Nishanth Prajith

Written Report

Case study : Car Image Classification

For this final project, I have decided to undertake the Car Image Classification ML problem and the concept of Convolutional Neural Networks (CNN). Image Classification (often referred to as Image Recognition) is the task of associating one (single-label classification) or more (multi-label classification) labels to a given image. It is one of the most fundamental tasks in computer vision. And for that reason, it has revolutionized and propelled technological advancements in the most prominent fields, including the automobile industry, healthcare, manufacturing, and more. On the other hand, a convolutional neural network, or CNN, is a deep learning neural network designed for processing structured arrays of data such as images. They are the primary type of neural network used when working with any image classification ML problem. To understand the methodology of the experiments conducted in this project, one must first understand neural networks and transfer learning. Understanding these concepts will help better make sense of the code and the final results.

First, let's talk about Neural networks, a subset of machine learning, and are at the heart of all deep learning algorithms. Their name and structure are inspired by the neurons in the human brain, mimicking how biological neurons signal to one another. The structure of a neural network comprises an input layer, one or more hidden layers, and an output layer. Each node connects to another and has its associated weight and a threshold value. And

like all ML models, neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are tuned for accuracy, they become powerful tools, allowing us to classify and cluster data. An example of a neural network that is most familiar to us is Google's search algorithm, a recommendation engine that provides relevant results based on a search query.

On the other hand, transfer learning is learning something new from the knowledge of a related task. In other words, it is taking the understanding of an already trained machine learning model and applying it to a different but related problem. Learned features of a model are often transferable to different data. For example, a model trained on a large dataset of bird images will contain learned features like edges or horizontal lines that one could transfer to a different dataset of dogs and cats. So we can see the primary goal of transfer learning is to improve the learning of the target task by leveraging knowledge from the learning of an earlier task. Overall, transfer learning has several benefits, including improved training time, better performance (in most cases), and not requiring a lot of data. These are all quite important as, usually, a lot of data is necessary to train a neural network, but since access to that data isn't always available, transfer learning comes in handy. With transfer learning, one can build a solid machine learning model with comparatively little training data because the model is already pre-trained. Additionally, training time is reduced because sometimes training a new neural network from scratch would mean the weights of each node would have to be updated several thousand times. While using a pre-trained model, most of the time, the weights would only have to be updated a few times, bringing down the training time from days or even weeks. Now there are many approaches to

transfer learning. For this project, I made use of 2 such methods: Fine-Tuning and Feature Extraction.

Fine-tuning and Feature Extraction are very similar but follow different training ideologies. They both use a pre-trained model like *DenseNet,* but the part they differ is which weights they train. Unlike regular neural networks, convolutional neural networks contain additional layers, specifically convolutional and pooling layers. And convolutional layers, like other layers in a neural network, have their own weights, and updating these weights during training is where the feature extraction and fine-tuning methods differ. The Fine-Tuning method updates the weights of all the layers in the CNN, the convolutional and fully connected layers. While Feature Extraction only updates the weights of the fully connected layers and freezes the weights of the convolutional layers. And as you probably can tell, each method has its benefits and drawbacks. In the case of Fine-Tuning, the benefit is that the pre-trained model weights get more tailored to the new training dataset allowing for greater accuracy, but the drawback is that it takes more time to train as a deep CNN has a lot of weights that need to be updated. And in the case of Feature-extraction, the training time is better than the fine-tuning method as only the weights of the fully connected layers are updated. Still, the drawback is that the accuracy might not be great if the pre-trained model was trained so well on the original task that it might not be able to adapt to the new task at hand. There are different ways to improve accuracy using the Feature Extraction, but that would come at the cost of training time as one would need to train the model for more iterations to achieve greater accuracy. A visual representation of Fine-Tuning and Feature extraction structure is shown in **Image 1**, where the blue boxes represent the areas where the model's weights are updated.

Now that a brief understanding of neural networks and transfer learning has been established let's get into the specifics of the experimental setup. The classification problem being tackled here is car image detection. In other words, given an image of a car, is it possible to build a classification model that can recognize the car in the image and give some information about the car. To help tackle this problem, we will use the Stanford Cars Dataset **(Link 1)**, which includes 16,186 images with 196 classes (cars types) and the cars' make, model, and year. An example of one image from this dataset is shown in **Image 2**. Regarding the ML model, we will be using some pre-trained models; specifically, we will use ResNet34, VGG16, and DenseNet models. And for all 3 of these pre-trained models, we will apply both the Fine-Tuning and Feature extraction transfer methods. Applying both feature extraction and fine-tuning transfer methods should give us a better understanding of which pre-trained models work best and how the same pre-trained model compares when used with different learning methods.

In addition to using a pre-trained model for improving training time and performance, one additional technique commonly used and we will use is image augmentation. Image augmentation is the idea of increasing the size of the dataset but adding healthy noise as what would be reflected in the real world. This is done to make the model more general as not all images taken in the real world are perfectly aligned, symmetrical, and have perfect color and clarity. One can change many things in an image, such as the brightness, the contrast, the color jitter, and even skewing the original image somehow to make it different from the original image. In this project, two image augmentation techniques are used: random horizontal flip, which randomly chooses to flip an image with 50% probability horizontally, and random image rotation by 15 degrees. You can see how these two

augmentation techniques affect an image in the visual shown in **Image 3.** In the **Image 3** visual, we can see that the original image was transformed into other images after applying the aforementioned image augmentation techniques. One thing to note here is that although it looks like the two techniques work independently of each other, both modifying the original image. It is still possible for both the augmentation methods to work one after another, as demonstrated in **Image 4,** where random horizontal flip and random rotation work one after another.

In terms of optimization of the ML model, we will use the SGD (Stochastic Gradient Descent) optimizer. Optimizers are what help tweak the weights of the model during the training process, to make predictions as correct and optimized as possible. They tie together the loss function[1] and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by playing around with the weights (ML Glossary). In our case, we will use the SGD optimizer tuning the hyperparameters[2] learning rate and momentum. Learning rate is what determines the step size at each iteration while moving toward a minimum of a loss function. And so if the learning rate is too low, we might not reach the minimum and if it is too big, we might overshoot. Hence, keeping the learning rate somewhere in the middle is essential for the best performance. To achieve a reasonable learning rate, one additional optimization technique will be used: a learning rate scheduler. A learning rate scheduling algorithm helps reduce or increase the learning rate with respect to

[1] The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

[2] A hyperparameter is a parameter that is set before the learning process begins.

a particular metric. In all of the experiments conducted for this ML project we will use the *ReduceLROnPlateau* learning rate scheduling algorithm. The specific hyperparameters for this algorithm will be 'mode' which will be set to 'max' and 'patience' which will be set to '3'. The 'mode' hypermeter with the 'max' value will reduce the learning rate when the value monitored has stopped increasing, in our case we will make use of the training accuracy. Essentially during each iteration in the training process, we will calculate the training accuracy of the model and give that to the learning rate scheduling optimization algorithm and have that make a determination as to whether to reduce the learning rate. The 'patience' hyperparameter is how many iterations the scheduling algorithm waits before changing the learning rate. It waits to see if there is an improvement in the metric with the initial learning rate and then if not it changes the learning rate. The learning rate scheduling optimization algorithm overall is perfect because we don't need to worry about guessing too low for the learning rate, we can start at a high learning rate (not too high) and then let the algorithm do the rest.

Now that the experimental setup for the project has been established, let's talk about the results of each of the experiments with the different pre-trained models. First, we will discuss the results we got on the ResNet34. The result of this model is shown in **Image 5.** As we can see the model managed to get an accuracy of 88.62% on the Fine-Tuning transfer method and 37.79% on the Feature Extraction transfer method. Comparing the two accuracies, it is quite clear how better Fine-Tuning works than the Feature Extraction method. This was mentioned when we talked about the two transfer learning methods. We did suggest that it is possible that the model might not achieve good accuracy when using the feature extraction method unless, of course, it was trained for longer iterations. In this

case, we trained both methods for only 10 iterations. Hence, it is possible that feature extraction did not have enough time to tune the weights. Now taking a closer look at the architecture of ResNet34 **(Link 2)** we can see something interesting, there is only one fully connected layer for the model, and the number of outputs for this model was 1000 as it was classifying a dataset with 1000 classes. Since our model has only 196 classes, it is quite clear that this single fully connected layer did not have the necessary time to adapt as this model has a lot of convolutional layers making it quite deep. On the other hand, Fine-Tuning was able to perform better because it was able to modify the weights of the convolutional layers and take advantage of the deep complexity of the model. Overall, even though the Feature Extraction method model had a lower accuracy, it is still quite impressive what it managed to do with what it had to work with.

Now, let us talk about the results of the VGG16 model. The results are shown in **Image 6.** As we can see, the model achieved an accuracy of 71.58% using the fine-tuning method and 31.24% using the feature extraction method. Comparing the accuracies to the ResNet34 model, the accuracy of Fine-Tuning decreased by 17.04%, and for the feature extraction, it decreased by 6.55%. The decrease in accuracy for fine-tuning is quite staggering, but it still shows that not all models provide the same results. To get a deeper understanding of why the accuracy decreased so much, we can look at the visualization of the layers of the VGG16 model **(Link 3)** and compare that with the visualization of the ResNet34 model. Looking at the visualization, we see two things compared to the ResNet34 model: there are fewer convolutional layers, and the second there are more fully connected layers. The fewer convolutional layers compared to the ResNet34 helps explain the decrease in accuracy. This

means fewer features the model detects on an image, meaning it is less abstract than what one would expect from a model like the ResNet34 with more convolutional layers.

Finally, let us talk about the results of the DenseNet model. The results are shown in **Image 7.** As we can see, the model achieved an accuracy of 92.00% using the fine-tuning method and 91.69% using the feature extraction method, both of which are very close. Comparing this to the previous two models, the difference is quite huge. Only ResNet34 using the fine-tuning method with an accuracy of 88.62% comes close to this model. But one significant downside of achieving this accuracy is the training time. Specifically for the fine-tuning method, which we already know takes a while for training, this model took around 44 minutes. Comparing this to ResNet34, which took less than 25 minutes, the difference in accuracy starts to seem less meaningful/significant. Granted, it is the better model, but it is very much possible with more iterations, the ResNet34 could have achieved an accuracy equal to or greater than the current accuracy of the DenseNet model. Nonetheless, let us look at the architecture of the DenseNet and see what makes it special **Image 8.** One of the first things we notice is that in a DenseNet architecture, each layer is connected to every other layer. And for each layer, the feature maps of all the preceding layers are used as inputs, and their feature maps are used as input for each subsequent layer. This idea sounds quite simple and has several advantages. To cite the authors, “DenseNets have several compelling advantages: they alleviate the vanishing gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters.” Going back to our observation of how close the accuracy of both fine-tuning and feature extraction we can see that this can be explained by the general structure of the DenseNet. Since one of the benefits of DenseNet is feature reuse, it seems

quite likely that during the training process, the features the model picked out were so abstract that they could be used to apply to other image classification problems without significant loss in accuracy. This results in both the feature extraction and fine-tuning methods being close to each other in terms of accuracy. Although it is quite interesting, as I did not expect the two methods applied to the same model to be so close, in the end, it was clear that the model had more to offer than what meets the eye.

To understand our DenseNet Fine-Tuning model (which had the highest accuracy) a little better, let us look at two misclassified images **Image 9.** Looking at the misclassified images, we see that most of the images are either close calls like here with the Acura RL Sedan or are misclassified due to unconventional angles like with the Acura Integra. The difference is quite minuscule in the Acura RL and Acura TL. The only visual difference is with the body shape and the windows. And for the Acura Integra, we see the backside of the car, and it makes sense why it might be hard for the model to identify the car. Even a human who does not have much understanding of cars would not be able to properly distinguish cars like the Acura RL and RL sedan or identify a car from the backside of the car. Hence, overall the model has done a good job classifying the cars and having such good accuracy.

In conclusion, for the future I hope to try more pretrained models along with different optimization methods like 'Adam' to see how that affects the accuracy. This would give a better picture of which pretrained models work best and which optimization works best for a particular pretrained model. Another thing I would like to try is more image augmentation techniques to try to improve accuracy. We have used some image augmentation but looking at the misclassified images, we see that some of the images that were misclassified have some conventional angles so we could try using more augmentation to help the model deal

with this and better classify those misclassified images. However, this would mean that more data would need to be gathered first to make the model more better and general. Overall the project was really fun and I learned a lot of ML pretrained models, different transfer learning techniques, image augmentation techniques, and most importantly PyTorch for working with machine learning.

Images

Image 1

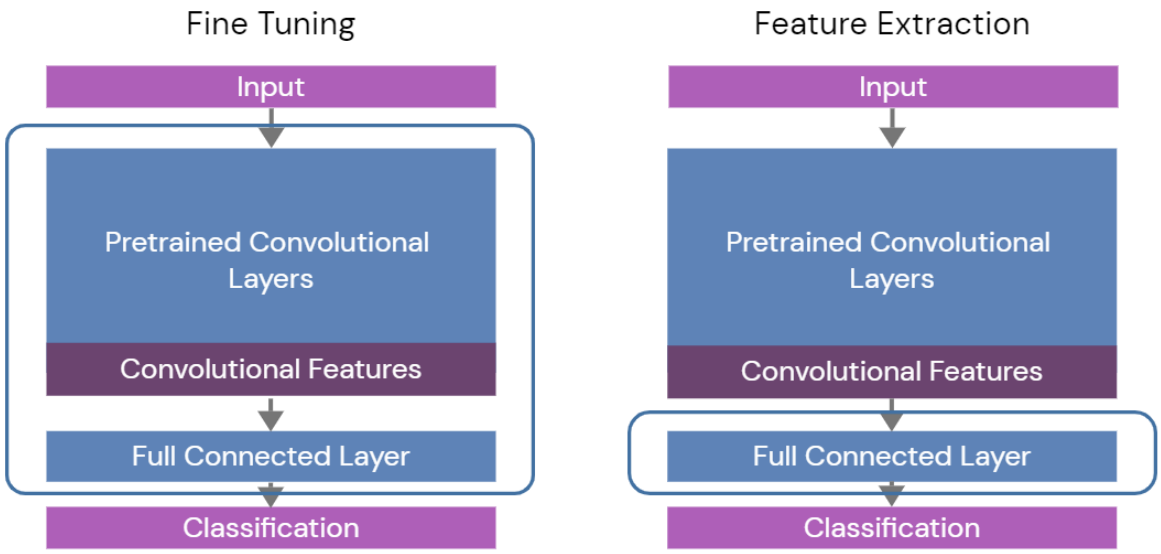

Image 2

Aston Martin Virage Coupe 2012

Image 3

Original Image

Horizontal Flip

Rotation by 15°

Image 4

Horizontal Flip and Rotation by 15°

Image 5

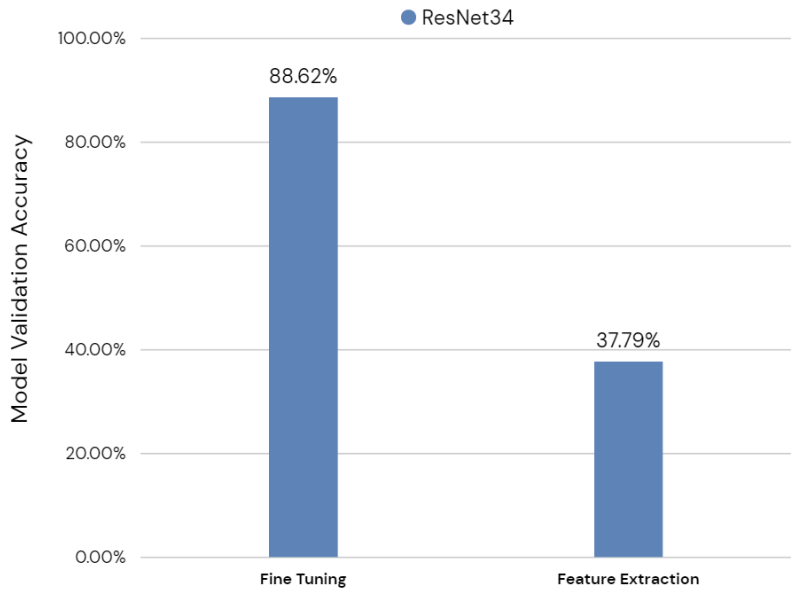

Image 6

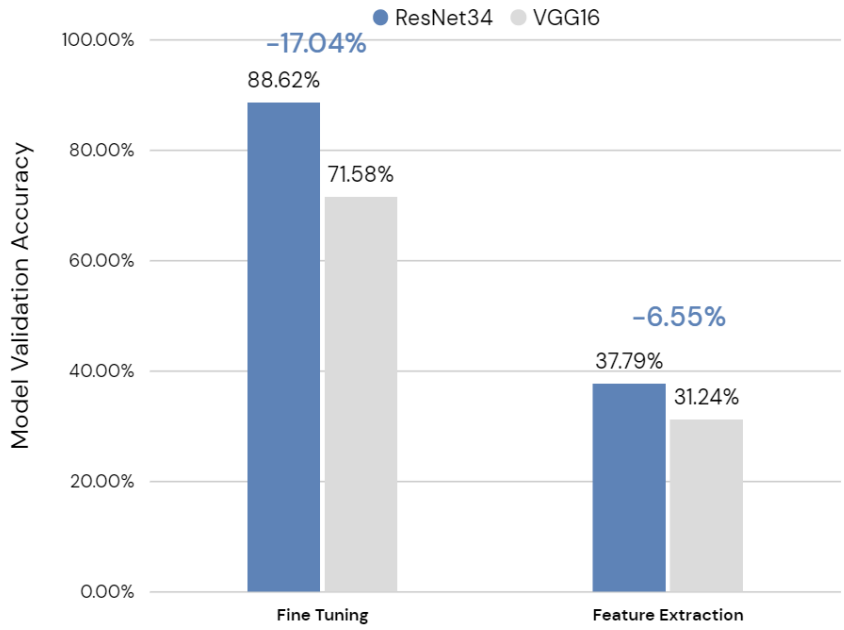

Image 7

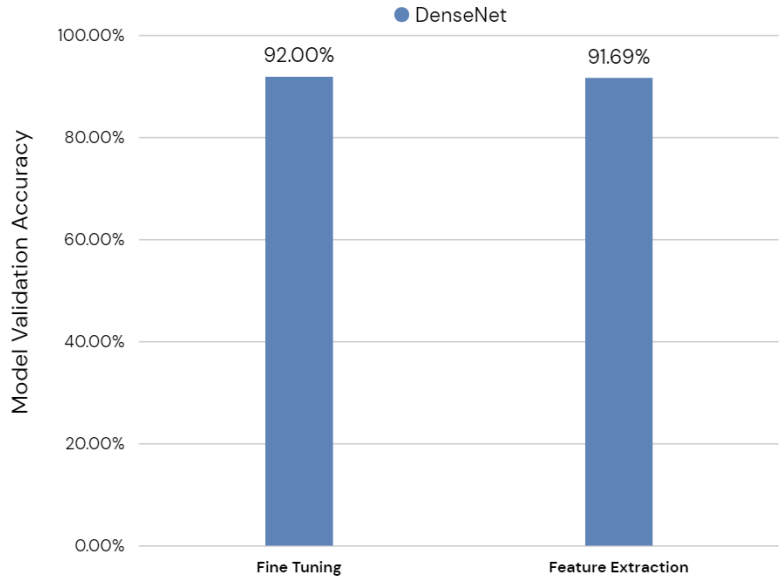

Image 8

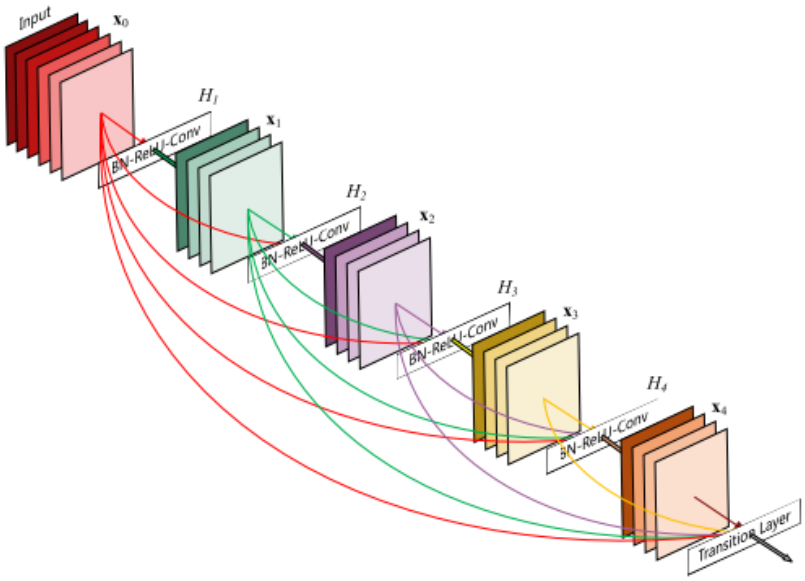

Image 9

Acutal		
	Acura Integra Type R 2001	Acura RL Sedan 2012
Predicted		
	Dodge Challenger SRT8 2011	Acura TL Sedan 2012

Links

Link 1 : https://ai.stanford.edu/~jkrause/cars/car_dataset.html

Link 2 : ResNet-34 | Kaggle

Link 3 : Netscope (ethereon.github.io) -> VGG16 model architecture