ALZHEIMER'S DISEASE DIAGNOSTICS BY ADAPTATION OF CONVOLUTIONAL NETWORK

Prepared for: Hossein Haji Abolhasan, Professor at Shahid Beheshti University
Prepared by: Tara Rasti, Data Science Student
Supervisor: Esmail Mafakheri
Fall 2020

Alzheimer Disease

Alzheimer's disease (AD) is a progressive brain disorder and the most common case of dementia in the late life. AD leads to the death of nerve cells and tissue loss throughout the brain, thus reducing the brain volume in size dramatically through time and affecting most of its functions. The estimated number of affected people will double for the next two decades, so that one out of 85 persons will have the AD by 2050. An estimated 5.5 million people aged 65 and older are living with AD, and AD is the sixth-leading cause of death in the United States. The global cost of managing AD, including medical, social welfare, and salary loss to the patients' families, was $277 billion in 2018 in the United States, heavily impacting the overall economy and stressing the U.S. health care system. Because the cost of caring the AD patients is expected to rise dramatically, the necessity of having a computer-aided system for early and accurate AD diagnosis becomes critical. Several popular non-invasive neuro-imaging tools, such as structural MRI (sMRI), functional MRI (fMRI), and positron emission tomography (PET), have been investigated for developing such a system. The sMRI has been recognized as a promising indicator of the AD progression.

Motivation

Early diagnosis, playing an important role in preventing progress and treating the Alzheimer's disease (AD), is based on classification of features extracted from brain images. The features have to accurately capture main AD-related variations of anatomical brain structures, such as, e.g., ventricles size, hippocampus shape, cortical thickness, and brain volume. Convolutional neural network, which has shown remarkable performance in the field of image recognition, has also been used for the diagnostic classification of AD with multimodal neuro-imaging data. This paper proposed to predict the AD with a deep convolutional neural network (CNN), which can learn generic features capturing AD biomarkers and adapt to different domain datasets.

Dataset

For this purpose we use a dataset which is available through the link below:

https://www.kaggle.com/legendahmed/alzheimermridataset

The dataset consists of 6,400 MRI axial slices. It is separated into test and train set. The test set totally consists of 1,279 images. This number is 5,121 for the train set. The test set is categorized into four groups, similar to the train set. The groups consist of non demented, very mild demented, mild demented and moderate demented categories. The dimensions of all photos are 176 * 208 and all of them are jpg.

Deep Learning Algorithm

To train deep learning algorithms, the following ingredients are required:

- Training data (inputs and targets)
- The model (also called the network)
- The loss function (also called the objective function or criterion)
- The optimizer

Training

The training process for deep learning algorithms is an iterative process. In each iteration, we select a batch of training data. Then, we feed the data to the model to get the model output. After that, we calculate the loss value. Next, we compute the gradients of the loss function with respect to the model parameters (also known as the weights). Finally, the optimizer updates the parameters based on the gradients. This loop continues. We also use a validation dataset to track the model's performance during training. We stop the training process when the performance plateaus.

Model

What is CNN?

Convolutional networks (also known as convolutional neural networks or CNNs) are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn. Convolutional networks, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. Parameter sharing refers to using the same parameter for more than one function in a model. In a convolutional neural net, each member of the kernel is used at every position of the input. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. To say a function is equivariant means that if the input changes, the output changes in the same way.

A typical layer of a convolutional network consists of three stages. In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage. In the third stage, we use a pooling function to modify the output of the layer further. A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood.

The loss function

As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model. An important aspect of the design of a deep neural network is the choice of the cost function. In most cases, our parametric model defines a distribution p(y | x;θ) and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function. Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution. This cost function is given by $J(\theta) = -E_{x,y} \log p_{model}(y \mid x)$.

The optimizer

Optimization used as a training algorithm for a machine learning task differs from pure optimization. Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier. In such situations, one typically optimizes a surrogate loss function instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation. A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. Instead, a machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion is satisfied.

Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. Adam is an adaptive learning rate optimization algorithm and is presented in here:

Adam Algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while
```

Data Preprocessing

Data preprocessing is the process of converting raw data into a well-readable format to be used by a machine learning model. Data preprocessing includes cleaning, Instance selection, normalization, transformation, feature extraction and selection, etc. The product of data preprocessing is the final training set.

Activation Function

- ReLU

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. In modern neural networks, the default recommendation is to use the rectified linear unit or ReLU defined by the activation function $g(z) = \max\{0, z\}$. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces. We can build a universal function approximator from rectified linear functions. One drawback to rectified linear units is that they cannot learn via gradient- based methods on examples for which their activation is zero. A variety of generalizations of rectified linear units guarantee that they receive gradient everywhere. Three generalizations of rectified linear units are based on using a non-zero slope αi when $zi<0$: $hi =g(z,\alpha)i =\max(0,zi) +\alpha i\min(0,zi)$. A leaky ReLU fixes αi to a small value like 0.01. Leaky ReLU has a small slope for negative values, instead of altogether zero. For example, leaky ReLU may have $y = 0.01x$ when $x < 0$. Unlike ReLU, leaky ReLU is more "balanced," and may therefore learn faster.

- Softmax

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

Dealing with Overfitting

What is Overfitting?

Overfitting occurs when the gap between the training error and test error is too large. A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization. There are a great many forms of regularization available to the deep learning practitioner.

- Data Augmentation

The best way to make a deep learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. The performance of deep learning neural networks often improves with the amount of data available. Data augmentation is a technique to artificially create new training data from existing training data. Dataset augmentation applies transformations to your training examples: they can be as simple as flipping an image, or as complicated as applying neural style transfer. The idea is that by changing the makeup of your data, you can improve your performance and increase your training set size.

- Dropout

Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. The term dropout refers to dropping out units (hidden and visible) in a neural network. Dropout is a simple way to prevent neural networks from overfitting. Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer.

- Weight Decay

Weight decay is a regularization technique by adding a small penalty, usually the L2 norm of the weights (all the weights of the model), to the loss function. Because we are subtracting a constant times the weight from the original weight. This is why it is called weight decay. We use weight decay to prevent overfitting, to keep the weights small and avoid exploding gradient.

What is Transfer Learning?

Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks.
In transfer learning approach knowledge is transferred from one model to another. Transfer learning is an optimization that allows rapid progress or improved performance when modeling the second task. Transfer learning is the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned. To deal with the lack of data, we make use of a technique called transfer learning. Briefly, transfer learning is a method that allows us to use knowledge gained from other tasks in order tackle new but similar problems quickly and effectively. This reduces the need for data related to the specific task we are dealing with.

What is VGG Neural Network?

VGG is an innovative object-recognition model that supports up to 19 layers. Built as a deep CNN, VGG also outperforms baselines on many tasks and datasets outside of ImageNet. VGG is now still one of the most used image-recognition architectures.

- VGG16

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes.

- VGG19

VGG-19 is a convolutional neural network that is 19 layers deep. You can load a pre-trained version of the network trained on more than a million images from the ImageNet database. The pre-trained network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

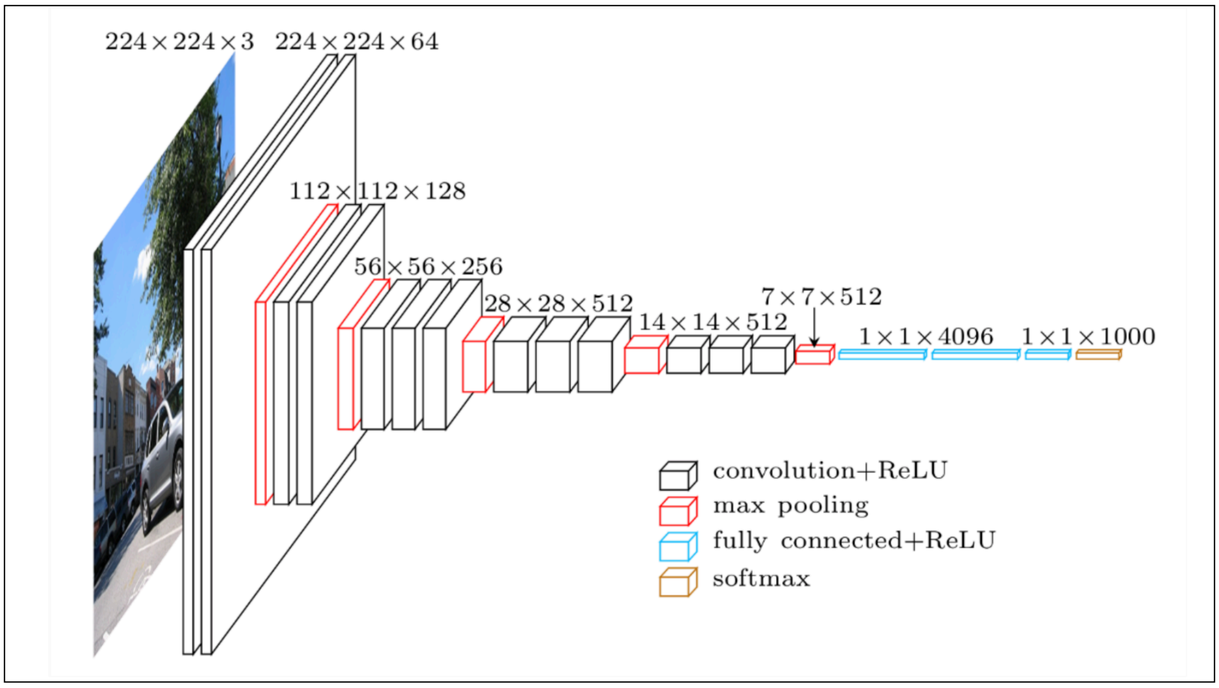

Figure 1)Architecture of VGG16

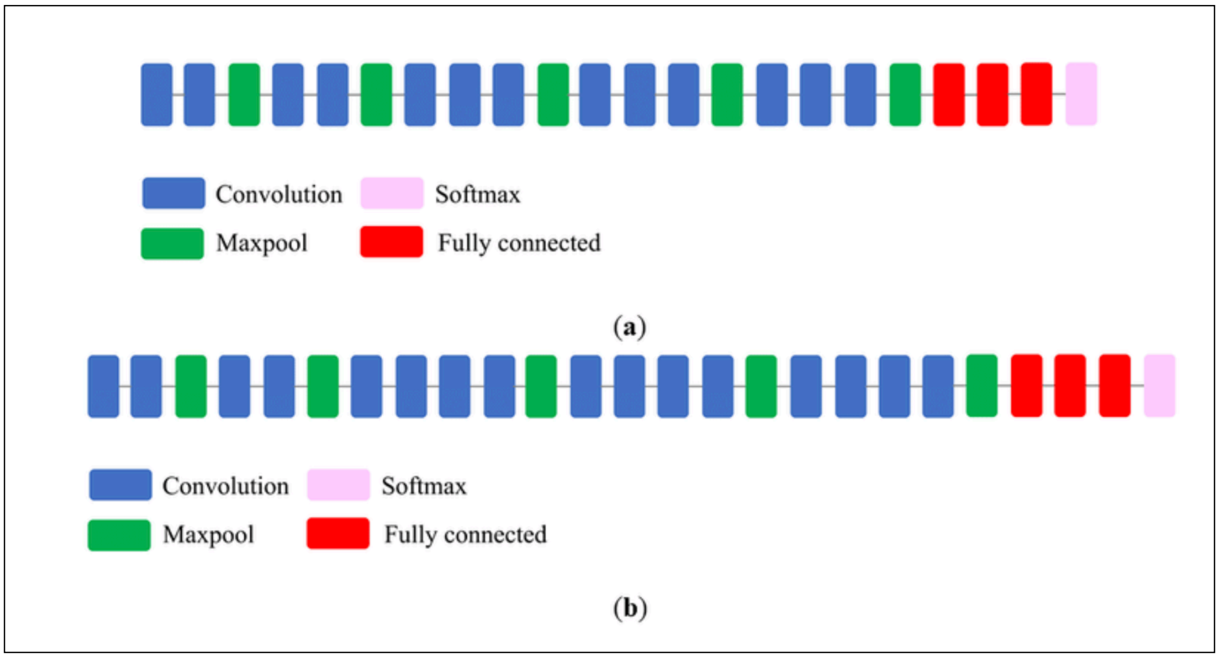

Figure 2)Schematic diagram of (a) VGG16 and (b) VGG19 models

Why PyTorch

There has been significant progress in computer vision because of deep learning in recent years. This helped to improve the performance of various tasks such as image recognition, object detection, image segmentation, and image generation. Deep learning frameworks and libraries have played a major role in this process. PyTorch, as a deep learning library, has emerged since 2016 and gained great attention among deep learning practitioners due to its flexibility and ease of use. PyTorch is a deep learning framework developed by Facebook's artificial intelligence research group.

Report

In the first five trials I utilized the dataset with a ratio of 20% for the validation and 80% for the train set. I implemented a CNN model with 5 convolutional layers, 3 max pooling layers and three linear layers. For activation functions I used ReLU, Leaky ReLU and Softmax functions. For regularization I used dropout. After testing several figures I understood that a 0.3 dropout performs better for the model. For the loss function I used cross entropy and for the optimizer, Adam with 1e-4 learning rate and 1e-5 weight decay. After 100 epochs the result was 65% on validation set and 87% for train, set which shows a great gap between training and validation set. In the second trial I did the same except that I trained the model for 200 epochs. The results show that the model includes overfitting. In the 3rd and 4th trials I used the VGG16 model and trained it for 20 epochs with two optimizers; first SGD with an accuracy of 75% on validation set, second Adam with an accuracy of 76% on validation set. After comparing these two models I found that Adam optimizer performs better on the data, hence I used this optimizer for the other proceeding models. In the 5th trial I applied VGG19 with 50 epochs and a performance of 79% on the validation set. However, the model still shows overfitting.

After those five first trials I decided to change the ratio of the validation and train sets. Therefore, instead of 20% for the validation set I changed it to 30%. Incredibly, the results changed and the overfitting was gone. In the next four trials I used CNN model with 100 and 200 epochs with accuracies of 85% and 92% on validation set respectively. Then I used the VGG16 again and this time I got a result of 93% on validation and 97% on training set. In the final trial I exerted the VGG19 model and got the best results within all previous models. For the training set 98% accuracy and for the validation set 96.3%.

As you can see below, a table is illustrated which shows accuracy and other information about the models:

Trial	Model	Validation Ratio	Optimizer	Epochs	Accuracy(val)	Accuracy(train)
1st	**CNN**	**0.2**	**Adam**	**100**	**65%**	**87%**
2nd	**CNN**	**0.2**	**Adam**	**200**	**68%**	**96%**
3rd	**VGG16**	**0.2**	**SGD**	**20**	**75%**	**95%**
4th	**VGG16**	**0.2**	**Adam**	**20**	**76%**	**96%**
5th	**VGG19**	**0.2**	**Adam**	**50**	**79%**	**99%**
6th	**CNN**	**0.3**	**Adam**	**100**	**85%**	**90%**
7th	**CNN**	**0.3**	**Adam**	**200**	**92%**	**96%**
8th	**VGG16**	**0.3**	**Adam**	**30**	**93%**	**97%**
9th	**VGG19**	**0.3**	**Adam**	**30**	**96%**	**98%**

Table Showing Accuracy for all Implemented Models

In the following you can see plots that show accuracy and loss figures for the last four models; for trial 6th to trial 9th respectively.

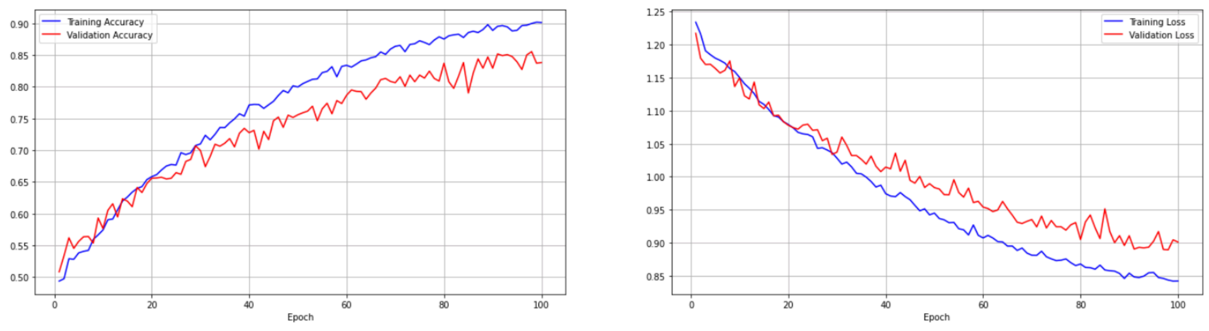

Figure 3)CNN Model - plots showing Accuracy and Loss for training and validation data

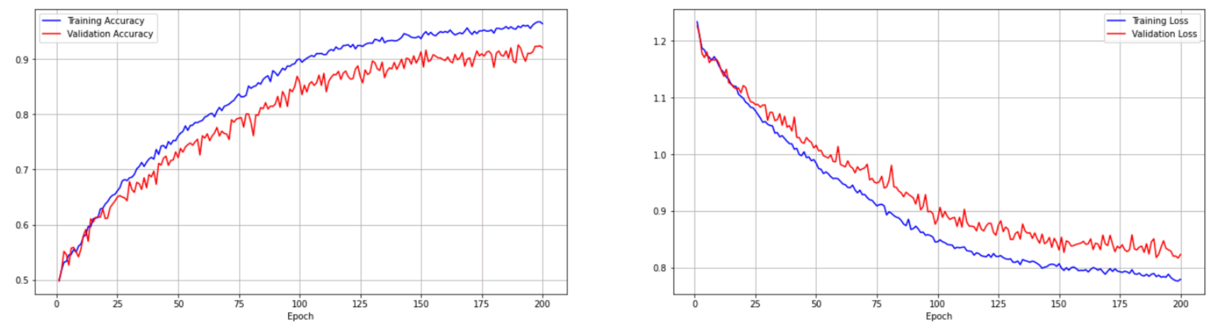

Figure 4)CNN Model - plots showing Accuracy and Loss for training and validation data

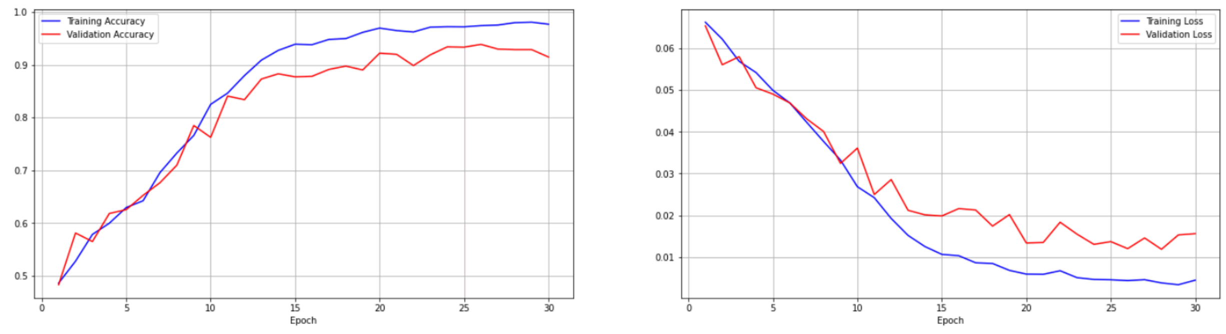

Figure 5)VGG16 Model - plots showing Accuracy and Loss for training and validation data

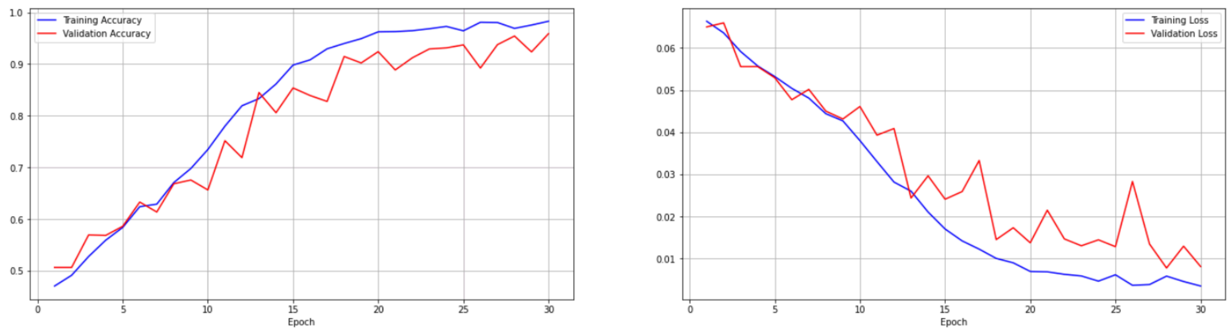

Figure 6)VGG19 Model - plots showing Accuracy and Loss for training and validation data