**Course number: ECE 1512**

# Fruit Recognition using CNN with Transfer Learning

Yisi Zou 1001826321
Tianyi Xie 1000570679

Date Due: Monday, April 1, 2019
Date handed in: Monday, April 1, 2019

## Contents:

## 1.Introduction

Fruits are popular in our life because they are delicious and nutritious. However, sometimes it is hard to recognize fruit categories if they have similar colors and shapes. Different types of apples, such as Gala and Fuji, is a good example. Current fruit recognition process relies on product tags, but tags do not provide information such as place of origin and nutritive values, which might be interested by consumers. Also, currently cashiers need to enter the product codes stuck on fruits to get the price. If a fruit can be recognized directly to get its price by scanning when weighing it, it will be more convenient for cashier and save time for the consumers. In this case, we are proposing a system of fruit recognition that intends to automatically get images of a single fruit and detect its category. Further information such as price, place of origin, and nutrition could be added by the grocery shops based on our fruit recognition system.

In the past few years, several computer vision strategies are used to recognize fruit. Some researchers used fruit’s basic features to recognize the category of the fruit. For example, a fruit recognition system was proposed based on analysis on color shape and size of each fruit and used k nearest neighbors [1], while another algorithm used 4 basic features, intensity, color, shape and texture, to recognize fruit based on minimum distance [2]. In addition, some researchers used neural networks to recognize fruit. The models introduced by Zhang et al. [3] and Wang et al. [4] uses feedforward neural network to do fruit classification while the model proposed by Zhang et al. [5] used convolutional neural networks. All these models can achieve accuracy more than 80%, however, they got the results based on less than 20 classes which do contain the detailed classification of each fruit. For example, instead of ‘Gala Apple’ and ‘Fuji Apple’, they only contain the category ‘Apple’. The fruit with similar features cannot be recognized correctly by these models.

The model provided by Muresan and Oltean [6] used fruit dataset with 95 classes which classify the fruit in detail. The neural network has 4 convolutional layers and 2 fully connected layers. We decided to use the same dataset as this model and optimize the neural network structure to increase the accuracy of fruit recognition.

In our design, we focus on training a Convolutional Neural Network (CNN) and apply transfer learning to proceed with the classification process. PyTorch library will be used because it provides pre-trained models on 1000 classes ImageNet data [7], which is easier to implement transfer learning. The input image dataset has been preprocessed in both the Fruit-360 dataset [6] and using PyTorch. Three CNN models, ResNet, InceptionNet and DenseNet are used to do fruit recognition based on the same training and test datasets to compare the accuracy of results.

# 2. Background

This section provides technical background information related to the topic.

## 2.1 CNN

Convolutional Neural Networks (CNN) are multilayer neural networks which can recognize visual patterns directly from images with little preprocessing [8]. Compared with standard feedforward neural network with similarly sized layers, CNN is trained with a back-propagation algorithm. They have fewer connections and parameters, which make them easier to train [9]. Recently, CNN has proved to be an outstanding model in large-scale image recognition because of the development of large public image repositories, such as ImageNet, and high-performance computing systems, such as GPUs [10].

Here, we mainly introduce and implement three modern architectures of convolutional neural networks: InceptionNet, ResNet, and DenseNet.

**2.1.1 InceptionNet** (also known as Inception Neural Network):
In order to improve the performance of deep neural networks, the most straightforward ways are to increase the number of layers and to increase the number of units at each layer [11]. However, these ways have two drawbacks. One of them is that neural networks have a lot of parameters, which makes the enlarged network more prone to overfitting, especially for small training dataset. The other is that increasing network size results in increasing the use of computational resources. For example, one additional convolutional layer chained to the model would result in a quadratic increase of computation [11]. InceptionNet, derives from network in network, are proposed as one solution to solve these problems. InceptionNet is a deep network with computational efficiency while the accuracy is maintained. It applies parallel filter operations with different filter sizes on the input from previous layers to make the network 'wide' instead of 'deep'. To avoid patch alignment issues, the convolution filter sizes are restricted to 1×1, 3×3 and 5×5 pixels, and the max pooling filter has size 3×3 pixels. The outputs of filters are concatenated into a single output vector and sent to the next inception module. To avoid expensive computation, "bottleneck" layers that use 1x1 convolutions are added to reduce feature depth [11].

An induction model with dimension reduction is shown in Figure 2.1. As we can see from the figure, the 1×1 convolution layer is added before the 3×3 and 5×5 convolution filter layer and after the 3×3 max pooling filter layer.

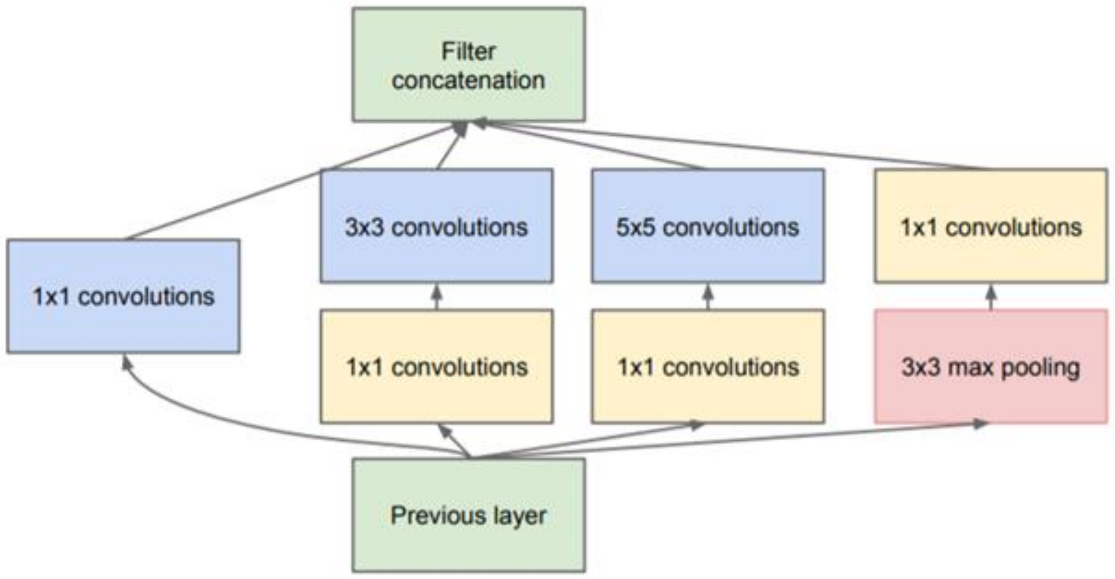

**Figure 2.1 Inception module with dimension reductions[11]**

InceptionNet V3 is the third version of InceptionNet deep convolutional neural network. It factorizes convolutions to reduce the number of connections, or parameters, without decreasing the network efficiency [12]. It uses two 3×3 convolutions with 18 parameters to replace one 5×5 convolution with 25 parameters, which reduces the number of parameters by 28%. The model also uses one 3×1 and one 1×3 convolution filters with 6 parameters to replace one 3*3 convolution filter with 9 parameters, which reduces the number of parameters by 33% [12]. As the number of parameters is reduced, the network in InceptionNet V3 would be harder to overfit comparing to the original InceptionNet.

**2.1.2 ResNet** (also known as Residual Network):
In theory, very deep networks can represent very complex functions, which is useful in real life cases. However, there are some obstacles. One problem is the vanishing or exploding gradient. When the network is too deep, gradients of the loss function is easily shrinking to zero, which results in the weights values never update and the network performs no learning [13]. Another problem is when the network is very deep, as the network depth increases, the accuracy gets saturated and then degraded rapidly. This degradation has no relation with overfitting and adding more layer to the model makes the situation worse [13].

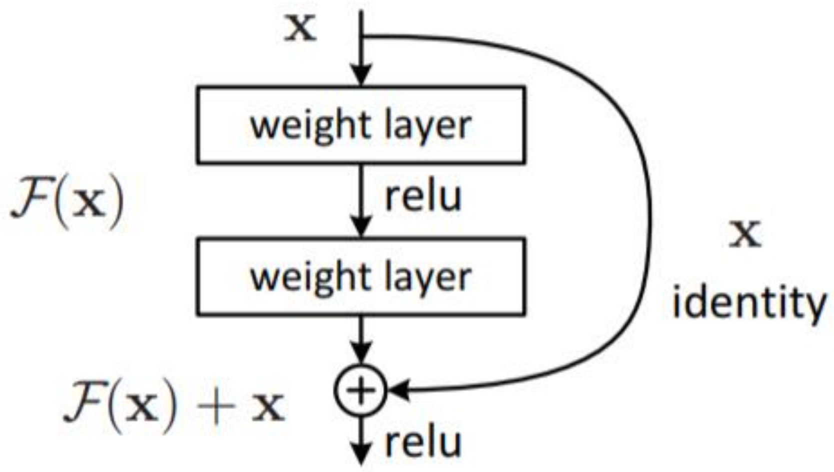

**Figure 2.2: Residual learning：a building block [13]**

As shown in Figure 2.2, in the ResNet process, by skipping connections, it takes the activation from one layer and suddenly feed it to another layer even much deeper in the neural networks. Through the skipping connections, the gradients can flow directly from the later layers backward to initial filters [13]. ResNet enables people to train extremely deep networks with considerably high accuracy.

**2.1.3 DenseNet** (also known as Dense Convolutional Network):
Similar to ResNet, DenseNet tried to solve the problem of vanishing gradient for exceedingly deep neural networks. Compare with ResNet, DenseNet makes the connection patterns simplified between layers. In order to achieve information flow maximization between the layers in the network, DenseNet connects all layers with matching feature-map sizes directly with each other, and each layer has direct access from all preceding layers [14].

Figure 2.3 shows this layout. Instead of combining features using addition before passing them to a new layer in ResNet, DenseNet combines features by concatenating them. DenseNet exploits the potential of the network through feature reuse which makes it has high parameter efficiency and easy to train. Also, it has high computing efficiency and maintains low complexity features [14].

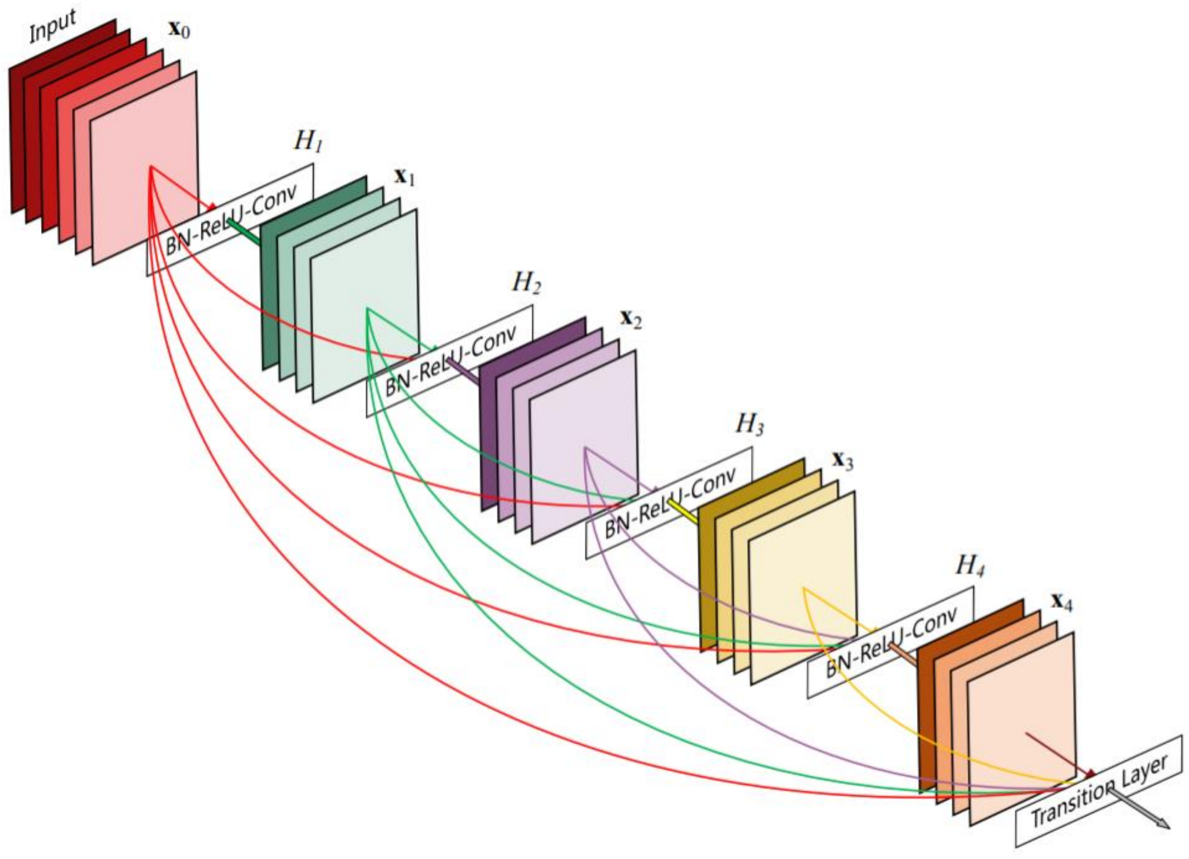

**Figure 2.3: A 5-layer dense block with a growth rate of k = 4. Each layer takes all preceding feature-maps as input. [14]**

## 2.2 Transfer Learning

Time and data-expensive are considered as major drawbacks for CNN.  Transfer learning is proposed as one of the solutions. It is known as a technique that applies a pre-trained machine learning model to a different but relative problem. Applying transfer learning could avoid the shortage of dataset applied, to reduce training time and to reduce re-calibration effort [15]. Performing transfer learning requires first finding an existing deep neural network with pre-trained models and keep the lower layers setup of its. The pre-trained parameters are then transferred to the target tasks. Parameters in the upper layers are trained based on the new data. The upper layers of the original model are not as useful as lower layers because the high-level features of the new data are different than original data [16]. Transfer learning saves training time and requires fewer data to train in order to get acceptable accuracy.

## 2.3 ImageNet

ImageNet is a large image dataset used for research. It is organized according to the WordNet hierarchy which each node of the hierarchy contains an average of more than 500 images. Images of each concept are quality-controlled and human-annotated [17]. Because of its quantitative advantage and detailed classification, ImageNet is widely used for many research projects and it is a good dataset that can be used for a pertained model for transfer learning.

## 2.4 PyTorch

PyTorch is an open-source machine learning library for python based on Torch.  Torch is a tensor library for manipulating multidimensional matrices of data employed in machine learning and many other math-intensive applications [18]. Tensors can be used on GPUs that support CUDAs, which can improve the efficiency of computing and training. PyTorch provides pre-trained models, such as ResNet, AlexNet, VGG, SqueezeNet, DenseNet, and InceptionNet on the 1000-class ImageNet dataset, which makes it easy to finetune and apply feature extraction based on these models [7].

# 3. Methodology

This section provides a description of Python experiments using three CNN models for fruit recognition. It explains data input used in Python experiment, preprocessing, experiment setup and performance measurement.

## 3.1 Dataset

The dataset we used for transfer learning is ImageNet, which contains 1000 categories of data [7]. The target dataset is called Fruits-360[6], which is a dataset of different types of fruits. It consists of 65429 100 x 100 pixels color images in 95 classes and will be used as a target dataset for fruit classification. Our training set has 48905 images (one fruit per image), and our test set has 16421 images (one fruit per image).

## 3.2 Preprocessing

There are two parts of data preprocessing. First, the Fruit-360 dataset has been enlarged when it is created. According to the Fruit-360 dataset properties, each fruit is planted in the shaft of a 3 rpm low-speed motor and a 20 seconds movie is recorded [6]. Images with different angles of the fruit are taken from the video. Also, the rotated images of the original images are generated and included in the Fruit- 360 dataset, which enlarges the dataset. Second, the dataset is enlarged using Torchvision transform function before training and validating. Images are randomly cropped and flipped horizontally. After these two enlargement methods, the neural networks could be trained on a larger dataset, thus the result generated from the neural network becomes theoretically more accurate.

## 3.3 Experiment Setup and Performance Measurement

For the purpose of transfer learning, we fix the parameters in the first three major layers of ResNet, InceptionNet, and DenseNet that are pre-trained based on the ImageNet dataset. Figure 3.1, 3.2 and 3.3 show the training structure of the three network models that we implement. It is noteworthy that minor layers such as transition layers and down sampling layers are not counted as major layers. The batch size is set to 48. Both the number of fixed layers and the batch size are fine-tuned based on the training time spent and accuracy for the first epoch of all three models. We set the learning rate as 0.01, which is the recommended learning rate provided by torchvision [7]. The output layer is set to have 95 nodes in order to match the number of classes in the dataset. We train the upper layers in all three neural networks for ten epochs, recording their training errors and test accuracies for each epoch. Measuring both training error and test accuracy allows us to detect overfitting issues. Training time spent is also a factor that we measure in order to compare these three models.

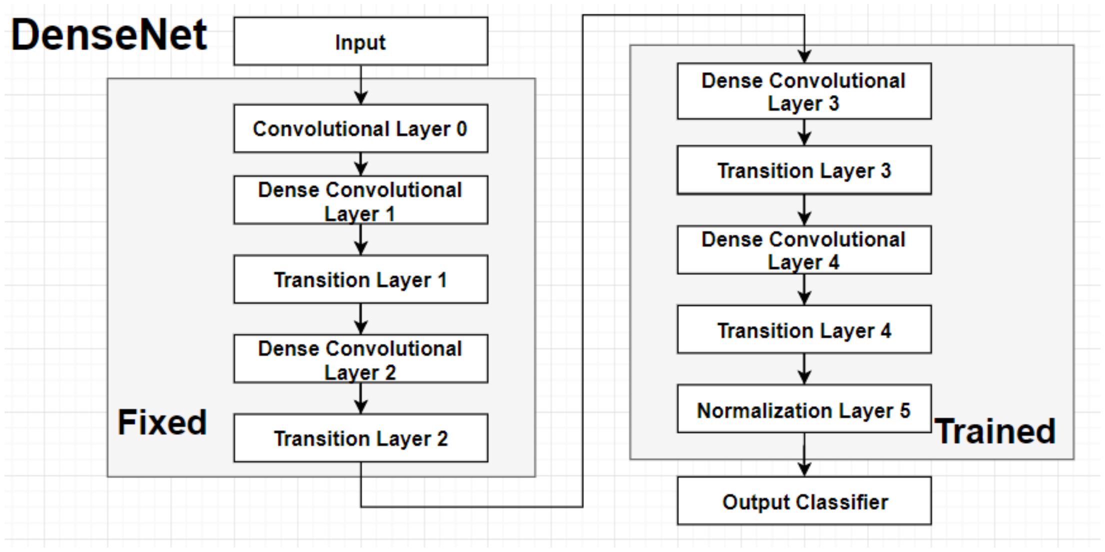

Figure 3.1: Applied DenseNet Training Structure

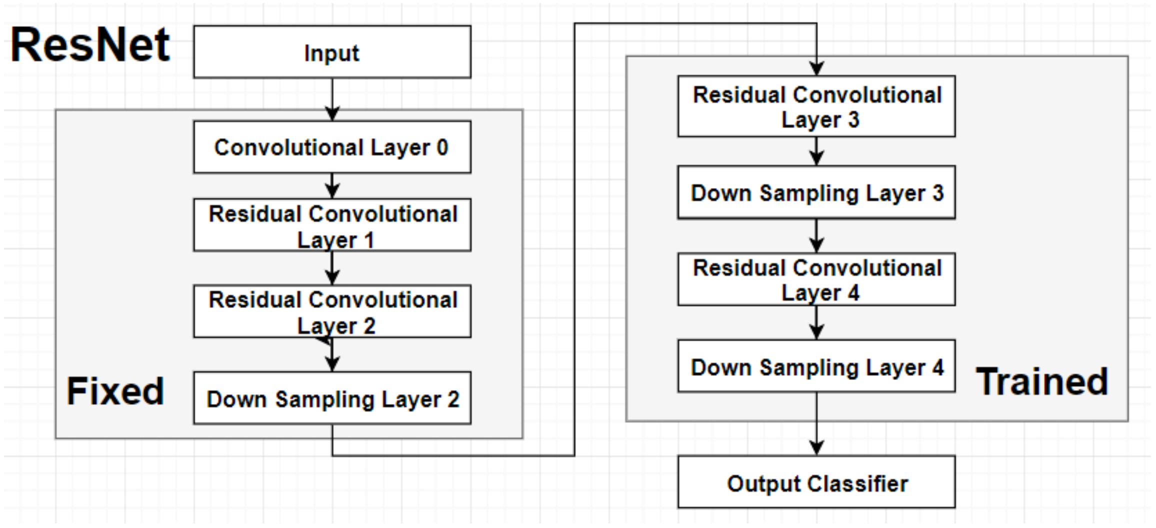

Figure 3.2: Applied ResNet Training Structure

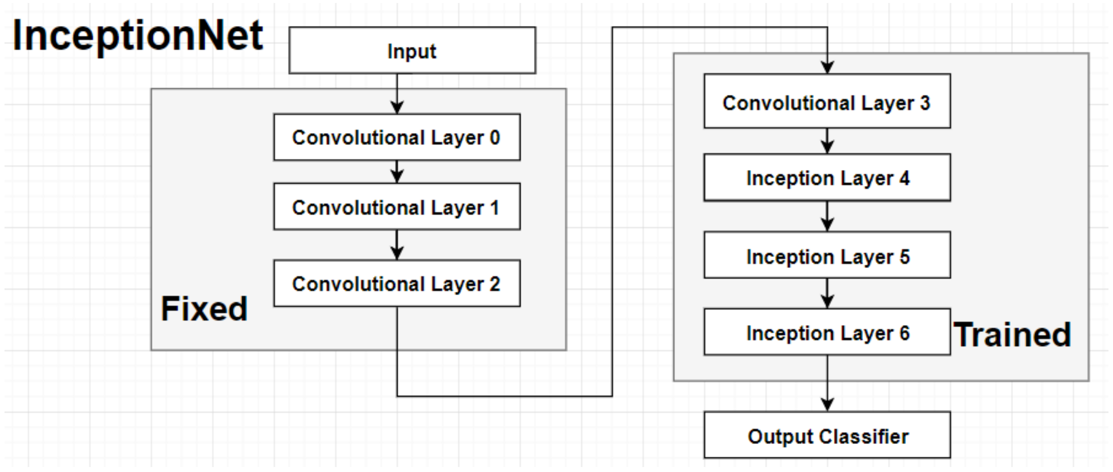

Figure 3.3: Applied InceptionNet Training Structure

# 4. Results

Three different models, namely ResNet, InceptionNet and DenseNet, were executed for 10 epochs each. Graphs of the results are provided as below:

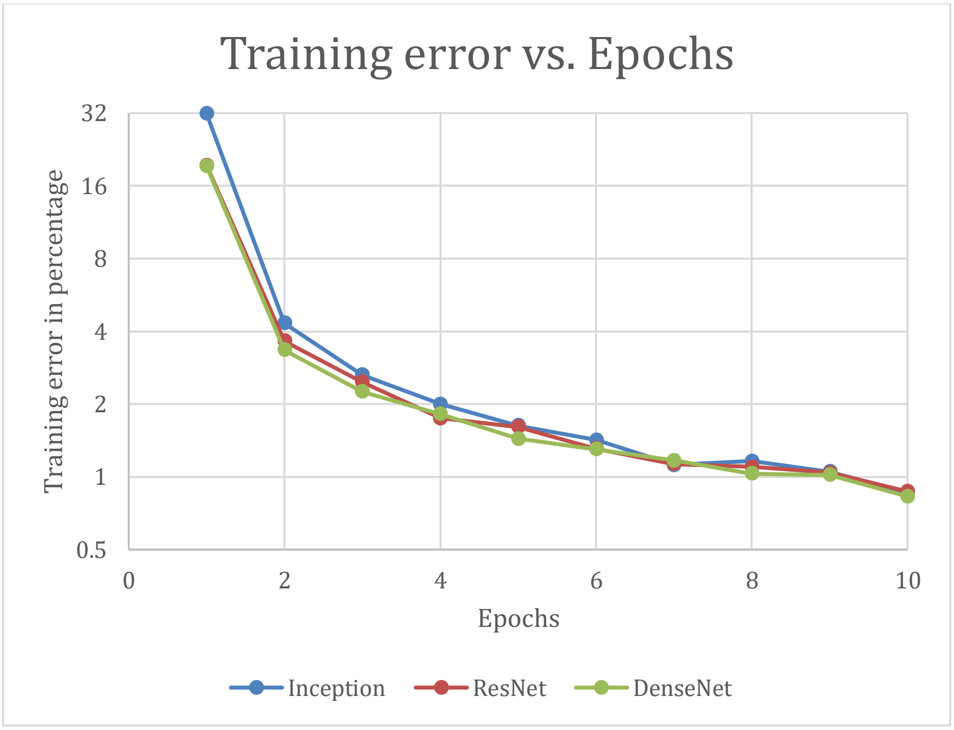

**Figure 4.1: Training error vs. Epochs for different models**

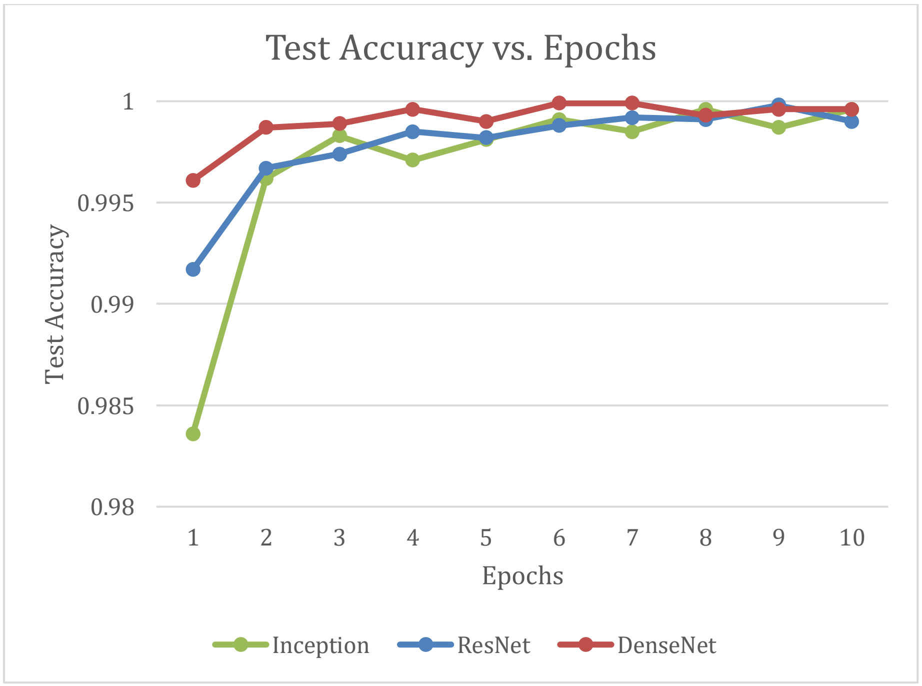

**Figure 4.2: Test Accuracy vs. Epochs for different models**

As shown in Figure 4.1, as epoch number increases, the training error declines rapidly for the first 3 epochs and declines slowly in later epochs. All three models have the same tendency of training error when the epoch number increases.

As shown in Figure 4.2, as epoch number increases, the test accuracy increases rapidly for the first 3 epochs and converges in later epochs. Test accuracy fluctuates after it converges. Comparing the three models, DenseNet has the highest test accuracy.

Combined with Figure 4.1 and Figure 4.2, we observe that even though the training errors are still decreasing in the last three epochs for all three models, the validation accuracies start fluctuating or even declining. We, therefore, claim that having additional epochs would not significantly increase the accuracies of all three models, as overfitting issues may occur.

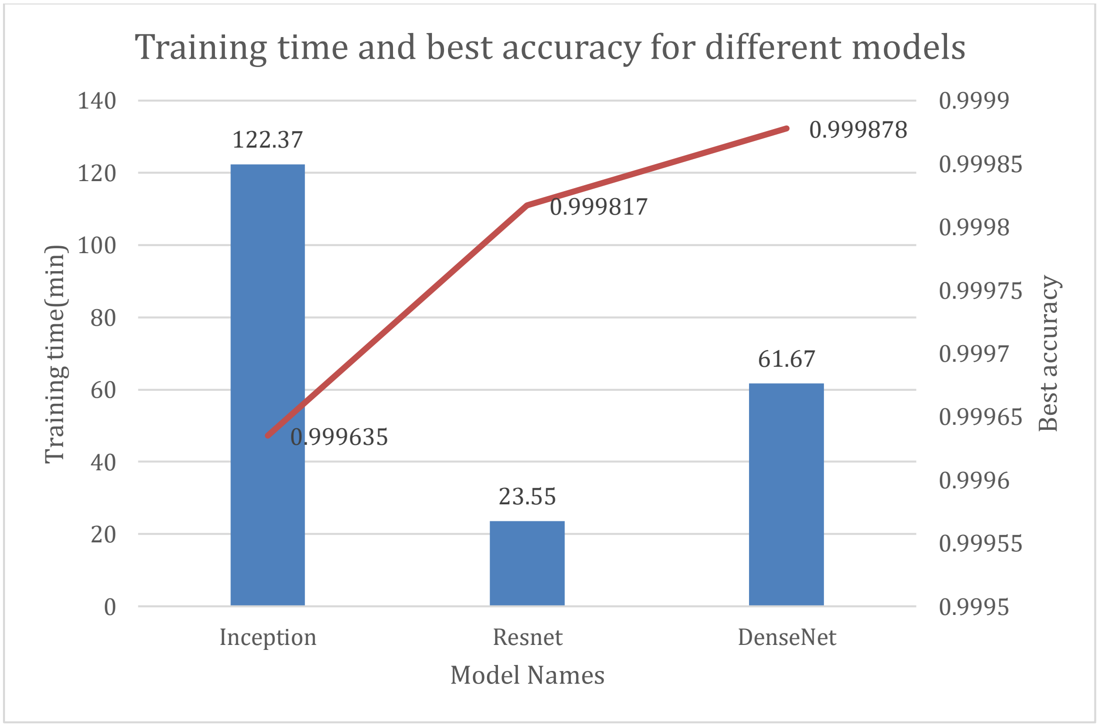

**Figure 4.3: Training time and best test accuracy for different models**

As shown in Figure 4.3, after training and testing, InceptionNet model has the longest training time while the accuracy is relatively low. Both ResNet and DenseNet produce acceptable results. ResNet has the shortest training time, which is 23.55 minutes, while DenseNet has the highest accuracy, which is 99.9878%. If training time is not a critical factor, DenseNet is recommended as the optimal model.

## 5. Conclusions and Future Improvements

All three models of convolutional neural network that we choose, namely ResNet, InceptionNet and DenseNet, can recognize different categories of fruits from Fruit-360 dataset with considerably high accuracy. We conclude that DenseNet has the highest performance among the three chosen models, since it has the highest accuracy, and it is relatively inexpensive computationally. Using the neural network that we trained, grocery shops may develop applications that could recognize a specific type of fruit and provide further information on their product.

We also compared our results to the neural network trained by the dataset provider [6]. In their report, data preprocessing such as converting RGB images into HSV + Grayscale images was applied. We conclude that this is not feasible for our neural network, as all three neural networks we chose RGB images as their input. Nevertheless, the best test accuracy achieved in their report was 97.04%, which is lower than our results 99.9878%. We conclude that this is because we choose more complex neural network structures compared to their models, in addition to applying transfer learning from a larger dataset.

Future improvements are required before this algorithm becomes commercialized. Most images from Fruit-360 dataset have one piece of fruit per image. If we want to recognize more than one piece of fruit in a single image, the model may not recognize their category correctly. Furthermore, all images in Fruit-360 dataset have pure white backgrounds. Whether the trained neural network is feasible of detecting fruit categories with a noisy background or not is still undetermined. A dataset with multiple fruits in one image in addition to an ambient background is required for future improvement.

## **Bibliography**

[1]W. C. Seng and S. H. Mirisaee, “A new method for fruits recognition system,” *2009 International Conference on Electrical Engineering and Informatics*, 2009.

[2] S. Arivazhagan, R. Newlin Shebiah, S. Selva Nidhyanandhan, L. Ganesan, “ Fruit Recognition using Color and Texture Features”, Journal of Emerging Trends in Computing and Information Sciences, VOL. 1, NO. 2, pp. 90-94, 2010.

[3]Y. Zhang, S. Wang, G. Ji, and P. Phillips, “Fruit classification using computer vision and feedforward neural network,” *Journal of Food Engineering*, vol. 143, pp. 167–177, 2014.

[4]S. Wang, Y. Zhang, G. Ji, J. Yang, J. Wu, and L. Wei, “Fruit Classification by Wavelet-Entropy and Feedforward Neural Network Trained by Fitness-Scaled Chaotic ABC and Biogeography-Based Optimization,” *Entropy*, vol. 17, no. 12, pp. 5711–5728, 2015.

[5]Y.-D. Zhang, Z. Dong, X. Chen, W. Jia, S. Du, K. Muhammad, and S.-H. Wang, “Image based fruit category classification by 13-layer deep convolutional neural network and data augmentation,” *Multimedia Tools and Applications*, vol. 78, no. 3, pp. 3613–3632, 2017.

[6]H. Mureşan and M. Oltean, “Fruit recognition from images using deep learning,” *Acta Universitatis Sapientiae, Informatica*, vol. 10, no. 1, pp. 26–42, 2018.

[7]“Finetuning Torchvision Models,” Finetuning Torchvision Models - PyTorch Tutorials 1.0.0.dev20190327 documentation. [Online]. Available: https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html. [Accessed: 01-Apr-2019].

[8]Y. LeCun, *MNIST Demos on Yann LeCun's website*. [Online]. Available: http://yann.lecun.com/exdb/lenet/. [Accessed: 01-Apr-2019].

[9]A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[10] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” arXiv preprint arXiv:1409.1556, 2014.

[11]C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[12]C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[13]K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[14]G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[15]S. Pan and Q. Yang, "A Survey on Transfer Learning", *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345-1359, 2010. Available: 10.1109/tkde.2009.191.

[16]M. Oquab, L. Bottou, I. Laptev, and J. Sivic, “Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks,” *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[17]*ImageNet*. [Online]. Available: http://image-net.org/about-overview. [Accessed: 01-Apr-2019].

[18]S. Yegulalp, S. Yegulalp, and InfoWorld, “Facebook brings GPU-powered machine learning to Python,” *InfoWorld*, 19-Jan-2017. [Online]. Available: https://www.infoworld.com/article/3159120/facebook-brings-gpu-powered-machine-learning-to-python.html. [Accessed: 01-Apr-2019].

## Appendix: Code

```
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
if __name__ == '__main__':

    print("PyTorch Version: ",torch.__version__)
    print("Torchvision Version: ",torchvision.__version__)

    # Top level data directory. Here we assume the format of the directory conforms
    #   to the ImageFolder structure
    data_dir = "./data"

    # Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inception]
    # model_name = "resnet"
    model_name = "densenet"
    # model_name = "inception"

    # Number of classes in the dataset
    num_classes = 95

    # Batch size for training (change depending on how much memory you have)
    batch_size = 48

    # Number of epochs to train for
    num_epochs = 10

    # Flag for feature extracting. When False, we finetune the whole model,
    #   when True we only update the reshaped layer params
    feature_extract = True

    def train_model(model, dataloaders, criterion, optimizer, num_epochs=25,
is_inception=False):
        since = time.time()
    val_acc_history = []

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    # Get model outputs and calculate loss
                    # Special case for inception because in training it has an auxiliary output. In train
                    #   mode we calculate the loss by summing the final output and the auxiliary
output
                    #   but in testing we only consider the final output.
                    if is_inception and phase == 'train':
                        # From https://discuss.pytorch.org/t/how-to-optimize-inception-model-with-
auxiliary-classifiers/7958
                        outputs, aux_outputs = model(inputs)
                        loss1 = criterion(outputs, labels)
                        loss2 = criterion(aux_outputs, labels)
                        loss = loss1 + 0.4*loss2
                    else:
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)

                    _, preds = torch.max(outputs, 1)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

                # save best model
                print('Saving..')
                state = {
                    'model': model.state_dict(),
                    'acc': epoch_acc,
                    'epoch': epoch,
                }
                if not os.path.isdir('checkpoint'):
                    os.mkdir('checkpoint')
                torch.save(state, './checkpoint/ckpt_'+ model_name + '.t7')

            if phase == 'val':
                val_acc_history.append(epoch_acc)

        print()

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)

    return model, val_acc_history

def set_parameter_requires_grad(model, feature_extracting, model_name):
    if feature_extracting:
        for name, param in model.named_parameters():
            # for resnet:
            if model_name  == "resnet":
                if name.find("layer1") != -1 or name.find("layer2") != -1 or\
                        name == "conv1.weight" or name == "bn1.weight" or   \
                        name == "bn1.bias" :
                    param.requires_grad = False

            # for densenet
            elif model_name  == "densenet":
                if name.find("conv0") != -1 or name.find("norm0") != -1 or  \
                        name.find("denseblock1") != -1 or                   \
                        name.find("transition1") != -1 or                   \
                        name.find("denseblock2") != -1 or                   \
                        name.find("transition2") != -1 :
                    param.requires_grad = False

            # for inception net:
            elif model_name  == "inception":
                if name.find("Conv2d_") != -1:
                    param.requires_grad = False

def initialize_model(model_name, num_classes, feature_extract, use_pretrained=True):
    # Initialize these variables which will be set in this if statement. Each of these
    #   variables is model specific.
    model_ft = None
    input_size = 0

    if model_name == "resnet":
        """ Resnet18
        """
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract, model_name)
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "densenet":
            """ Densenet
            """
            model_ft = models.densenet121(pretrained=use_pretrained)
            set_parameter_requires_grad(model_ft, feature_extract, model_name)
            num_ftrs = model_ft.classifier.in_features
            model_ft.classifier = nn.Linear(num_ftrs, num_classes)
            input_size = 224

        elif model_name == "inception":
            """ Inception v3
            Be careful, expects (299,299) sized images and has auxiliary output
            """
            model_ft = models.inception_v3(pretrained=use_pretrained)
            set_parameter_requires_grad(model_ft, feature_extract, model_name)
            # Handle the auxilary net
            num_ftrs = model_ft.AuxLogits.fc.in_features
            model_ft.AuxLogits.fc = nn.Linear(num_ftrs, num_classes)
            # Handle the primary net
            num_ftrs = model_ft.fc.in_features
            model_ft.fc = nn.Linear(num_ftrs,num_classes)
            input_size = 299

        else:
            print("Invalid model name, exiting...")
            exit()

        return model_ft, input_size

    # Initialize the model for this run
    model_ft, input_size = initialize_model(model_name, num_classes, feature_extract,
use_pretrained=True)

    # Print the model we just instantiated
    # print(model_ft)

    # Data augmentation and normalization for training
    # Just normalization for validation
    data_transforms = {
        'train': transforms.Compose([
            transforms.RandomResizedCrop(input_size),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'val': transforms.Compose([
            transforms.Resize(input_size),
            transforms.CenterCrop(input_size),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }

    print("Initializing Datasets and Dataloaders...")

    # Create training and validation datasets
    image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for
x in ['train', 'val']}
    # Create training and validation dataloaders
    dataloaders_dict = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
shuffle=True, num_workers=4) for x in ['train', 'val']}

    # Detect if we have a GPU available
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    # Send the model to GPU
    model_ft = model_ft.to(device)

    # Gather the parameters to be optimized/updated in this run. If we are
    #  finetuning we will be updating all parameters. However, if we are
    #  doing feature extract method, we will only update the parameters
    #  that we have just initialized, i.e. the parameters with requires_grad
    #  is True.
    params_to_update = model_ft.parameters()
    print("Params to learn:")
    if feature_extract:
        params_to_update = []
        for name,param in model_ft.named_parameters():
            if param.requires_grad == True:
                params_to_update.append(param)
                print("\t",name)
    else:
        for name,param in model_ft.named_parameters():
            if param.requires_grad == True:
                print("\t",name)

    # Observe that all parameters are being optimized
    optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
    # Setup the loss fxn
    criterion = nn.CrossEntropyLoss()

    # Train and evaluate
    model_ft, hist = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft,
num_epochs=num_epochs, is_inception=(model_name=="inception"))

    # Plot the training curves of validation accuracy vs. number
    #  of training epochs for the transfer learning method and
    #  the model trained from scratch
    ohist = []

    ohist = [h.cpu().numpy() for h in hist]

    save_h = ",".join(str(h) for h in ohist)
    f = open("save.txt", "a")
    f.write(save_h)
    f.close()
```