CAPSTONE PROJECT REPORT

AWS Machine Learning Engineer Nanodegree program

I. OVERVIEW

1.1. Problem statement

Dementia is an umbrella term for loss of memory and other thinking abilities severe enough to interfere with daily life. Dementia is known as old people's disease. However, in the recent years, the younger people who are in their 40s get diagnosed with dementia or mild cognitive disorder and the number is increasing. The most common dementia type is Alzheimer's disease which constitutes 60-80% of all dementia cases.

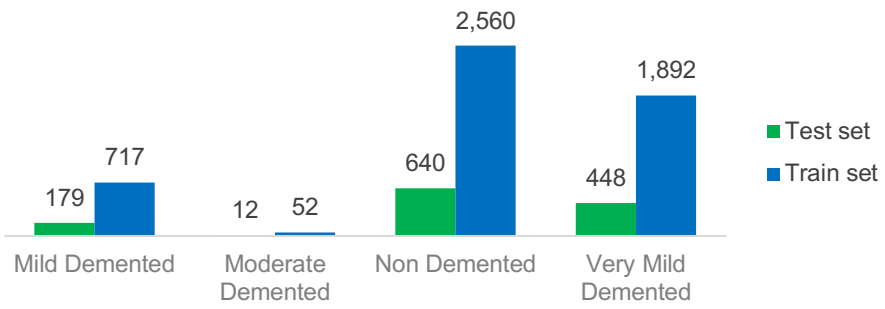
There are many factors that contribute to dementia throughout the lifetime of a person. The diagnosis of dementia is typically made by a physician or neurologist based on considering genetic components, multiple lab tests, neurological exams such as brain MRI or CAT scans, and psychiatric assessment of mental health & cognitive skills.

Alzheimer's is the disease that majority of dementia patients get diagnosed therefore, the goal of this project is to build AI models to predict severity of Alzheimer's disease based on MRI images.

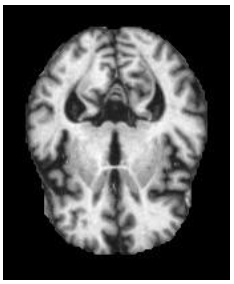
1.2. Dataset & Data Preparation

Labeled brain MRI image dataset that is available on Kaggle platform which was downloaded to be used in this project. The dataset consists of train and test images, total of 6,400 images. The images were labeled as one of the following severity levels and the number of images in each dataset is shown below.

Visual 1. Number of images in each category



Each category is given as a folder with the corresponding images. The dataset is given as **.jpg** format. The train set constitutes 81.6% of the dataset. Each brain image is given as a black and white 2D image with size of 208x176.



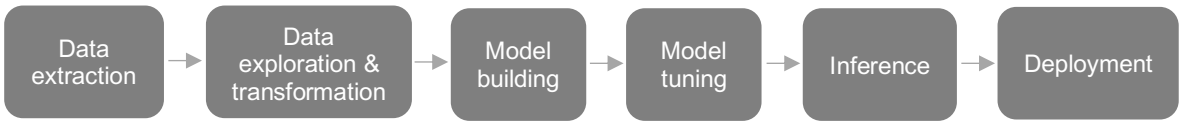
Since the dataset doesn't contain validation set, 20% of the training data was randomly chosen to create validation set. The train, test and validation data were uploaded to the designated S3 bucket which was used for model training.

II. SOLUTION PLANING & DESIGN

The project includes the element of following process.

1. Use Sagemaker script mode
2. Tune the hyperparameters
3. Use debugger and profiler
4. Use multi-instance training if model training job exceed 1 hour GPU training
5. Create endpoint for inference and test the models

The following is the project design diagram.



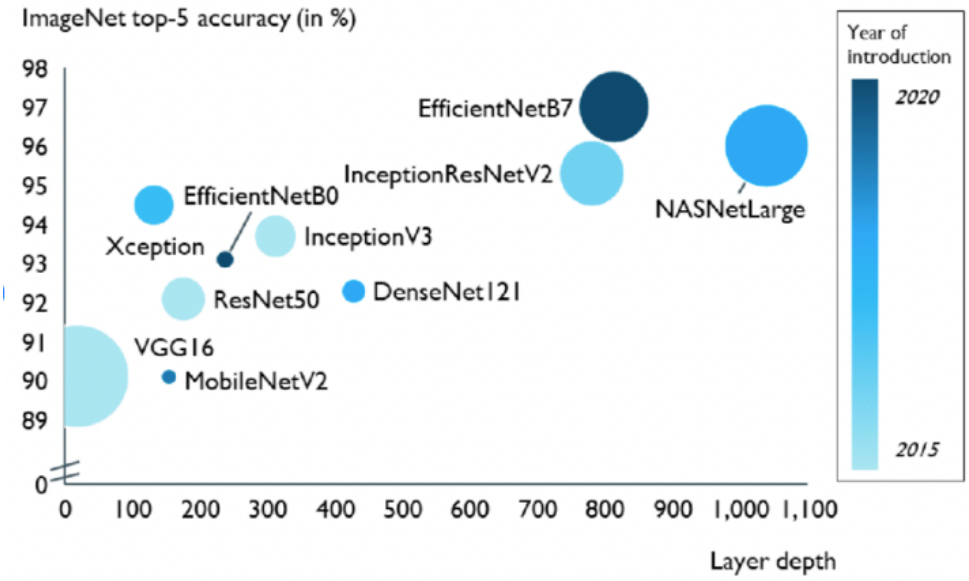
III. EXECUTION AND RESULTS

The goal of the project is to build model that predicts the Alzheimer's disease severity from MRI images. Because it's an image classification task, we needed neural network models. Training neural model is computationally expensive, therefore, I started with pre-trained model.

3.1. Pre-trained model tuning approach

There are many models developed over the years by researchers and industry experts. However, not every model is the same. Older models such as VGG and ResNet are larger (i.e. high number of parameters) and require more resources to train. On the other hand, the model that are developed in the recent years have smaller number of parameters and faster to train.

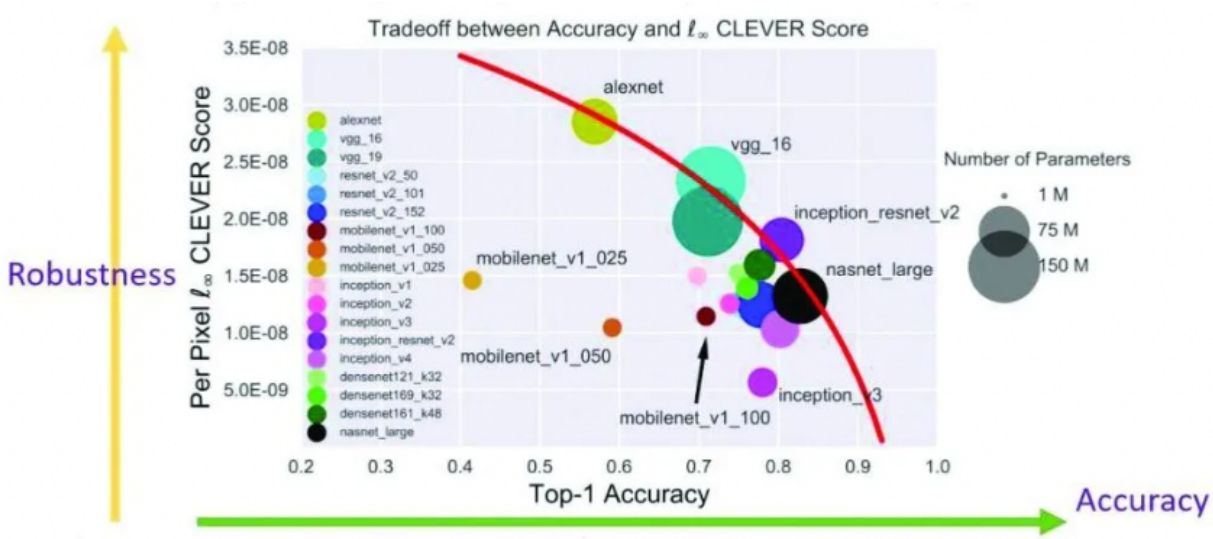
Visual 2. Model accuracy, size and developed time period[1].



Although high accuracy is important especially in medical machine learning applications, the model needs to robust i.e. less sensitive to new data to be reliable. The model robustness[2] refers to the degree that a model's performance changes when using new data versus training data.

[1] https://www.researchgate.net/figure/Comparison-of-Convolutional-Neural-Network-Architectures-in-Terms-of-Size-and-Performance_fig3_363738835
[2] https://vectorinstitute.ai/machine-learning-robustness-new-challenges-and-approaches/#:~:text=What%20is%20model%20robustness%3F,tool%20depends%20on%20reliable%20performance.

Visual 3. Model accuracy, robustness and size[3].



Based on accuracy and robustness trade-off, I decided to use robust model VGG16.

3.2. VGG16 tuning

To speed up the training process, the distributed training was utilized, and the following instance was used.

```
instance_count=2,
instance_type="ml.m5.2xlarge",
```

VGG16's classifier[0] layer was replaced to work with our dataset.

```
nn.Sequential(
    nn.Linear(num_features, 2048),
    nn.ReLU(inplace=True),
    nn.Linear(2048, 516),
    nn.ReLU(inplace=True),
    nn.Linear(516, 4)
)
```

The data was loaded and resized to (200x200) size and normalized and the train and valid/test sets were transformed as follows.

```
transform_train = transforms.Compose([
                                transforms.Resize(200),
                                transforms.RandomHorizontalFlip(),
                                transforms.ToTensor(),
                                transforms.Normalize((0.497,0.402,0.
                                425), (0.308, 0.325, 0.301))])
```

[3] https://www.eetimes.eu/ai-tradeoff-accuracy-or-robustness/

```
transform_valid = transforms.Compose([
                                transforms.Resize(200),
                                transforms.ToTensor(),
                                transforms.Normalize((0.497,0.402,0.
                                425), (0.308, 0.325, 0.301))])
```

The following 2 hyperparameters were tuned over 30 epochs.

```
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.001, 0.1),
    "batch-size": CategoricalParameter([32, 64]),
}
```

3.4. VGG16 model training & evaluation

The best estimator returned following hyperparameter configuration.

```
{"batch-size":32, "lr":"0.002385083227823211"}
```

The hyperparameters were used to train the model over 30 epochs and deployed to the endpoint to make prediction over the test set. The confusion matrix of the test data is shown below.

Table 1. VGG16 Confusion matrix

		Predicted Severity			
		Mild	Moderate	Non	Very Mild
Actual Severity	Mild	0	0	19	160
	Moderate	0	0	2	10
	Non	0	0	427	213
	Very Mild	0	0	139	309

The model accuracy on test data is 57.5%. If we look at the confusion matrix, The most of mild and moderate cases were predicted as Non or Very Mild.

The accuracy can be improved by increasing the number of epochs and applying other image transformations.

IV. ALTERNATE APPROACH

4.1. Custom CNN model structure

Based on the outcome of VGG16, I decided to create my own CNN model and train to see the result. I created Convolutional and Dense layer blocks (i.e. functions) that can be called and included in the model instead of writing large CNN model. Each block includes CNN/ Dense layers, activation function, batch normalization and max pooling layers.

The Convolutional NN block is shown below:

```
def conv_block(input_size, kernel_size):

    block = nn.Sequential(
         nn.Conv2d(input_size, input_size+96,
                   kernel_size=kernel_size, stride=1, padding=1),
        nn.ReLU(),
        nn.Conv2d(input_size+96, input_size+64,
                  kernel_size=kernel_size, stride=1, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(input_size+64),
        nn.MaxPool2d(2, 2)
    )

    return block
```

The Dense NN block is shown below:

```
def dense_block(input_size, output_unit, dropout_rate):

    block = nn.Sequential(
        nn.Linear(input_size, output_unit),
        nn.ReLU(),
        nn.BatchNorm1d(output_unit),
        nn.Dropout(dropout_rate)
    )

    return block
```

The final model structure:

```
def net(num_classes, input_size, kernel_size, dropout_rate):

    model = nn.Sequential(
          nn.Conv2d(input_size,128,kernel_size=kernel_size,padding='same')
          nn.ReLU(),
          nn.Conv2d(128, 64, kernel_size=kernel_size, padding='same'),
          nn.ReLU(),
          nn.MaxPool2d(2, 2),
          conv_block(input_size=64, kernel_size=kernel_size),
          conv_block(input_size=128, kernel_size=kernel_size),
          conv_block(input_size=192, kernel_size=kernel_size),
          nn.Dropout(dropout_rate),
          conv_block(input_size=256, kernel_size=kernel_size),
          nn.Dropout(dropout_rate),
          nn.Flatten(),
          dense_block(input_size=8000, output_unit=512,dropout_rate=0.7),
          dense_block(input_size=512, output_unit=128, dropout_rate=0.5),
          dense_block(input_size=128, output_unit=64, dropout_rate=0.3),
          nn.Linear(64, num_classes))

    return model
```

4.2. Data transformation

There are some transformations that I decided not to use. For example, `RandomHorizontalFlip` which flips the image randomly along the horizontal axis. The brain has 2 hemispheres which, depending on the disease, cannot be flipped. In this case, applying random flip would cause confusion in the model. Moreover, the image is 2D black and white image which sometimes can appear too dark to observe any abnormality even for a human radiologist. Therefore, `ColorJitter` transformation added to adjust the color composition of the images. Also, the image size was reduced to 176.

```
transform_train = transforms.Compose([
          transforms.Resize(176),
          transforms.ColorJitter(brightness=0.5, contrast=1,
                                 saturation=0.1, hue=0.5),
          transforms.ToTensor(),
          transforms.Normalize((0.497,0.402,0.425), (0.308, 0.325,
          0.301))])
```

4.3. Model training & evaluation

To speed up the training process, the distributed training was utilized, and the following instance was used.

```
instance_count=2,
instance_type="ml.m5.2xlarge",
```

The following hyperparemers were used train the model.

```
{'batch-size': 120, 'lr': '0.0011', 'epochs':'100'}
```

The model was trained successfully and deployed to endpoint to predict the test data. The confusion matrix of the test data is shown below.

Table 2. Custom CNN Confusion matrix

		Predicted Severity			
		Mild	Moderate	Non	Very Mild
Actual Severity	Mild	83	0	25	71
	Moderate	2	3	2	5
	Non	20	1	426	193
	Very Mild	29	3	101	315

Although training and validation accuracy reached 80%, the test was 64.66% which is 8% improvement from VGG16 model. If we look at the confusion matrix, the model predicts Mild severity slightly better than VGG16 model. As for Non and Very Mild, the model started to make slight errors compared to VGG16. Overall, custom CNN model outperformed VGG16.

V. BENCHMARK COMPARISON

The objective of the models was to minimize validation `CrossEntropyLoss`.

$$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

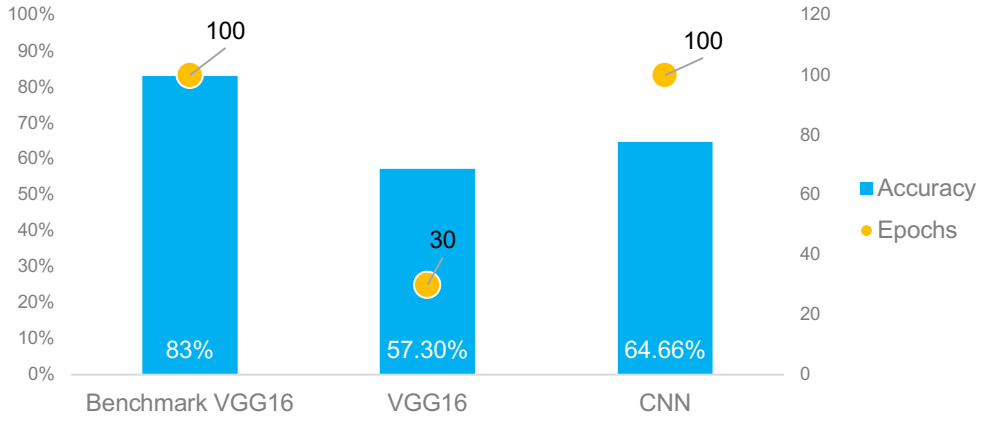
`CrossEntropyLoss` measures the difference between the discovered probability distribution of a machine learning classification model and the predicted distribution[4].

The model comparisons made based on accuracy. Accuracy is calculated as:

$$\text{Accuracy} = \frac{(TP + TN)}{(TP + FP + TN + FN)}$$

[4] https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmlldzoxMDA5NTMx

Visual 4. Benchmark accuracy and epoch comparison



The benchmark model was VGG16 which scored 83% accuracy and trained over 100 epochs. Our VGG16's accuracy was 57.3% and CNN's accuracy was 64.66% which are lower than the benchmark model's. The reasons that our models have lower accuracy can be explained:

- VGG16 model was trained for only 30 epochs. Due to the size of the model, it looks like the model needs to be trained longer epochs to tune and yield good accuracy.
- Although CNN model was trained over 100 epochs, the model structure is different from VGG16. It is possible that the model required more epochs.
- CNN model was not tuned in this project. It's possible that hyperparameters were not optimized, therefore, yielded less accurate result.
- Image transformations of the benchmark VGG16 differed from our transformations. The benchmark model's transformations include horizontal and vertical flips, rotation and zoom. Our transformation may have been slightly conservative. Due to the nature of the disease, we made an assumption that the brain's left and right hemispheres needed to be distinguishable and cannot be flipped.

VI. CONCLUSION

In this project, I used pre-trained VGG16 model and built custom CNN model to predict severity of Alzheimer's disease from brain x-ray images. The following tools and concepts were utilized throughout the project.

- The data was downloaded from Kaggle platform, split, and uploaded to S3 bucket for further use.
- The models were built in python scripts then called to train and predict. The script mode was utilized in both model training instances.
- Due to the size of the models, distributed training i.e. 2 instances were used.
- Debugging and profiling were setup to conduct debugging during and after training (specific rules and hook configurations can be found in the code notebook)
- Trained models were deployed using endpoint to make inference on test data.
- Compute instance types were carefully chosen to not to incur unnecessary cost.
- Endpoint deployment was deleted after inference or decided unnecessary.

Although the project followed the plan accordingly and successfully demonstrated the skillset learned throughout the course, due to the time and financial constraints, certain modules and concepts were not utilized. The model results and project can be further improved by:

- Use more powerful resources to speed up the training and inference process and time. Although it would increase cost, it will allow us to test more models, large number of epochs and obtain higher accuracy.
- Because the project was not going live, I didn't consider or setup latency and high volume throughput which can be handled by setting up autoscaling and provision concurency.
- In addition to the second point, I didn't consider security issues. Patient's medical information is very sensitive in nature and require high level security. To assure secure machine learning application, we can setup virtual private cloud and use the EC2 instance to train the model. In this case, model training code script will be slightly different from what I used in script mode.

- • • • -