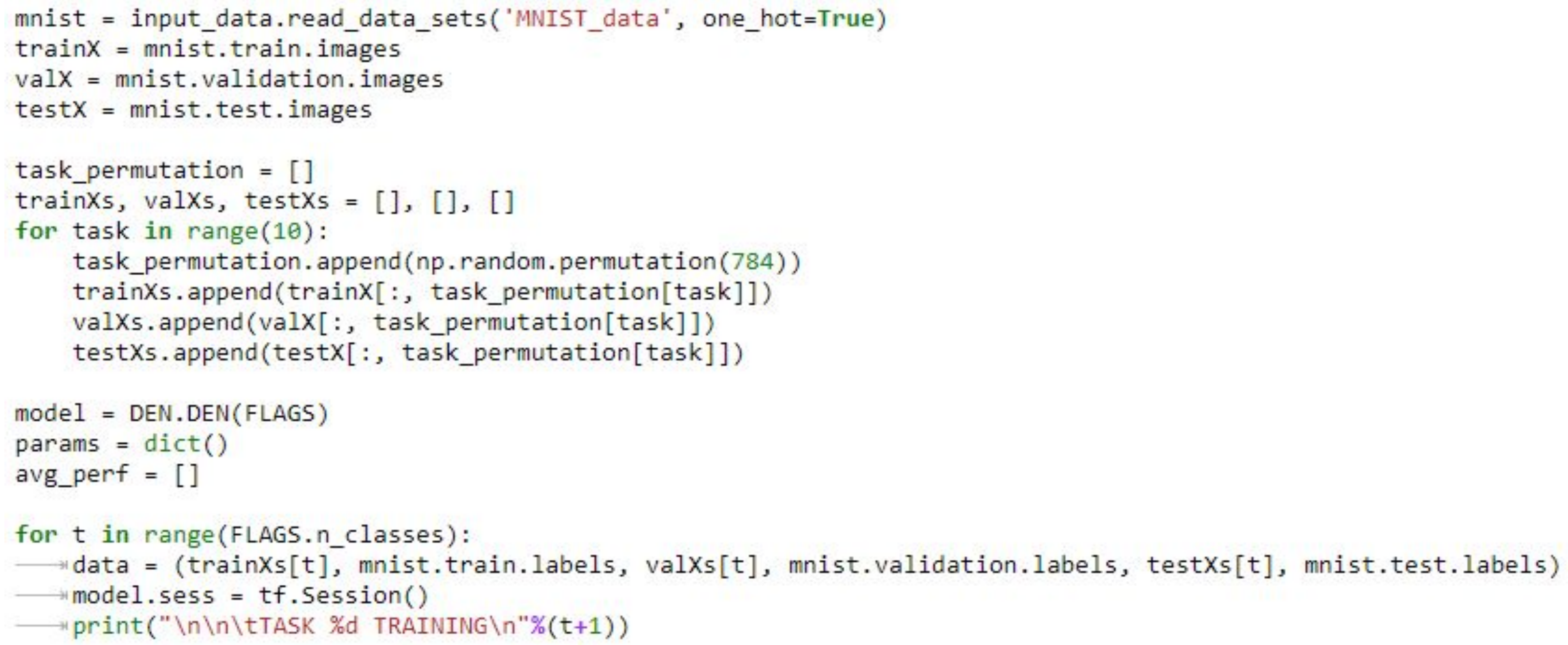
1. The test set consists of the sessions #3, #7 and #10 amongst the 11 sessions.
2. Rest of the sessions belong to the training set.
3. I have used session #5 and #8 from the training set for validation.
4. The dataset has the following dimensions (164866, 128, 128, 3)

Updates as of August 13th, 2019

- Last week I spoke to you about a new dataset called CORe50.
- This week first I figured out the training and test sets for the different classes in the previous paper on Dynamically Expandable Network.
- I also read the paper on the CORe50 dataset that you sent me. It had a lot of things that I had to read and understand from scratch like layer freezing, CWR method for continual learning, etc.
- I have explained that paper in detail next.
- I am still working on the implementation of that paper as it has many dependencies to be installed first.
- Next week I will be working on that implementation and also working further on the DEN method.

DEN: Training, validation and testing sets for each task

The pixels in each image are permuted and different such permutations are used for different tasks in all the three sets.



```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
trainX = mnist.train.images
valX = mnist.validation.images
testX = mnist.test.images

task_permutation = []
trainXs, valXs, testXs = [], [], []
for task in range(10):
    task_permutation.append(np.random.permutation(784))
    trainXs.append(trainX[:, task_permutation[task]])
    valXs.append(valX[:, task_permutation[task]])
    testXs.append(testX[:, task_permutation[task]])

model = DEN.DEN(FLAGS)
params = dict()
avg_perf = []

for t in range(FLAGS.n_classes):
    data = (trainXs[t], mnist.train.labels, valXs[t], mnist.validation.labels, testXs[t], mnist.test.labels)
    model.sess = tf.Session()
    print("\n\n\tTASK %d TRAINING\n"%(t+1))
```

Fine-Grained Continual Learning

1) Some research has shown that reducing the size of training batches makes continual learning more challenging.
2) CORe50 is a video-based dataset and frames extracted from videos would constitute one or more small mini-batches containing highly correlated patterns from a single class.
3) This challenging task was completed using various methods in the paper which are described on the following slide.
4) I also came across many new terms while reading this paper which I will talk about.

CWR

- CWR+ is used instead of CWR: CopyWeights with Re-init where layers fc6 and fc7 in the CNN are skipped and directly connect pool5 to a final layer fc8 (followed by softmax) while maintaining the weights up to pool5 fixed. This allows isolating the subsets of weights that each class uses.
- In CWR+ the fully connected layer is implemented as a double memory, is quite effective to control forgetting in the Single Incremental Task (SIT) - New Classes (NC) scenario.
- There is also an extension of CWR+ given in this paper called CWR* which works both under NC and NIC update type.
- Both the algorithms are given next.

```
1: procedure CWR+
2:     $cw = 0$
3:     init $\bar{\Theta}$ random or from pre-trained model (e.g. on ImageNet)
4:     for each training batch $B_i$:
5:         expand output layer with neurons for the new classes in $B_i$
6:         $tw = 0$ (for all neurons in the output layer)
7:         train the model with SGD on the $s_i$ classes of $B_i$:
8:             if $B_i = B_1$ learn both $\bar{\Theta}$ and $tw$
9:             else learn $tw$ while keeping $\bar{\Theta}$ fixed
10:        for each class $j$ among the $s_i$ classes in $B_i$
11:            $cw[j] = tw[j] - avg(tw)$
12:        test each class $j$ by using $\bar{\Theta}$ and $cw$
```

Here, cw are the consolidated weights used for inference and tw the temporary weights used for training.

The mean-shift in line 11 allows to adapt the scale of parameters trained in different batches.

```
procedure CWR*
    $cw = 0$
    $past = 0$
    init $\bar{\Theta}$ random or from pre-trained model (e.g. on ImageNet)
    for each training batch $B_i$:
        expand output layer with neurons for the new classes in $B_i$ never seen before
        $tw[j] = \begin{cases} cw[j], & \text{if class } j \text{ in } B_i \\ 0, & \text{otherwise} \end{cases}$
        train the model with SGD on the $s_i$ classes of $B_i$:
            if $B_i = B_1$ learn both $\bar{\Theta}$ and $tw$
            else learn $tw$ while keeping $\bar{\Theta}$ fixed
        for each class $j$ in $B_i$:
            $wpast_j = \sqrt{\frac{past_j}{cur_j}}$, where $cur_j$ is the number of patterns of class $j$ in $B_i$
            $cw[j] = \frac{cw[j] \cdot wpast_j + (tw[j] - avg(tw))}{wpast_j + 1}$
            $past_j = past_j + cur_j$
        test the model by using $\bar{\Theta}$ and $cw$
```

For already known classes, instead of resetting weights to 0, we reload the consolidated weights. The weight wpastj used for the first term is proportional to the ratio pastj / curj where pastj is the number of times patterns of class j were encountered in past batches whereas curj is their count in the current batch.

Batch Renormalization

- In Batch Normalization (BN) the mini-batch moments like mean and variance are used to normalize the input values.
- However, if mini-batches are small, the moments are not stable.
- Hence, Batch Renormalization (BRN) is done as follows:

$$\hat{x_i} = \frac{x_i - \mu_{mb}}{\sigma_{mb}} \cdot r + d, \text{ where } r = \frac{\sigma_{mb}}{\sigma}, d = \frac{\mu_{mb} - \mu}{\sigma}$$

where μ, σ are computed as moving averages during training

- Therefore, here there is a dependency on the global moments.
- r is in the range [1 / rmax, rmax] and d in the range [-dmax, dmax].
- Start by keeping rmax = 1 and dmax = 0 where BRN = BN, and then increase the values.

Depthwise Layer Freezing

- For example, a 5×5×32 filter spans a spatial neighborhood of 5×5 along 32 feature maps; on the contrary, in Depth-wise Separable Convolution (DWSC) we first perform 32 5×5×1 spatial convolutions (an independent convolution on each feature maps) and then combine results with a 1×1×32 filter working as a feature map pooler.
- The proposed strategy where the weights of depthwise convolution layers are frozen achieves the best result and, with respect to a full tuning, also allows skipping some gradient computations and can reduce the amount of memory used to store weight associated data.
- Freezing prevents the weights of a neural network layer from being modified during the backward pass of training, so here the weights of depthwise convolution layers will not be modified, which prevents forgetting.

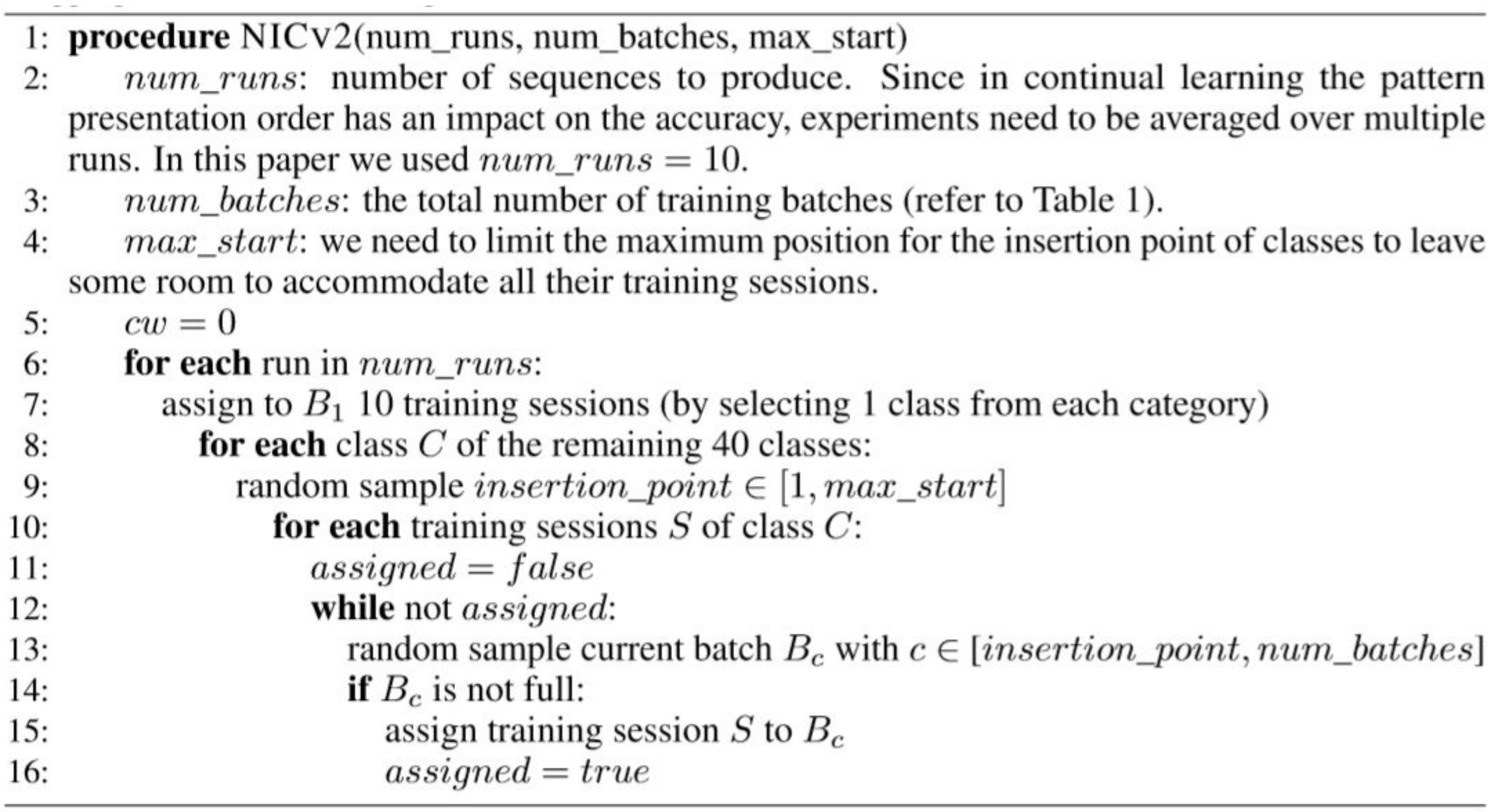
CORe50

1) Stands for (C)ontinuous (O)bject (Re)cognition is a collection of 50 objects from 10 categories.
2) Classification can be performed at object level (50 classes) or category level (10 classes).
3) The dataset consists of 15 second videos of the objects in 11 distinct sessions.
4) As these are videos where objects gently move in front of the camera there is temporal coherence which simplifies object detection and improves classification accuracy.
5) The dataset consists of 164,866 128×128 RGB-D images: 11 sessions × 50 objects × (around 300) frames per session.

Training set scenarios

1. NI : New Instances - Training a model on the first session and tuning it 7 times, for the remaining 7 sessions, where all the classes are known from the first batch, and successive batches provide new instances of these classes to refine and consolidate knowledge.
2. NC : New Classes - Each sequential batch consists of the whole training sequences of a small group of classes of objects. The first batch includes 10 classes, and the subsequent batches 5 classes each.
3. NIC : New Instances and Classes - Just like NC, the first batch includes 10 classes, and the subsequent batches 5 classes each, but only one training sequence per class is here included in a batch, thus resulting in a double partitioning scheme.

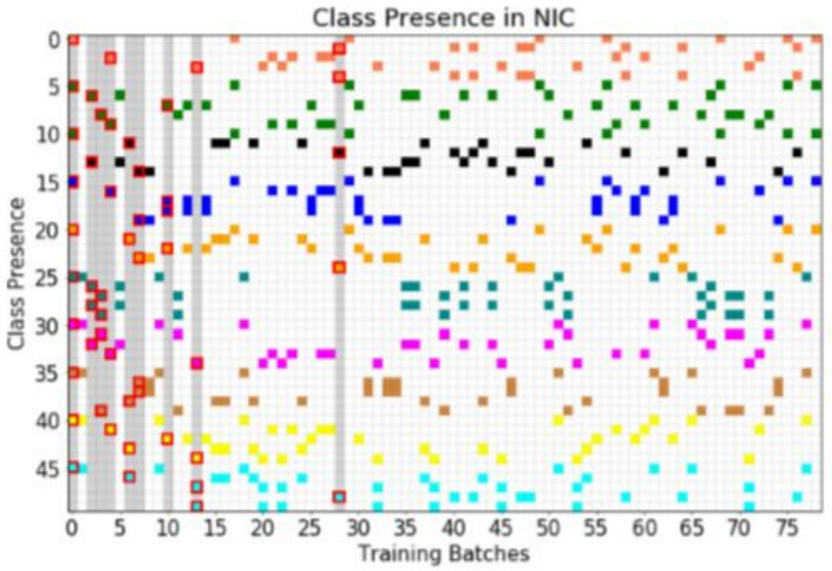
Algorithm



```
1:  procedure NICv2(num_runs, num_batches, max_start)
2:      $num\_runs$: number of sequences to produce. Since in continual learning the pattern
    presentation order has an impact on the accuracy, experiments need to be averaged over multiple
    runs. In this paper we used $num\_runs = 10$.
3:      $num\_batches$: the total number of training batches (refer to Table 1).
4:      $max\_start$: we need to limit the maximum position for the insertion point of classes to leave
    some room to accommodate all their training sessions.
5:      $cw = 0$
6:      for each run in $num\_runs$:
7:          assign to $B_1$ 10 training sessions (by selecting 1 class from each category)
8:              for each class $C$ of the remaining 40 classes:
9:                  random sample $insertion\_point \in [1, max\_start]$
10:                 for each training sessions $S$ of class $C$:
11:                     $assigned = false$
12:                     while not $assigned$:
13:                         random sample current batch $B_c$ with $c \in [insertion\_point, num\_batches]$
14:                         if $B_c$ is not full:
15:                             assign training session $S$ to $B_c$
16:                             $assigned = true$
```

Experiment

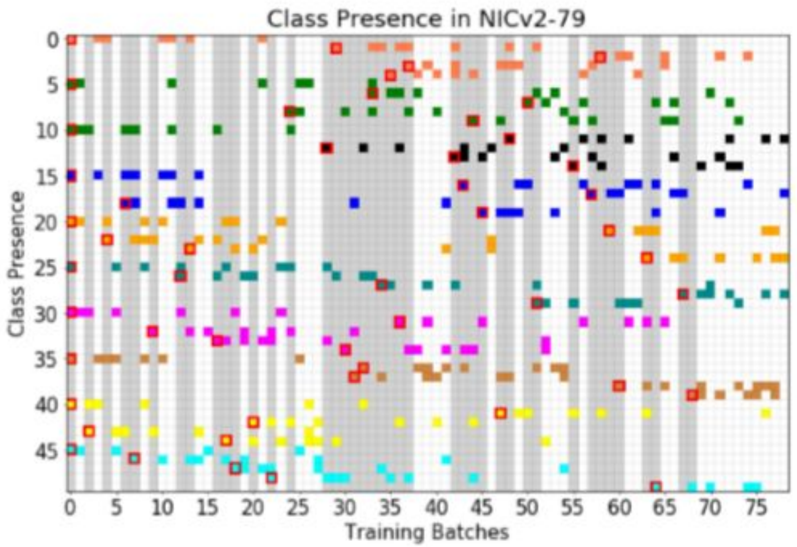
- If a random generation procedure is used for training then all the classes are introduced in the first 10-15 batches making this protocol very close to an NI scenario.



- In this figure, there are 79 training batches used. There are 50 objects, hence, 50 classes. The different colors are for different categories of objects.
- A colored cell is used to indicate that at least one training session of the row class is present in the column batch.
- The red-framed cells denote the first introduction of a class. Gray vertical bands highlight batches where at least one class is seen for the first time.

Experiment

- Hence, a better method called NICv2 protocol is used where objects will be discovered also later in time. Here, classes first introduction is more balanced over the training batches and the batch size is progressively reduced, leading to a higher number of fine-grained updates.
- This protocol forces the classes first introduction to be evenly distributed across the batches thus producing runs that are both more challenging and realistic.



Experiment and conclusion

- The dataset is tested for different number of batches like 79, 196 and 391 with the initial batch have a constant number of classes and the incremental batches having fewer classes with an increase in number of batches.
- Now, the training set consisted of 8 sessions out of the 11 sessions in the dataset. While the test set has the remaining 3 sessions (#3, #7 and #10).
- It is seen that CWR* performs well where even there the well known continual learning techniques like Naive perform badly.
- In the paper it is shown that CWR* works well for fine grained continual learning with batches as small as having one class at a time.
- Thus, in continual learning, it is important to learn upper layers in isolation as in CWR* especially in SIT setting.

Implementation details

1. The original image frames are 350 x 350 but the cropped ones are 128 x 128 which consist of the CORe50 benchmark and their bounding boxes are mentioned with respect to the original image size.
2. Both object detection and object segmentation can be done using the CORe50 dataset.

Updates as of August 5th, 2019

1) Last week I compared the DEN system with regular feed forward neural network which confirmed the advantages of lifelong learning.
2) I also read and understood some papers on different applications of continual learning in computer vision.
3) This week I reviewed a new dataset which is suitable for continual learning.
4) Next week I wish to implement this dataset.

CORe50: a new Dataset and Benchmark for Continual / Lifelong Deep Learning

1) Stands for (C)ontinuous (O)bject (Re)cognition is a collection of 50 objects from 10 categories.
2) Classification can be performed at object level (50 classes) or category level (10 classes).
3) The dataset consists of 15 second videos of the objects in 11 distinct sessions.
4) As these are videos where objects gently move in front of the camera there is temporal coherence which simplifies object detection and improves classification accuracy.
5) The dataset consists of 164,866 128×128 RGB-D images: 11 sessions × 50 objects × (around 300) frames per session.

Updates as of July 18th, 2019

1) Last week I implemented the paper DEN and studied the results.
2) This week I compared the DEN system with regular feed forward neural network which confirmed the advantages of lifelong learning.
3) I also read and understood some papers on different applications of continual learning in computer vision which I have explained later.

What have we learned?

1. Lifelong learning is useful in scenarios where the distribution of data stays the same but data keeps coming.
2. Neural networks face the problem of catastrophic forgetting because when a new task is introduced to the network, new adaptations to the weights are overwritten on the previous ones.
3. Continual learning optimizes models for accuracy, improves model performance and saves retraining time.
4. Lifelong learning is a promising research direction for building better classifiers.

Comparison of lifelong learning with regular feed-forward network

MNIST Handwritten Digits Classification using 3 Layer Neural Net gave 98.7% Accuracy. 785 neurons (including bias) were used in the input layer, 300 in the hidden layer and 10 in the output layer for the 10 classes.
Source:https://github.com/aditya9211/MNIST-Classification-with-NeuralNet

But in in continual learning using dynamically expandable network we achieve an accuracy of about 99%.

Also, deep neural networks learn multiple tasks only if all the data is presented at once. But in real life situations this is not possible.

Towards continual learning in medical imaging

By Chaitanya Baweja, Ben Glocker and Konstantinos Kamnitsas

1. Elastic Weight Consolidation (EWC) is used with Fisher information for classification.
2. The two tasks used are multi-class segmentation of cerebrospinal fluid (CSF), grey matter (GM), white matter (WM) and segmentation of white matter lesions (WML).
3. This paper compares the L2 regularization with the EWC regularization.
4. In EWC, importance of each parameter with respect to behaviour of the model is investigated.
5. The Fisher Information Matrix, F quantifies how much a change of a parameter’s value is expected to affect the output of a network $p(y|x,\theta)$.

Lifelong Learning for Sentiment Classification

By Zhiyuan Chen, Nianzu Ma, Bing Liu.

1. It used Naive Bayes with stochastic gradient descent optimization. Binary classification was done.
2. 20 datasets (for 20 tasks) of 20 products was crawled from Amazon.com. Each dataset had 1000 reviews. A rating of greater than 3 was treated as positive and that of smaller than 3 was negative.
3. Penalty terms are introduced to effectively exploit the knowledge gained from past learning.
4. The intuition here is that if a word w can distinguish classes very well from the target domain training data, we should rely more on the target domain training data in computing counts related to w.

Lifelong Learning for Sentiment Classification

5. F1 score of natural class distributed reviews was taken whereas accuracy of balanced class distributed reviews was obtained.

6. The results obtained for the first one were around 67% and for the latter were 83%.

7. The penalty terms were a little difficult for me to understand but was able to acquire the gist of the paper.

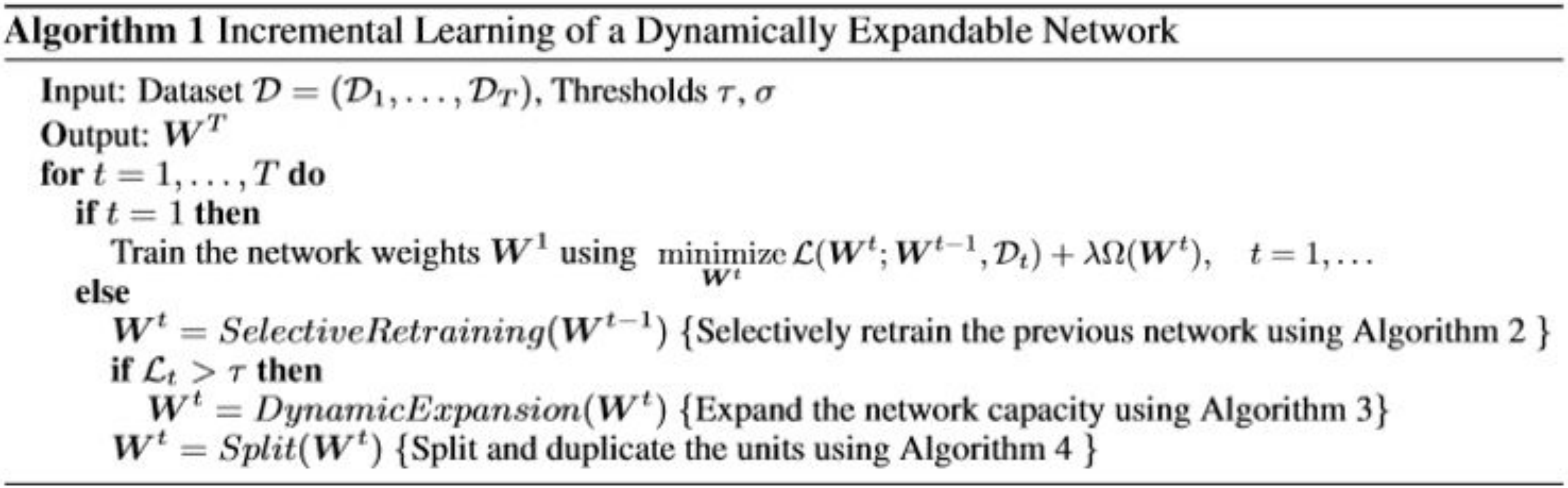
Updates as of July 11th, 2019

1) Last week I read the above mentioned paper and tried to understand the methodology.
2) This week I implemented the code for this paper and interpreted it and the results were obtained.
3) I also read the paper, “Learning Efficient Convolutional Networks through Network Slimming” in which sparsity-induced regularization is imposed on the scaling factors in batch normalization layers to identify unimportant channels.
4) Next week the plan probably should be to make changes in the existing system or combining two different systems and obtaining the results.

Dynamically expandable network

1) It can dynamically decide its network capacity as it trains on a sequence of tasks.
2) It performs selective retraining with only the necessary number of units.
3) It effectively prevents semantic drift or catastrophic forgetting by splitting or duplicating units or timestamping them.

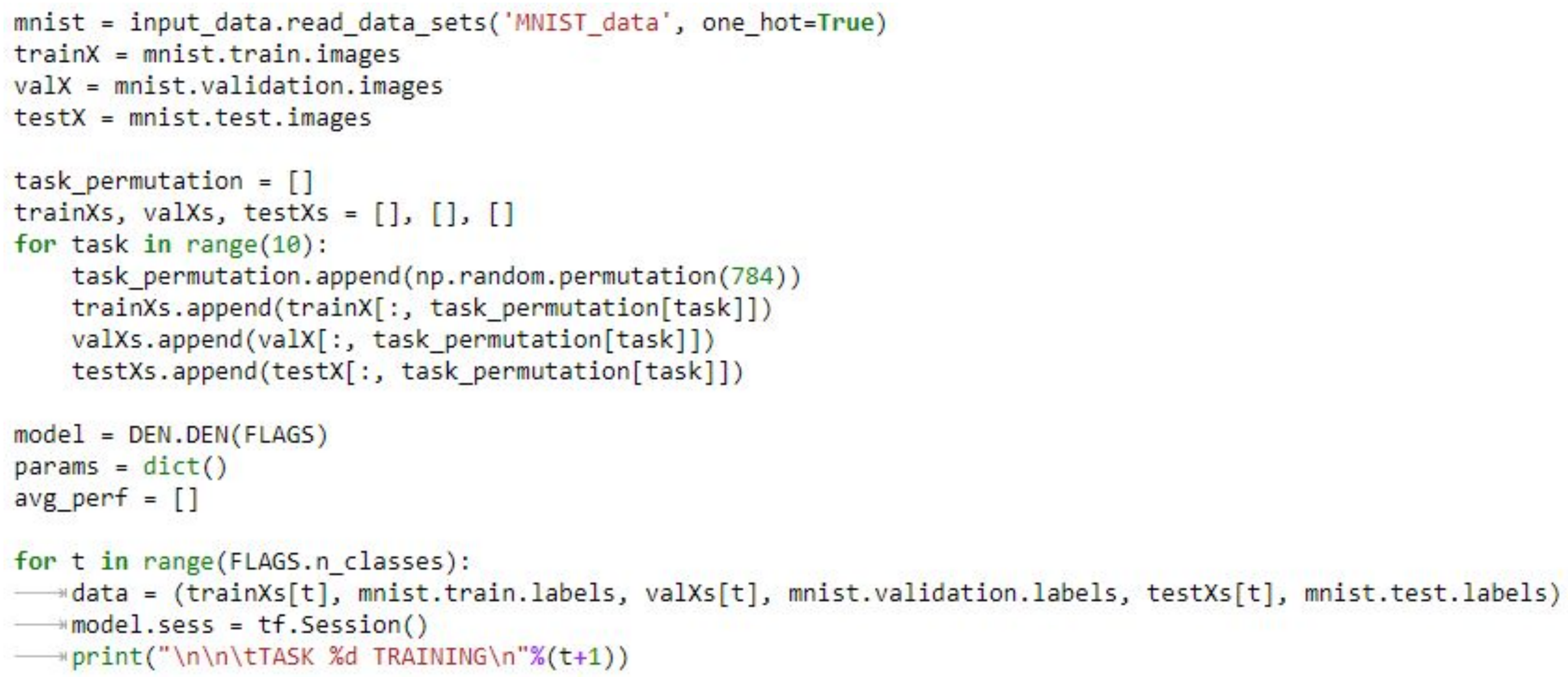
DEN algorithm



Algorithm 1 Incremental Learning of a Dynamically Expandable Network

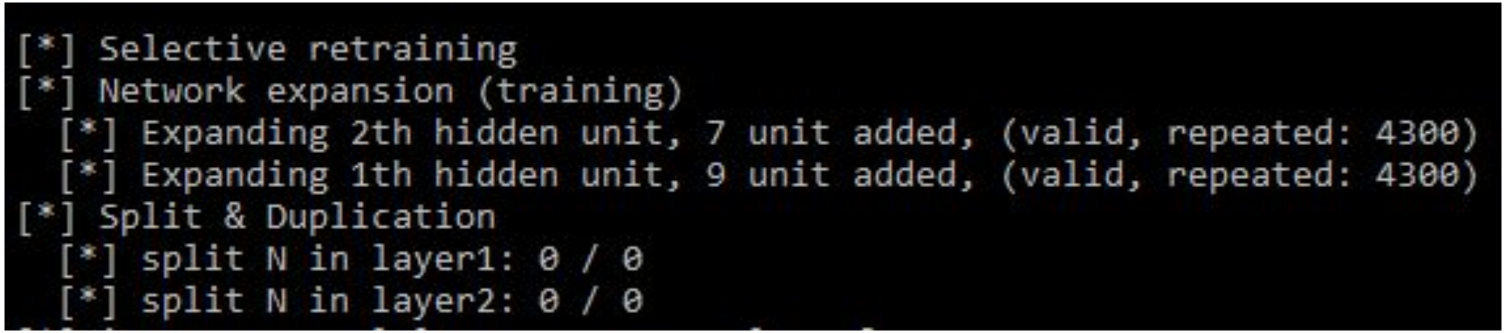
```
Input: Dataset $\mathcal{D} = (\mathcal{D}_1, \ldots, \mathcal{D}_T)$, Thresholds $\tau$, $\sigma$
Output: $\boldsymbol{W}^T$
for $t = 1, \ldots, T$ do
  if $t = 1$ then
    Train the network weights $\boldsymbol{W}^1$ using $\underset{\boldsymbol{W}^t}{\text{minimize}}\, \mathcal{L}(\boldsymbol{W}^t; \boldsymbol{W}^{t-1}, \mathcal{D}_t) + \lambda\Omega(\boldsymbol{W}^t), \quad t = 1, \ldots$
  else
    $\boldsymbol{W}^t = SelectiveRetraining(\boldsymbol{W}^{t-1})$ {Selectively retrain the previous network using Algorithm 2 }
    if $\mathcal{L}_t > \tau$ then
      $\boldsymbol{W}^t = DynamicExpansion(\boldsymbol{W}^t)$ {Expand the network capacity using Algorithm 3}
    $\boldsymbol{W}^t = Split(\boldsymbol{W}^t)$ {Split and duplicate the units using Algorithm 4 }
```

MNIST data is used in the implementation. The pixels in each image are permuted and different such permutations are used for different classes.



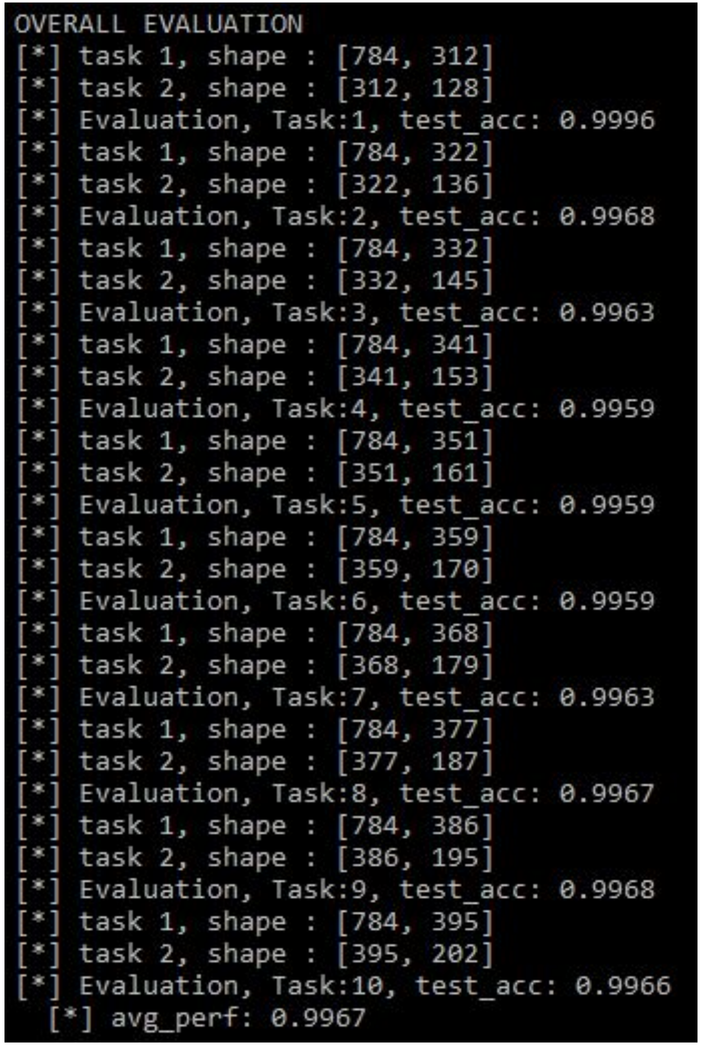
```python
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
trainX = mnist.train.images
valX = mnist.validation.images
testX = mnist.test.images

task_permutation = []
trainXs, valXs, testXs = [], [], []
for task in range(10):
    task_permutation.append(np.random.permutation(784))
    trainXs.append(trainX[:, task_permutation[task]])
    valXs.append(valX[:, task_permutation[task]])
    testXs.append(testX[:, task_permutation[task]])

model = DEN.DEN(FLAGS)
params = dict()
avg_perf = []

for t in range(FLAGS.n_classes):
    data = (trainXs[t], mnist.train.labels, valXs[t], mnist.validation.labels, testXs[t], mnist.test.labels)
    model.sess = tf.Session()
    print("\n\n\tTASK %d TRAINING\n"%(t+1))
```

Intermediate results



```
[*] Selective retraining
[*] Network expansion (training)
  [*] Expanding 2th hidden unit, 7 unit added, (valid, repeated: 4300)
  [*] Expanding 1th hidden unit, 9 unit added, (valid, repeated: 4300)
[*] Split & Duplication
  [*] split N in layer1: 0 / 0
  [*] split N in layer2: 0 / 0
```

The results for task number 10, show the different steps the DEN algorithm goes through. First the selective retraining is done. Then the network is expanded by checking if the loss is greater than a threshold, also useless units are deleted. Further, the highly drifted parameters are found and splitted.



```
OVERALL EVALUATION
[*] task 1, shape : [784, 312]
[*] task 2, shape : [312, 128]
[*] Evaluation, Task:1, test_acc: 0.9996
[*] task 1, shape : [784, 322]
[*] task 2, shape : [322, 136]
[*] Evaluation, Task:2, test_acc: 0.9968
[*] task 1, shape : [784, 332]
[*] task 2, shape : [332, 145]
[*] Evaluation, Task:3, test_acc: 0.9963
[*] task 1, shape : [784, 341]
[*] task 2, shape : [341, 153]
[*] Evaluation, Task:4, test_acc: 0.9959
[*] task 1, shape : [784, 351]
[*] task 2, shape : [351, 161]
[*] Evaluation, Task:5, test_acc: 0.9959
[*] task 1, shape : [784, 359]
[*] task 2, shape : [359, 170]
[*] Evaluation, Task:6, test_acc: 0.9959
[*] task 1, shape : [784, 368]
[*] task 2, shape : [368, 179]
[*] Evaluation, Task:7, test_acc: 0.9963
[*] task 1, shape : [784, 377]
[*] task 2, shape : [377, 187]
[*] Evaluation, Task:8, test_acc: 0.9967
[*] task 1, shape : [784, 386]
[*] task 2, shape : [386, 195]
[*] Evaluation, Task:9, test_acc: 0.9968
[*] task 1, shape : [784, 395]
[*] task 2, shape : [395, 202]
[*] Evaluation, Task:10, test_acc: 0.9966
  [*] avg_perf: 0.9967
```

The screengrab of the terminal for the overall evaluation shows that the test accuracies for all the tasks are approximately 99%.