New Jersey Institute of Technology
CS677 1J1 (Deep Learning)
Project 1

CVPR20 CLVision Continual Learning

Team Members:

- Sean Walsh (ucid: sw522)
- Jonathan Vidal (ucid: jkv5)
- Nayem Paiker (ucid: nrp66)

CORe50 dataset: It is an object recognition dataset designed for different CL scenarios. This dataset contains images of 50 domestic objects and a total of 10 categories and a total of 164,866 RGB-D images with 128 X 128 pixels.

Codebase: Our project's code was forked from the winning codebase found here https://github.com/RaptorMai/CVPR20_CLVision_challenge. We made updates to run some class activation maps utilizing torchcam.

Method: The method used in the project is called Batch-level Experience replay with Review, which is based on a model called Experience Replay (ER) and in the end a review step is added to avoid unexpected errors.

In this method, the ER stores a subset of samples from past batches in a memory buffer, which is of limited size. During the training, it concatenates the incoming mini-batch with another mini-batch of data retrieved from the memory buffer. Then using the combined batch, it takes a Stochastic Gradient Descent step and then updates the memory afterwards.

Except for the first batch, the algorithm performs a batch level experience replay. For every epoch, the data pulled from the memory buffer is different. By the time we finish the last epoch of the current batch, we will select examples of size (memory size / n), where n is the total number of batches from the whole current scenario.

After the training is finished for all batches, this method also contains a review step. In this step, it draws a batch from the memory and performs stochastic gradient descent again. In order to avoid overfitting, the learning rate in the review step is lower than the learning rate used to process the new incoming batches.

The pseudocode of the model is given below:

procedure(D, mem_size, replay_sz, review_sz, batch_sz, lr_replay, lr_review**)**

```
    # allocate memory of size mem_size
    M ← {} * mem_size
    For t ∈ {1….T} do
        For epochs do
            If t > 1 then
                # sample a batch of data with size equivalent to replay_sz from M
                D_M ~ M (equivalent to replay size)

                # concatenate the current data batch, D_t with the memory data batch, D_M
                D_train = D_M U D_t
            Else
                D_train = D_t
```

one pass minibatch gradient descent over D_{train}
$\theta \leftarrow$ SGD (D_{train}, theta, lr_replay, batch_sz)

memory update
M $\leftarrow$ UpdateMemory(D_t, mem_sz)

drawing a batch of data of size equivalent to review_sz from M
D_R ~ M (equivalent to review size)
one pass minibatch gradient descent over D_R
$\theta \leftarrow$ SGD(D_R DR, theta, lr_review, batch_sz)

Return θ

Approach: This project with CORe50 database consists of three different scenarios. Those are:

- **New Instances (NI)**: In this NI scenario of this codebase, there are 8 batches, which are presented sequentially. Each of these batches contains the same 50 classes. In both, training and testing environment, no batch label is given.

- **New Instances and Classes (NIC):** In this scenario, a total of 391 training batches are presented. Each of the training batches contains 300 images of a single class. Just like the NI scenario, no batch label is provided in this scenario either. In order to avoid catastrophic forgetting by fine-tuning the model, the NIC scenario uses the same method described earlier (Batch-level Experience Replay with Review).

- **Multi-Class New Classes (Multi-Task-NC):** In this scenario, a total of 9 different batches are used. These 9 batches contain a total of 50 classes. The first batch contains 10 classes and each of the other 8 batches contains 5 classes. In this scenario, task label will be provided during the training and test environment, and because of this, the task difficulty is a lot smaller than the previous 2 scenarios.

Architecture and preprocessing: For all three scenarios, this project uses the DenseNet-161 model, which is pre-trained on ImageNet. DenseNet-161 is the largest model in DenseNet group with size of almost around 100MB.

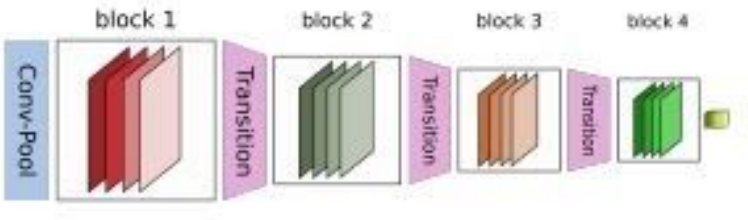

Figure 1: DenseNet-161 architecture

It consists of a total of 4 dense blocks. A DenseNet consists of a stack of 4 dense blocks followed by transition layers. Each block consists of a series of units. Each unit packs two convolutions, each preceded by Batch Normalization and ReLU activations. This experiment freezes all the layers before the third blocks to ensure that the pre-trained model can extract the basic features from the image and shorten the training.

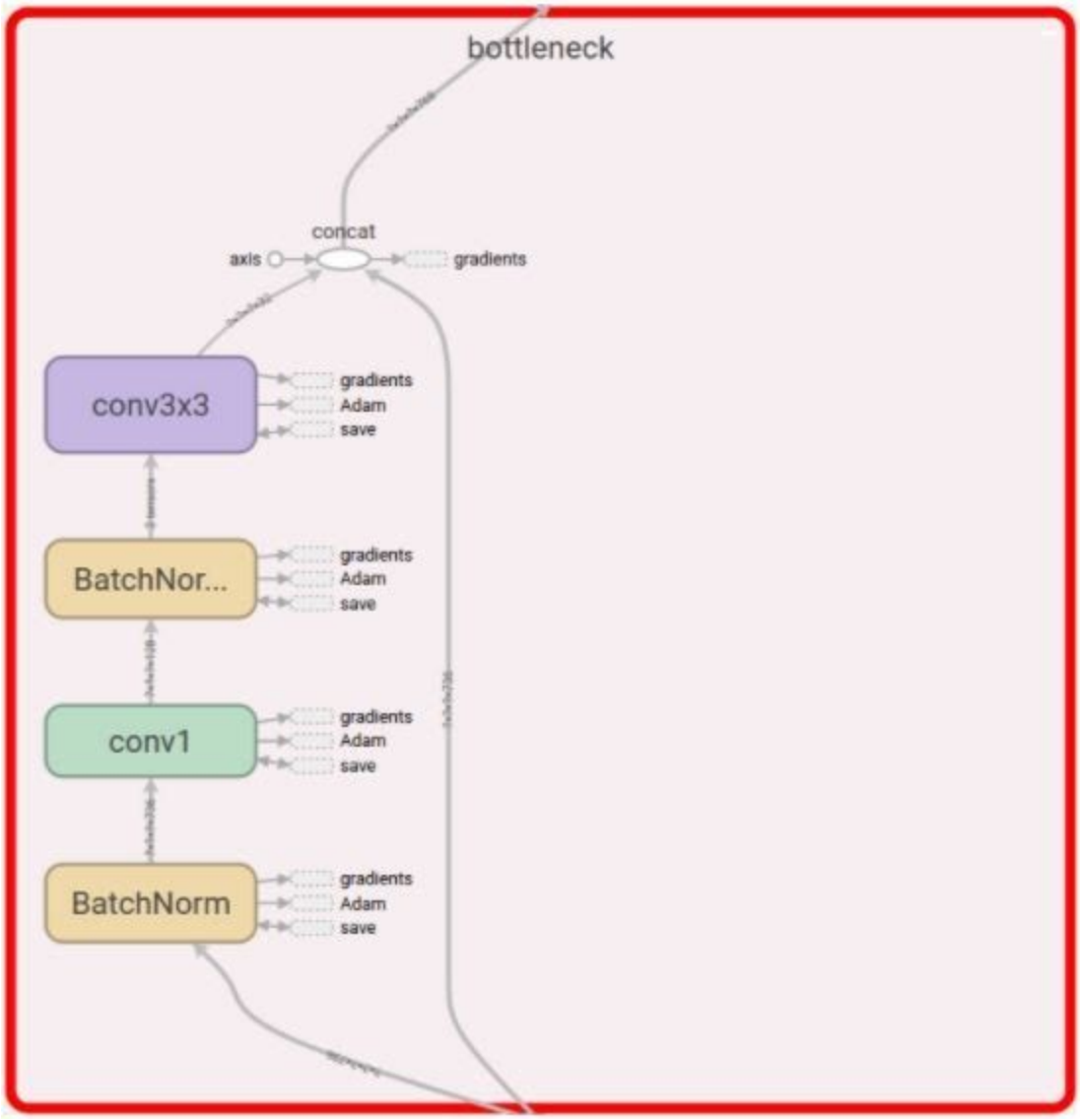

Figure 2: DenseNet block unit operations

On the other hand, all of the transition layer contains a Batch Normalization layer, which is followed by a 1x1 convolution, then followed by a 2x2 average pooling.

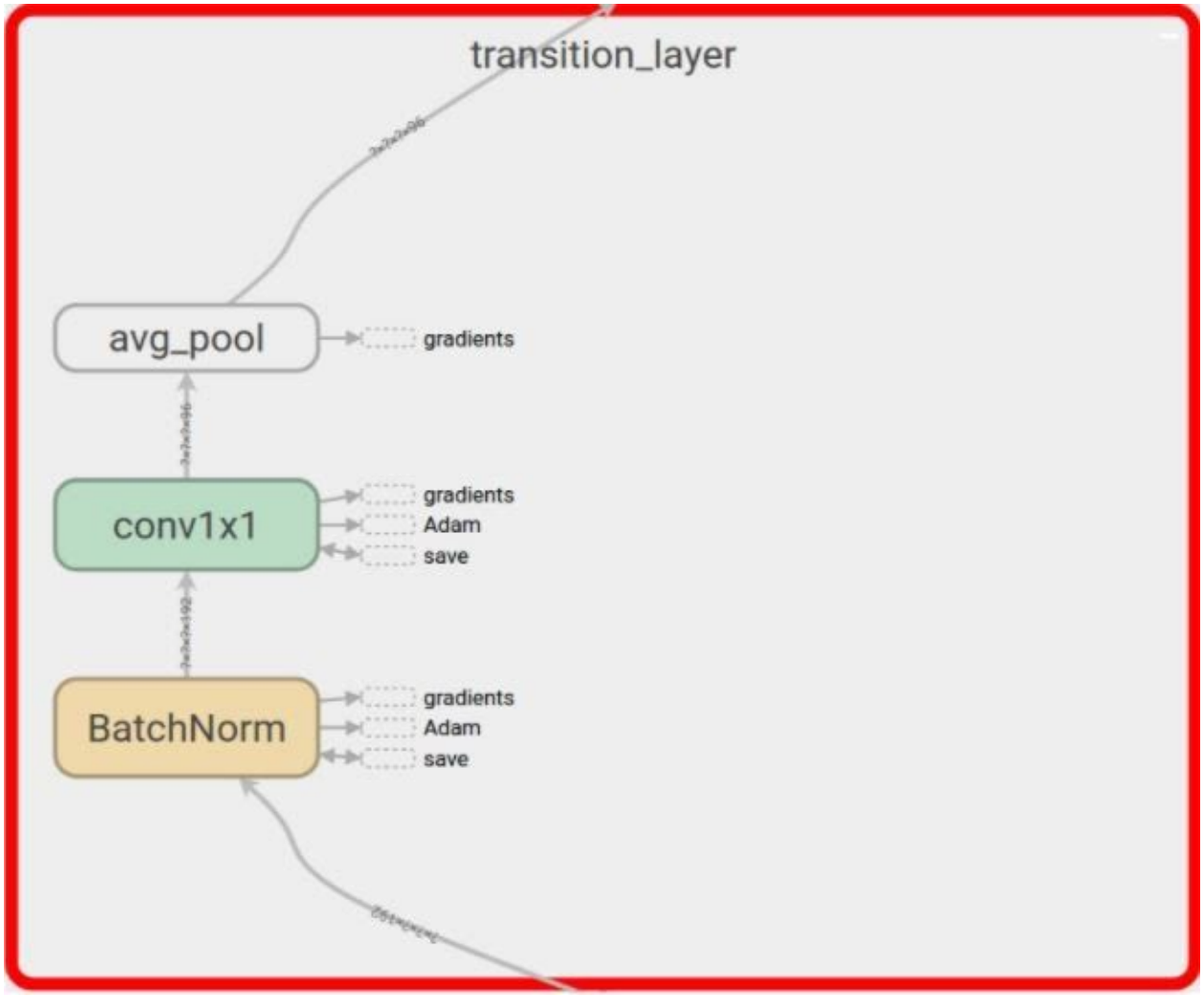

Figure 3: DenseNet transition layer

The data needs to be preprocessed before feeding into the models. In this process, every image is center-cropped with size (100, 100) and resized to (224, 224, 3) from (128, 128, 3). In this model, this is an important step since most of the target object is in the center of the image and center-cropping helps to mitigate backgrounds effect and lighting. As we do not train any layers before the third dense block, we resize the cropped image to the size of (224, 224, 3) to ensure no size discrepancy between the pre-trained model and the training images. Noted that this preprocessing step is applied to both training and testing images.

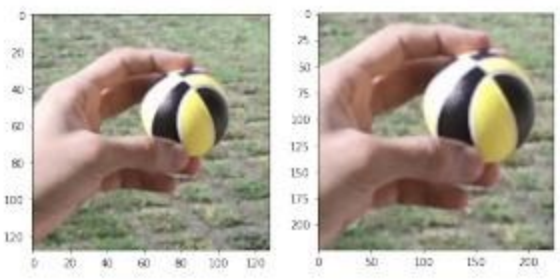

Figure 4: Example of Center-Cropping

Also, in order to get a better generalization of the data, pixel-level and spatial-level transformations are also used. For pixel level, RandomContrast, RandomGamma, and RandomBrightness and for spatial-level, HorizontalFlip, RandomRotate90, ElasticTransform, GridDistortion, and OpticalDistortion.

In the Independent Model, each batch will receive the pre-trained models from the previous batch. “The Baseline method in Multi-Task-NC shares all the layers of the model before the last fully-connected layer between all the batches and each batch has its own fully connected layer.” https://arxiv.org/pdf/2007.05683.pdf The baseline model experiences a large amount of forgetting compared to the independent model. This model is used for Multi-Task New Classes scenario. Unlike Batch-level Experience Replay with Review, this model will not hold subsets of samples from the previous batches. This model has less difficulty on tasks compared to NI and NIC scenarios. Also, there are no steps taken to avoid forgetting as it does not have a final review that reminds the model of what it has learned already.

This model resulted in 98.6 percent final accuracy with DenseNet freeze. However, with the preprocessing steps, the final accuracy is 99.3 percent. The preprocessing steps will modify the images specs where it centered, cropped and resized the object. In addition to preprocessing steps, images are manipulated for better generalization. The pixel of the images can be transformed using random contrast, gamma and brightness. Also, the orientation of the images is altered such as the horizontal flip, random rotation of 90 degree, and grid and optical distortion.

Inference Phase

Transparency in deep learning continues to be an area of importance. In order to gain acceptance in certain fields the logical inference of the models need to be interpretable by the humans working in those domains. Using visualization to demonstrate what feature sets the models focused on when making a decision has been a powerful tool in gaining the trust of users. For Convolutional Neural Network (CNNs), three of the popular methods are Gradient visualization, Perturbation, and Class Activation Map (CAM).

"CAM-Based explanation provides visual explanation for a single input with a linear weighted combination of activation maps from convolutional layers. CAM creates localized visual explanations but is architecture sensitive, a global pooling layer is required to follow the convolutional layer of interest." https://arxiv.org/pdf/1910.01279.pdf For this project we leveraged torchcam which allows you to leverage the class-specific activation of convolutional layers in PyTorch. Torchcam has several CAM methodologies available so we thought the best way to see which method was most applicable to our model was to look at them side by side.

For this visualization comparison we looked at a single object (sun glasses) at two different angles and through multiple classifiers.

Figure 5: Example 1, Upside-down sunglasses using ResNet18

Example 2: Upright sunglasses using ResNet18

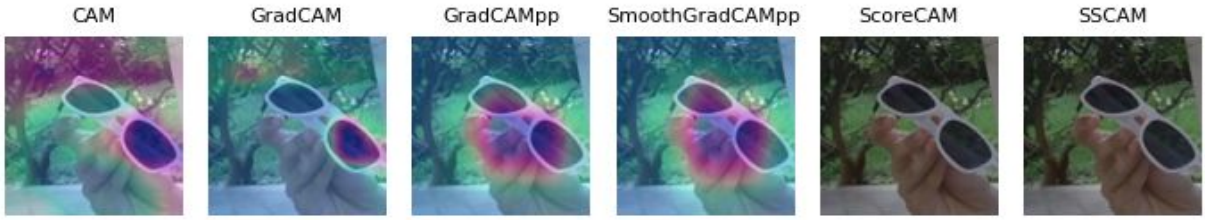

Figure 6: Example 2, Upright sunglasses using ResNet18

It seems in this example that the model does a better overall job at detecting the important features of the upright sunglasses. The averaging technique used in the SmoothGradCAMpp method seems to compensate for the rotation of the image.

Example 3: Upside-down sunglasses using DenseNet201

Figure 7: Example 3, Upside-down sunglasses using DenseNet201

Example 4: Upright sunglasses using DenseNet201

Figure 8: Example 4, Upright sunglasses using DenseNet201

Once again the upright glasses are easier to focus on but only the first two methods produce meaningful results. DenseNet seems to perform a little better on the upside down sunglasses, indicating it may do a better job than ResNet with the rotated version of the objects.

Experiments:

	NI	NIC	Multi-Task NC
avg valid_acc	0.905124499777679	0.5878713926848989	0.55045267489712
Time elapsed	245.92726756334304	497.3835241874059	67.85061559677
avg ram_usage	11142.463623046875	6403.070362452046	13504.37217882
max ram_usage	11353.8125	6872.5	15817.3125

parameters	parameters: scenario: ni sub_dir: ni cls: dense_freeze_till3 method: task_mem n_classes: 50 #Augmentation aug: True aug_type: center_224 #replay replay_examples: 10000 replay_used: 10000 replay_epochs: 2 #review review_lr_factor: 0.5 review_size: 20000 review_epoch: 1 #train batch_size: 32 epochs: 1 optimizer: SGD nesterov: True momentum: 0 weight_decay: 0.01 lr: 0.007 #misc verbose: False preload_data: False use_cuda: True	parameters: scenario: nic sub_dir: nic cls: dense_freeze_till3 method: task_mem n_classes: 50 #Augmentation aug: True aug_type: center_224 #replay replay_examples: 200 replay_used: 600 replay_epochs: 1 #review review_lr_factor: 1 review_size: 20000 review_epoch: 1 #train batch_size: 32 epochs: 1 optimizer: SGD nesterov: True momentum: 0 weight_decay: 0.001 lr: 0.01 #misc preload_data: False use_cuda: True	parameters: scenario: nic sub_dir: nic cls: dense_freeze_till3 method: task_mem n_classes: 50 #Augmentation aug: True aug_type: center_224 #replay replay_examples: 200 replay_used: 600 replay_epochs: 1 #review review_lr_factor: 1 review_size: 20000 review_epoch: 1 #train batch_size: 32 epochs: 1 optimizer: SGD nesterov: True momentum: 0 weight_decay: 0.001 lr: 0.01 #misc preload_data: False use_cuda: True

Conclusion: In summary, the Batch-Level Experience Replay with Review retrieves samples from memory when it receives data from a new batch and updates the memory after training the current batch. And it also adds a review step before the final testing to avoid overfitting issues.