Core50 Continual Deep Learning Network

Authors:
David Apolinar - da468@njit.edu
Shawn Cicoria - sc2443@njit.edu
Ted Moore - tm437@njit.edu

October 18, 2020

NJIT CS677 - Deep Learning
Fall 2020
Pantelis Monogioudis

Background

Introduction

Continual Learning in the world of Artificial Neural Networks refers to the ability of a Network to observe new data without losing its ability to accurately classify previously seen data. These new data can be new instances of *known classes*, instances in new *unknown classes,* or both. It has been observed that feeding new data into a previously trained network can result in *catastrophic forgetting*, which means that feeding in drastically different data to a previously trained network can significantly impede its ability to classify cases it observed in prior training, but has not seen again.

In this project we sought to bring the concepts of continual learning to the Core50 dataset as described in the CVPR-2020 CLVision challenge. The Core50 dataset includes images of 50 different object classes. Each object is rotated gently in front of the camera to provide several angles of a given object.

Per the requirements of the assignment, we focused our efforts on the "New Classes" challenge. In this scenario, the dataset is split into 9 unique batches of images (please note, this is not to be confused with "mini-batches" as used in training). Each batch contains images from classes *not yet*

observed by the algorithm. The first batch includes 10 such classes, and each of the 8 subsequent batches include 5 such classes.

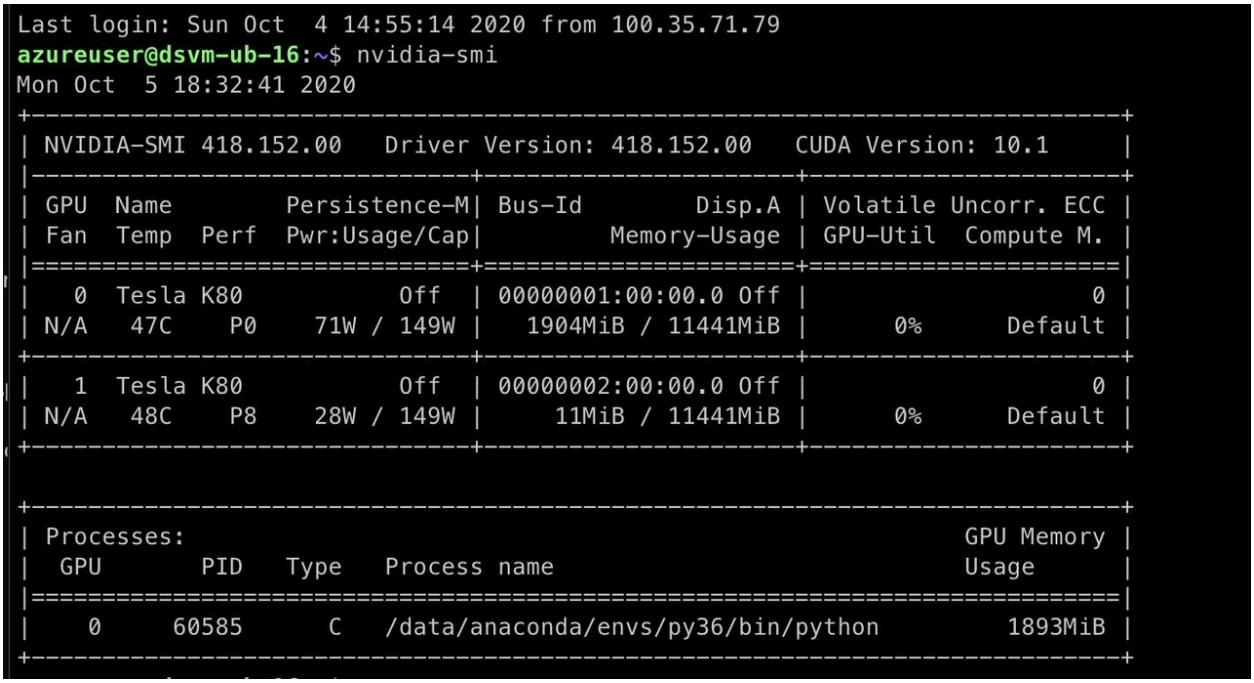
The dataset is roughly 27GB is size due to the many instances contained in the 11 sessions provided. The data folder structure is broken out as follows:

```
[o1, ..., o5] -> plug adapters
[o6, ..., o10] -> mobile phones
[o11, ..., o15] -> scissors
[o16, ..., o20] -> light bulbs
[o21, ..., o25] -> cans
[o26, ..., o30] -> glasses
[o31, ..., o35] -> balls
[o36, ..., o40] -> markers
[o41, ..., o45] -> cups
[o46, ..., o50] -> remote controls
```

Test Environment

One of the challenges with picking the Core50 dataset is the shear amount of time required to train our models. In order to make this project feasible, we chose to leverage Microsoft's Azure Data Science Virtual Machine. Using a cloud based machine gave us access to Nvidia Tesla K80 CUDA cores. For this particular environment, we used a Standard_NC12 virtual machine which consists of the following specs - output of the nvidia-smi command below:

Size	vCPU	Memory: GiB	GPU	GPU memory: GiB
Standard_NC12	12	112	2	24

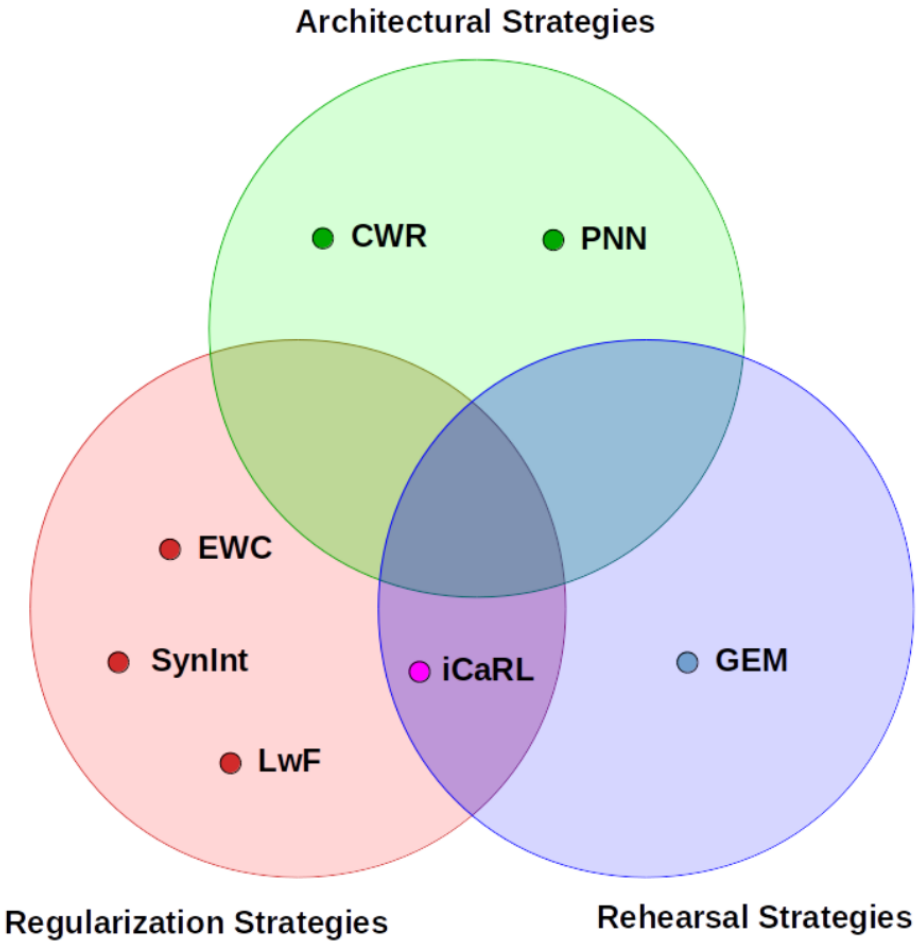


This virtual machine gave us easy access to a Jupyter hub development environment which made it easier to collaborate. Since we choose to use the core50 dataset, the primary deep learning framework used was PyTorch. This was primarily due to the continuum python library having dependencies on PyTorch and many research articles and coding examples referencing PyTorch.

Several Strategies

Continuous Learning Strategies are generally grouped into the following three strategies as per documented by Vincenzo Lomocaco in his core50 site:
https://vlomonaco.github.io/core50/strategies.html#start

- Architectural Strategies
- Regularization Strategies
- Rehearsal Strategies



Experiments

Naive

This is running standard Train and Test against the dataset without accommodating any Continual Learning techniques. Each task trains against the one set of the classes and validated against the full dataset of classes.

EWC

The team also looked at implementation of Elastic Weight Composition (EWC), with the Core50 dataset. However, the final implementation was not complete, and the code is annotated with some comments.

GEM - Replay

The team focused primarily on **Replay** with a technique that follows Gradient Episodic Memory (GEM) as articulated in [1 - Lopez-Paz,] section 3.

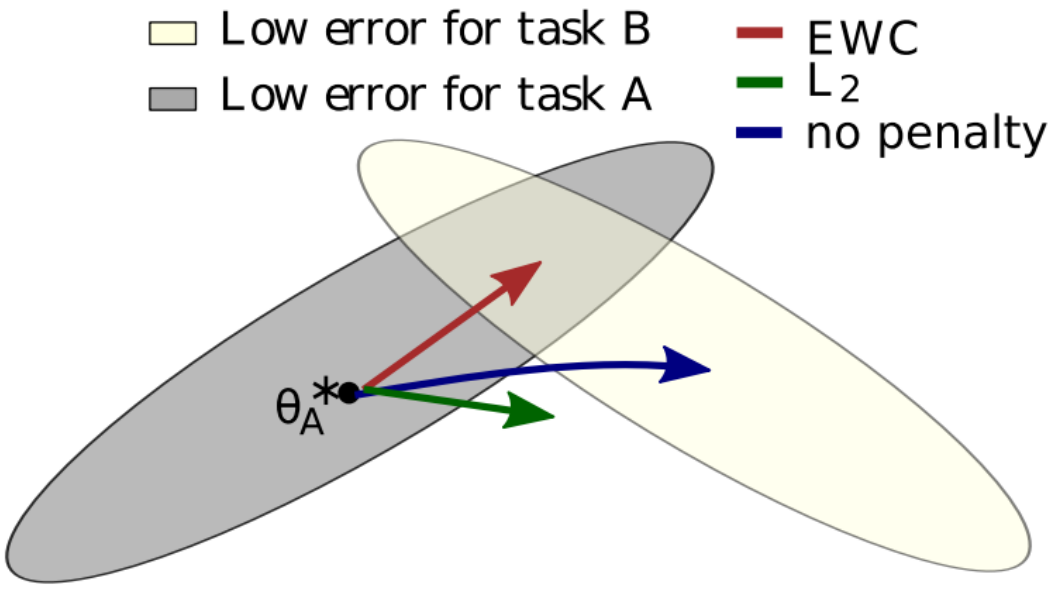
Approach

For our approach, we chose to implement a regularization strategy, specifically Elastic Weight Consolidation (EWC), and a rehearsal strategy, where previous trained task training data was fed into subsequent tasks to minimize catastrophic forgetting.

Elastic Weight Consolidation

Elastic Weight Consolidation is the process of ensuring that previously learned tasks are not forgotten while also enabling the possibility for continual learning. As documented in the research paper, **Overcoming Catastrophic forgetting in neural networks [**https://arxiv.org/abs/1612.00796], Continual learning in neural networks can be achieved by constraining important parameters to stay close to their old values.

The challenge with this method is finding and determining which constraints are the most important and determining the process for selecting those constraints. Furthermore, the combination of weights for a specific task A, B, and C, may affect the performance of the individual tasks if they are adjusted in favor towards the performance of a specific task.



Furthermore, instead of trial and error, we choose to leverage the research methods and key learnings documented in the arvix research paper [https://arxiv.org/pdf/1612.00796.pdf].

Concretely, stochastic gradient descent with dropout regularization is documented as not scalable beyond a few tasks. Therefore, EWC made more sense due to the Core50 Challenge set leveraging 50 classes, and our task of finding new classes, led us down this path.

GEM and Memory Replay (CLS Theory)

As described in "Gradient Episodic Memory for Continual Learning," Rehearsal strategies make it possible to use past examples and replayed, to current-mini batches in order to help minimize catastrophic forgetting (CF). The method that we chose to use for Replay is very similar to that of "GEM." The main feature of GEM is an episodic memory M(t), which stores a subset of the observed examples from task t. Essentially, we are taking a subset of the previously trained examples and mixing them with the current input and updating the weights according for the model to learn the new classes and to not completely forget previously learned tasks. This process does have some drawbacks. For example, incremental training datasets increase with each new training task. However, because we are using only a random subset of the previous dataset, this helps minimize the total dataset vs training the model against the full dataset.

NC - New Classes

. *Test Set-up*

For the task set-up, we leveraged a Python library called continuum [https://continuum.readthedocs.io/en/latest/index.html]. Continuum made it easy to gather the Core50 dataset and implement multiple scenarios, specifically the new classes incremental scenario. In this scenario, We want our model to train on subsequent tasks without forgetting all previous classes. The core50 is broken into 50 distinct categories. For all of our tests, we broke out the training in 9 separate tasks, each with a certain number of categories per task. The categories were broken out as follows:

1. 10 classes [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2. 5 classes [10, 11, 12, 13, 14]
3. 5 classes [15, 16, 17, 18, 19]
4. 5 classes [20, 21, 22, 23, 24]
5. 5 classes [25, 26, 27, 28, 29]
6. 5 classes [30, 31, 32, 33, 34]
7. 5 classes [35, 36, 37, 38, 39]
8. 5 classes [40, 41, 42, 43, 44]
9. 5 classes [45, 46, 47, 48, 49]

During the training process, tasks were fed one at a time and were checked for training accuracy on a separate validation set. We used continuum to split two scenarios to make this process simpler to rerun:

```
scenario = ClassIncremental(
        core50,
        increment=5,
        initial_increment=10,
        transformations=[ ToTensor(),
                         # following values come from the the mean
                        # and stddev of ImageNet - the basis of resnet.
                         Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])])
scenario_val = ClassIncremental(
        core50_val,
        increment=5,
        initial_increment=10,
        transformations=[ ToTensor(),
                         # following values come from the the mean
                        # and stddev of ImageNet - the basis of resnet.
                          Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])])
```

In addition to splitting the data into two scenarios, we also rescaled the data and applied normalization using the mean and standard deviations from the core50 set.

Resnet 18 Model

The Resnet18 has several convolutional layers and pooling layers that the core50 image set need to pass through during training. This many layers can also introduce the [vanishing gradients problem](), but can be remediated by using the RELU activation function as Resnet is below.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 64, 64]           9,408
       BatchNorm2d-2          [-1, 64, 64, 64]             128
              ReLU-3          [-1, 64, 64, 64]               0
         MaxPool2d-4          [-1, 64, 32, 32]               0
            Conv2d-5          [-1, 64, 32, 32]          36,864
       BatchNorm2d-6          [-1, 64, 32, 32]             128
....
more layers in between
....
```

```
           Conv2d-63            [-1, 512, 4, 4]       2,359,296
      BatchNorm2d-64            [-1, 512, 4, 4]           1,024
             ReLU-65            [-1, 512, 4, 4]               0
       BasicBlock-66            [-1, 512, 4, 4]               0
AdaptiveAvgPool2d-67            [-1, 512, 1, 1]               0
           Linear-68                  [-1, 50]          25,650
================================================================
Total params: 11,202,162
Trainable params: 11,202,162
Non-trainable params: 0
```

For our initial naive baseline run, we used the following parameters for a pre-trained Resnet model:

```
max_epochs = 4
lr = 0.00001
weight_decay = 0.000001
momentum = 0.9
```

We also used cross entropy as the loss function and stochastic gradient descent as the optimizer. The total running time was roughly 83 minutes using these parameters.

Resnet101 Model

The Resnet101 has additional convolutional and pooling layers as compared to Resnet18, but achieves much better training results as additional epochs are added. The trade off with a deeper neural network is the significant training time which takes roughly **240 minutes** in our environment.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 64, 64, 64]           9,408
       BatchNorm2d-2          [-1, 64, 64, 64]             128
              ReLU-3          [-1, 64, 64, 64]               0
         MaxPool2d-4          [-1, 64, 32, 32]               0
            Conv2d-5          [-1, 64, 32, 32]           4,096
....
more layers in between
```

```
....
            ReLU-338            [-1, 512, 4, 4]               0
          Conv2d-339           [-1, 2048, 4, 4]       1,048,576
     BatchNorm2d-340           [-1, 2048, 4, 4]           4,096
            ReLU-341           [-1, 2048, 4, 4]               0
      Bottleneck-342           [-1, 2048, 4, 4]               0
AdaptiveAvgPool2d-343           [-1, 2048, 1, 1]               0
          Linear-344                   [-1, 50]         102,450
================================================================
Total params: 42,602,610
Trainable params: 42,602,610
Non-trainable params: 0
```

Experiment Results

Resnet 18 Results - Naive Strategy

The performance with Resnet 18 was terrible. While the model performed decently while training categories within its respective task, it was not able to generalize beyond its trained tasks. Catastrophic forgetting was visible after the second task. In the output below, we can see how training the first task achieved a 96% accuracy but only a 65% validation accuracy for the data it was trained on. As an initial test, we trained it against all tasks to see how well it would perform and the results were extremely poor. The average accuracy over all tasks was just barely above 7%. Once the model reached task 3, the results incrementally got worse. As we can see from the results, training accuracy dipped to ~11%. Validation accuracy was ~19%.

```
cuda IS available
Namespace(batch_size=32, classifier='resnet18', convergence_criterion=0.004,
cuda=True, device='cuda:0', epochs=4, lr=1e-05, momentum=0.8, n_classes=50,
outfile='./2020_10_18-22_00_31.txt', replay=0.0, weight_decay=1e-06)


<-------------- Task 1 ---------------->
This task contains 10 unique classes
Training classes: [0 1 2 3 4 5 6 7 8 9]
...
Training accuracy: 96.34375%
Finished Training
Validating classes: [0 1 2 3 4 5 6 7 8 9]
```

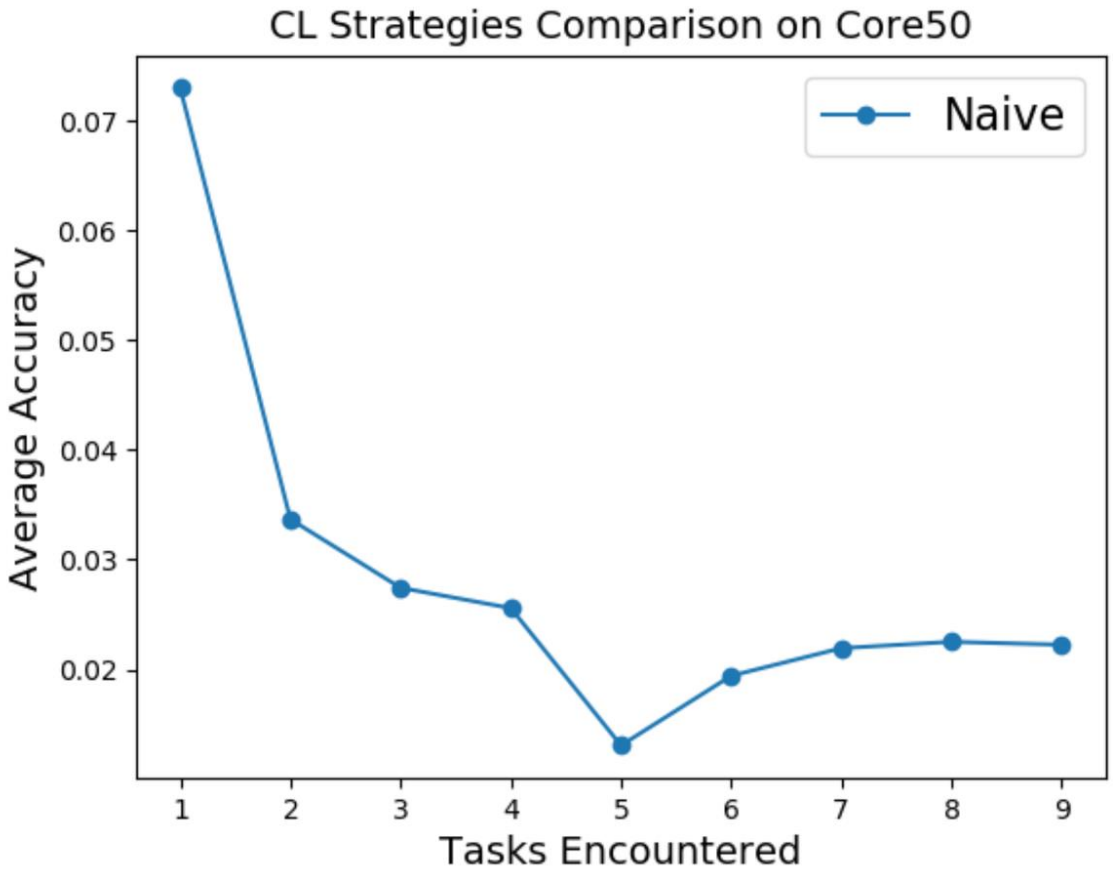
```
Validation Accuracy: 65.69205113952195%
Validating classes: [10 11 12 13 14]
Validation Accuracy: 0.0%
Validating classes: [15 16 17 18 19]
Validation Accuracy: 0.0%
Validating classes: [20 21 22 23 24]
Validation Accuracy: 0.0%
Validating classes: [25 26 27 28 29]
Validation Accuracy: 0.0%
Validating classes: [30 31 32 33 34]
Validation Accuracy: 0.0%
Validating classes: [35 36 37 38 39]
Validation Accuracy: 0.0%
Validating classes: [40 41 42 43 44]
Validation Accuracy: 0.0%
Validating classes: [45 46 47 48 49]
Validation Accuracy: 0.0%
Average Accuracy: 0.07299116793280218


<-------------- Task 3 ---------------->
This task contains 5 unique classes
Training classes: [15 16 17 18 19]
Training accuracy: 11.807343824511207%
...
Finished Training
Validating classes: [0 1 2 3 4 5 6 7 8 9]
Validation Accuracy: 0.0%
Validating classes: [10 11 12 13 14]
Validation Accuracy: 4.977777777777778%
Validating classes: [15 16 17 18 19]
Validation Accuracy: 19.710789766407117%
Validating classes: [20 21 22 23 24]
Validation Accuracy: 0.0%
Validating classes: [25 26 27 28 29]
Validation Accuracy: 0.0%
Validating classes: [30 31 32 33 34]
Validation Accuracy: 0.0%
Validating classes: [35 36 37 38 39]
Validation Accuracy: 0.0%
Validating classes: [40 41 42 43 44]
Validation Accuracy: 0.0%
Validating classes: [45 46 47 48 49]
```

```
Validation Accuracy: 0.0%
Average Accuracy: 0.027431741715760998
```

The cumulative accuracy over all 9 tasks steadily decreased using the naive method.

The very low average accuracy across each task is due to the validation uses the entire dataset and all classes, where each task only contains the respective subset of classes for that Training tasks. For future work, we would capture the Validation for the respective set of classes that the task was trained on.



Resnet 101 Results - Naive Strategy

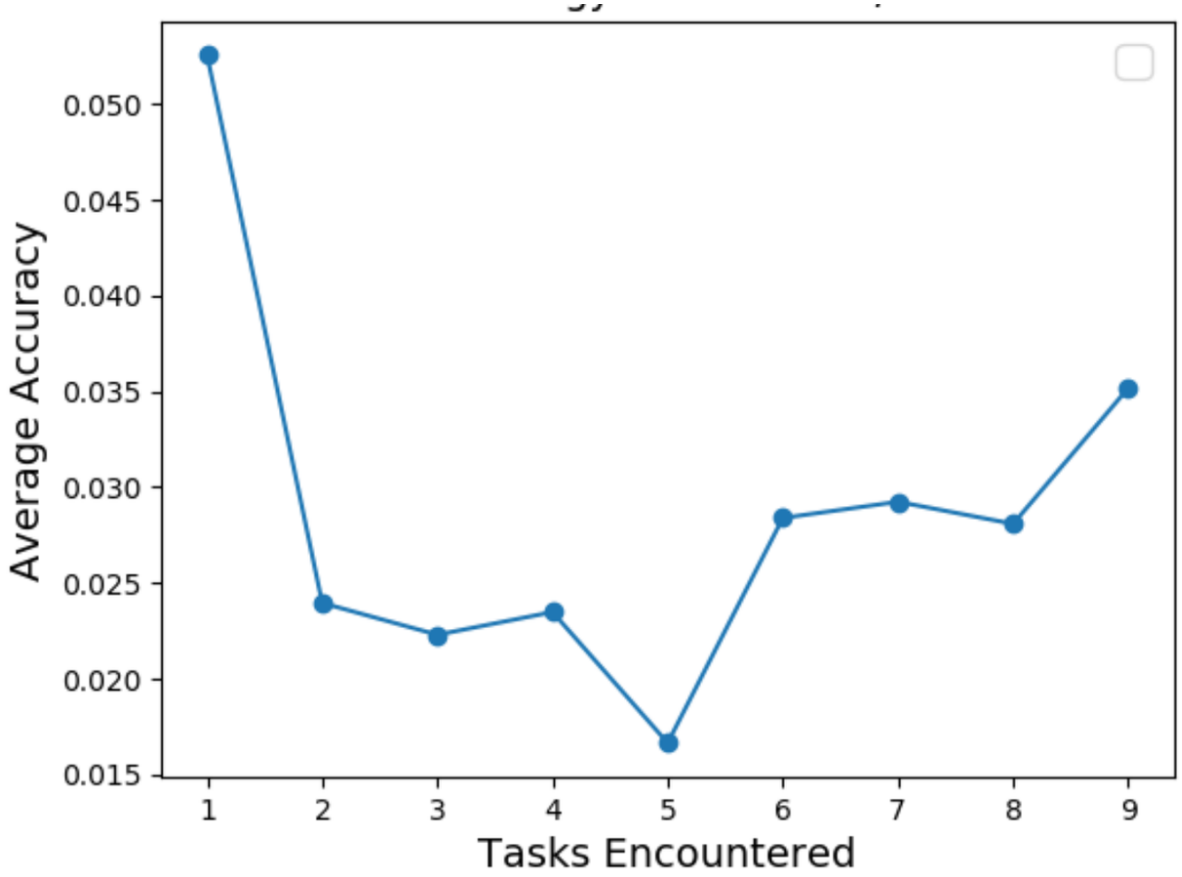
When shifting to a deeper layer network, we were able to see slightly better results during training. However, our overall results were not much different from Resnet18. Catastrophic

forgetting was clearly visible right after task 2. Any classes introduced after task 2, were poorly predicted by the model.

```
Namespace(batch_size=32, classifier='resnet101',
convergence_criterion=0.004, cuda=True, device='cuda:0', epochs=4, lr=1e-05,
momentum=0.8, n_classes=50, outfile='./2020_10_17-19_04_06.txt', replay=0.0,
weight_decay=1e-06)
<-------------- Task 1 ---------------->
This task contains 10 unique classes
Training classes: [0 1 2 3 4 5 6 7 8 9]
Training accuracy: 91.25834127740706%
Finished Training
.....
Validating classes: [0 1 2 3 4 5 6 7 8 9]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 47.29294052251251%
Validating classes: [10 11 12 13 14]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
<-------------- Task 2 ---------------->
This task contains 5 unique classes
Training classes: [10 11 12 13 14]


Validation Accuracy: 21.555555555555557%
Validating classes: [15 16 17 18 19]
```

The overall training with Resnet101 using a naive strategy was better than Resnet18 with replay. This was an unexpected result, but resnet101 does have a deeper neural network architecture than resnet18.



Replay Results

Resnet 18 with %15 Replay Results

Resnet with replay did not fare much better in our testing. The overall results were far worse than expected. We suspect that this may be due to the number of replay examples and classes that we introduced into subsequent training runs. For this test run, we used **15%** of our previous examples into subsequent tasks.

One interesting observation during our test runs is that our validation accuracy was fairly low. This could be due to overfitting or not having enough epochs. Due to the extended training time

required, we could not experiment with various batch sizes and/or epochs to determine whether overfitting and/or not enough epochs were the cause.

Again, each tasks introduces new classes as described in the **NC - New Classes above.**

. *Parameters used - all parameters were left consistent with the naïve baseline*

.

```
Namespace(batch_size=32, classifier='resnet18', convergence_criterion=0.004,
cuda=True, device='cuda:0', epochs=4, lr=1e-05, momentum=0.8, n_classes=50,
outfile='./2020_10_17-19_01_25.txt', replay=0.15, weight_decay=1e-06)
```

Training and Validation Accuracy - Task 1

```
<-------------- Task 1 ---------------->
This task contains 10 unique classes
Training classes: [0 1 2 3 4 5 6 7 8 9]
....
Training accuracy: 86.50619637750238%
Finished Training
Validating classes: [0 1 2 3 4 5 6 7 8 9]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 36.40911617565314%
```

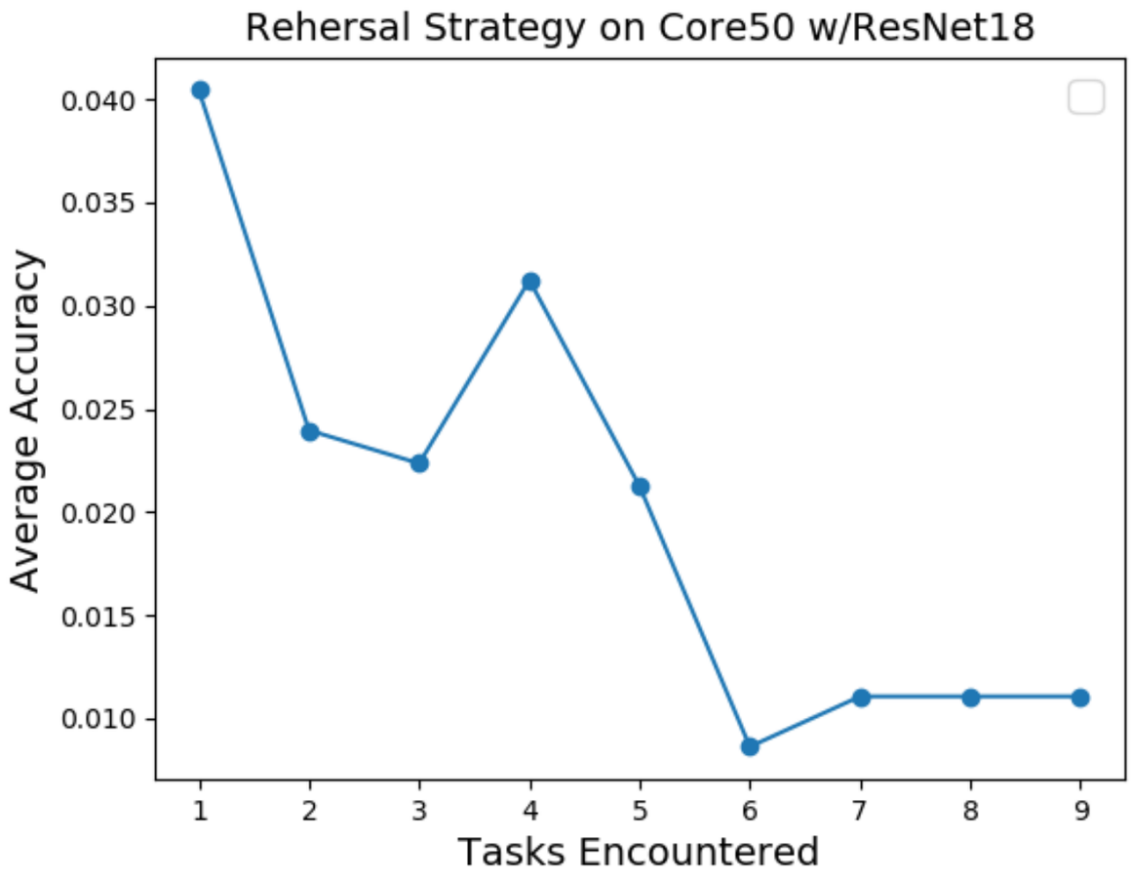
With this replay, we see overfitting as the Training accuracy is reasonable, yet validation results are significantly low - note that the Training run was only four epochs

Training and Validation Accuracy - Task 2

During task 2, we saw drastic drop in training and validation accuracy even though we had samples from task 1 included. Normally, we would have expected better results as we've seen in test examples of the MNIST class. The results were progressively worse as more tasks were trained. This is an area that we will need to investigate in the future.

```
<-------------- Task 2 ---------------->
This task contains 15 unique classes
Training classes: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
...
Training accuracy: 15.640818722030666%
Finished Training
Validating classes: [0 1 2 3 4 5 6 7 8 9]
Classes predicted: [ 1.  7.  9. 10. 12. 13. 14.]
Validation Accuracy: 0.011117287381878822%
Validating classes: [10 11 12 13 14]
Classes predicted: [10. 12. 13.]
Validation Accuracy: 21.555555555555557%
```

Resnet 18 with 15% Replay



Resnet101 with 15% Replay Results

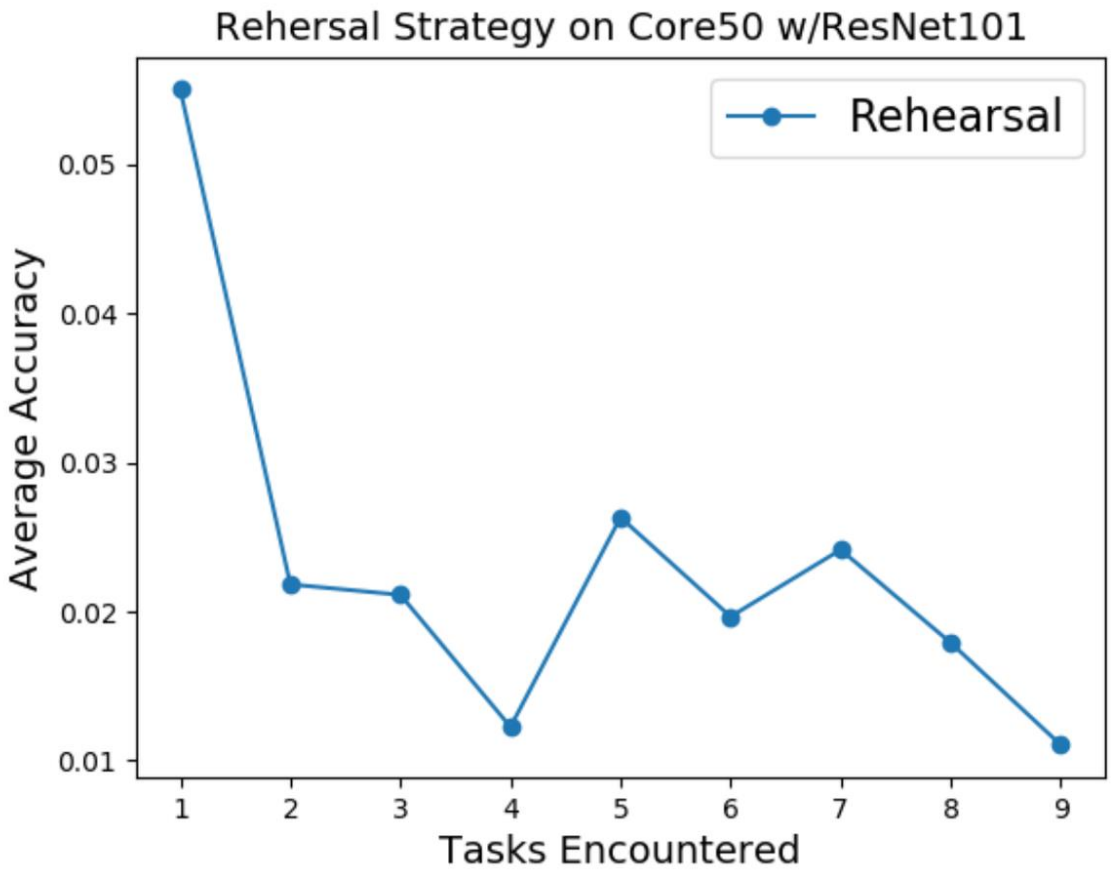
The additional layers with Resnet101 helped increase the overall accuracy compared to resnet18, however, the overall results were still not much better than resnet18. Note, the accuracy is low due to performing validation against unseen classes.

```
cuda IS available
Namespace(batch_size=32, classifier='resnet101',
convergence_criterion=0.004, cuda=True, device='cuda:0', epochs=4, lr=1e-05,
momentum=0.8, n_classes=50, outfile='./2020_10_18-14_08_47.txt',
replay=0.15, weight_decay=1e-06)


<-------------- Task 1 ---------------->
This task contains 10 unique classes
Training classes: [0 1 2 3 4 5 6 7 8 9]
<------ Epoch 1 ------->
[Mini-batch 100] avg loss: 0.08016
...
<------ Epoch 4 ------->
...
Training accuracy: 89.2469018112488%
Finished Training
Validating classes: [0 1 2 3 4 5 6 7 8 9]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 49.52751528627015%
Validating classes: [10 11 12 13 14]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [15 16 17 18 19]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [20 21 22 23 24]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [25 26 27 28 29]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [30 31 32 33 34]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
```

```
Validating classes: [35 36 37 38 39]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [40 41 42 43 44]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Validating classes: [45 46 47 48 49]
Classes predicted: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Validation Accuracy: 0.0%
Average Accuracy: 0.055030572540300166
Average Accuracy: 5.503057254030017%
....
Addtional output for readability
....
<-------------- Task 8 ---------------->
This task contains 45 unique classes
Training classes: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44]


Average Accuracy: 1.105536546183596%
--- 25029.19126224518 seconds ---
```



Code

All source code for the project is located in GitHub:

https://github.com/cicorias/njit-dl-project1

Running Project

The project Readme : https://github.com/cicorias/njit-dl-project1/blob/master/readme.md contains the setup and running of the Python script.

Conclusion

Based on our testing, we did not see a significant improvement when running Naïve vs Replay for the Core50 dataset. We suspect that our model most likely needs additional epochs and better parameter tuning. Our reliance on Resnet and transfer learning did not provide the initial anticipated accuracy and recall rate.

Given the challenges the team feels it would have taken the MNIST dataset route and challenge first, even narrowing down the dataset to just enough data to Train and Validate a model. The limited data would make it far easier to track down computational logic and implementation issues and confirm algorithms as well. The time for training is also of a great concern. Even though the team had GPU enabled cloud virtualized instances of Linux, from the PyTorch documentation there are significant gains that could also be done to greatly improve GPU utilization.

References

1. https://arxiv.org/abs/1910.02718
2. https://arxiv.org/pdf/2007.02443.pdf
3. https://arxiv.org/pdf/1910.02509.pdf
4. **Gradient Episodic Memory for Continual Learning -** David Lopez-Paz, Marc'Aurelio Ranzatohttp://papers.nips.cc/paper/7225-gradient-episodic-memory-for-continual-learning.pdf