COMP 499 - Final Project

Axel Bogos
Gina Cody School of Engineering
Concordia University
Montreal, QC
a_ogos@live.concordia.ca

Abstract

Artificial Intelligence, most notably through deep learning, has made tremendous advances in the last few years, achieving ever-higher performances on benchmark data sets and outperforming humans on complex tasks such as image classification on certain data sets. However, these performances heavily rely on large data sets for training: generalizing image classification from a few samples is a much harder task. We explore this problem through 2 challenges respectively consisting of learning on limited samples of CIFAR-10 (Krizhevsky and others 2009) without and with access to external data. We first review relevant literature on Few-shot Learning and more briefly review the structure and reasoning behind the VGG and Residual networks architecture. We then propose a simple architecture for learning without external data which outperforms a comparably heavier model and finally 2 models for few-shot learning with external data.

Introduction

This document details the two challenges part of the final project, the methods used as an attempt to solve these challenges and finally the respective quantitative results obtained through those methods. This project was completed as part of COMP 499 at Concordia University as an undergraduate student. All submissions on Codalab were made under the username **AxelB**. Due to unforeseen events, previous teammembers have had to drop this class; hence, this project was conducted alone. The common objective of these 2 challenges is to classify images collected from the CIFAR-10 with few training samples; albeit under different constraints in each challenge.

Literature Review

On Few-shot Learning (FSL)

In their work, Wang et al. present a formal definition of FSL and a thorough survey and taxonomy of current FSL methods and their respective issues. In recent years, the convergence of increasingly available computing power, large-scale data sets such as the aforementioned CIFAR-10 or ImageNet (Krizhevsky, Sutskever, and Hinton 2012) and the use of Convulational Neural Networks (CNNs), Long-short-term memory (LSTM) (Hochreiter and Schmidhuber 1997) or more recently the Tranformer architecture through Vision Tranformer (ViT) (Dosovitskiy et al. 2020) have allowed for great progress on tasks such as image classification. However, great performances on large-scale data sets do not necessarily translates into good generalization from a few examples. Acquiring and labelling such large data sets may be laborious, and in some cases impossible. For example, privacy concerns and little data availability are typical issues leading to a need for Few-Shot learning; both of these are common in domains like medical data analysis. Following the notation used by Wang et al., let us define a generic classification FSL problem as such:
let $D = \{D_{train}, D_{test}\}$, for $D_{train} = \{(x_i, y_i)\}_{i=1}^{I}$, $D_{test} = \{(x_i^{test}, y_i^{test})\}_{i=1}^{T}$, where I is small, $I << T$ and y_i^{test} is the ground-truth label of x_i^{test}. We will refer to a problem as an N-way-K-shot problem where $I = KN$ examples, that is to say K examples of N classes. Our goal is not unlike typical machine learning problems; that is we attempt to find hypothesis $\hat{h}$, where $\hat{h}$ is the optimal hypothesis from x to y. To do so, we optimize parameters θ such that the loss defined as $L(\hat{y}, y)$ is minimized over the predictions $\hat{y} = h(x, \theta)$ and ground-truths y. In order to achieve this despite having few training samples which reduces the *learnability* of θ, a number of methods are used. To name but a few, we consider augmenting the data, constraining the hypothesis search space (for example through multitask learning) or by modifying our search strategy (for instance through meta-learning). We give a simplified taxonomy of such methods in Figure 1. For the purpose of this project, we mainly explore augmenting data by transforming samples from the training set, multitask-learning and fine-tuning existing parameters. The reasons for these respective choices are further discussed in the *Challenge* sections, but we expand a bit on the respective processes here.

Transforming Samples from D_{train}: by applying affine transforms on a certain percentage of $(x_i, y_i) \in D_{train}$ batch-wise, we in effect augment our data set and are able to use the augmented data as prior knowledge to better classify the original samples (Miller, Matsakis, and Viola 2000). By extension, we also force the model to learn about invariance in the data domain. A particularly interesting meta-instance of such affine transformation procedure is the AutoAugment implementation showcased by Cubuk et al., where the augmentation procedure is itself learned on different data sets.

We make use of this method in Challenge 1.

Task Invariant Embedding Model: Embedding learning consists of embedding both x_{train} and x_{test} in a lower-dimension where they may be more easily discriminated. In short, the process involves learning a general embedding function from a large-scale data-set, embed D_{train} and the few samples D_{test} *without training*, and finally use a similarity metric or kNN classification on those embeddings to produce a prediction. While this method was considered and briefly explored for Challenge 2, the following method was preferred.

Fine-Tuning Existing Parameters with New parameters: Consider model m, trained on an external data set for which a set of good parameters θ_m has been found. Let $\theta_m = \theta_f + \theta_l$, where θ_f is the parameter set of feature layers and θ_l is the parameter set of the final linear classifier. Then we construct the new parameter set $\theta_{fsl} = \{\theta_f + \theta_{l'}\}$, where $\theta_{l'}$ is a new classifier we train on D_{train}. Therefore we take advantage of the pre-trained weight parameters by only concerning ourselves with learning a final classifier. Since this method also requires external data, we make use of it in Challenge 2.

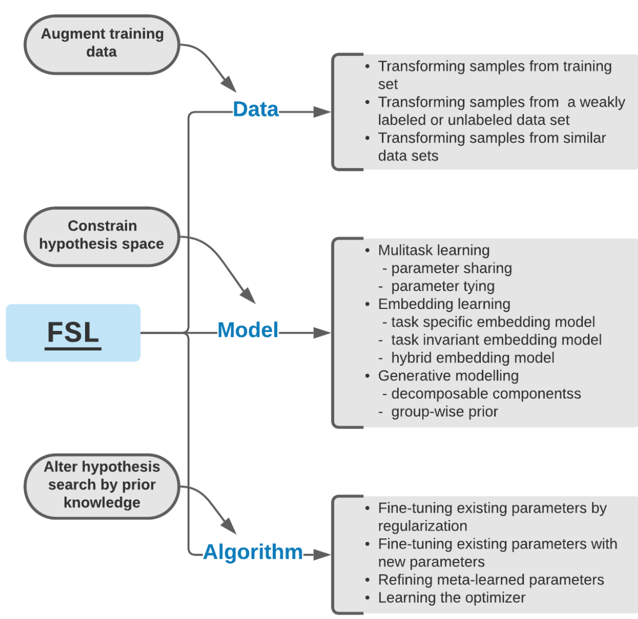


Figure 1: Taxonomy of FSL Methods. Inspired by Fig.3 of (Wang et al. 2020).

On the VGG Architecture

The VGG architecture, proposed by Simonyan and Zisserman in the 2015 paper *Very Deep Convolutional Networks for Large-Scale Image Recognition* achieved state-of-the-art performances on ImageNet in 2014. Despite not being in that position now, it remains an interesting architecture which will be further explored. It involves stacking 2 convolutional layers, each having a small 3x3 kernel followed by a max pooling layer. Each of these 3 layers unit may be referred to as a VGG block. Denominations such as VGG-16 or VGG-19 refer to the total number of weight layers, including the final linear layers.

On ResNet Models

Despite the success of the aforementioned AlexNet and VGG network architecture, it is clear that continuously increasing the depth of CNN networks is not a viable solution to increase the capacity of the network going forward; notably because of the vanishing/exploding gradient problem in deep networks. Residual Networks, presented in (He et al. 2015) introduce *identity shortcut connections* as a solution to this problem. By introducing an identity mapping between convolutional blocks, access is provided for the gradients to backpropagate throughout the network without vanishing through in the deep hidden layers. Consider a few hidden layers of a simple network with input x. Following the naming convention of He et al., let $H(x)$ be the mapping of x fitted by these hidden layers. Evidently, all gradients propagated to the first hidden layer of this set have already been propagated through all layers ahead of it. This may lead to both vanishing gradients as previously mentioned, or to what the authors refer to as the *degradation problem*, where the stacking of more non-linear layers undermines the estimation of identity mappings. As such, letting the hidden layers equivalently estimate a mapping of the residual function $H(x) - x$ and introducing a new identity mapping $+x$ connection (or shortcut) onto the next layer allows the optimization process to both backpropagate gradients through the identity mapping shortcuts and estimate the identity mapping itself if need be.

Consequently, this architecture allows the use of much deeper networks than the VGG architecture, up to 152 layers in the case of ResNet-152. Each residual block is a sequence of multiple convolutional layers with a small kernel, feeding into each other both the convolutional output and the notable shortcut identity mapping. A depiction of an abstract residual block is shown in Figure 2

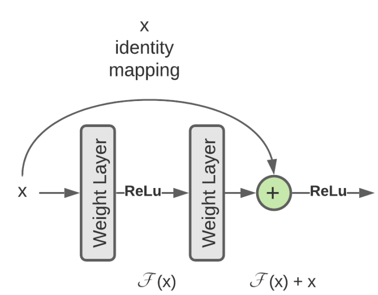


Figure 2: Residual learning block. Recreation of Fig.2 of (He et al. 2015)

Challenge 1

Overview

Challenge 1 consists of a 10-way-10-shot learning challenge on 100 samples of the test set of CIFAR-10. Based on the initial baseline model, we expand on and explore 2 main modelling architecture and a number of tuning and augmentation methods. We further detail the 2 main architectures explored and their justification below. Our first architecture is built upon the original baseline, while our second architecture makes use of VGG (Simonyan and Zisserman 2015) blocks in what we will refer to as a *VGG-like* architecture. We finally briefly mention other methods that have been considered but not fully explored.

Architectures Explored

4 Block Convnet Our first attempt on at developing a model architecture is based on the baseline. As pointed out in (Hasanpour et al. 2018), simple architectures may achieve similar performances as heavily-parameterized ones such as deep VGG nets. Furthermore, such models are also much less computationally heavy and hence allow for more tuning under limited hardware availability. In addition to that, highly parameterized architectures are subject to more risks of over-fitting, particularly in the context of few-shots learning. Hence, we begin by first exploring a simple convolutional network architecture. This first network architecture is shown in Figure 3. It is made of 4 convolution layers, each of which is batch-normalized. A max pool layer is used in-between convolution layers. Finally, an average pooling and a linear layer are used for classification. This architecture provides 113,738 weight parameters. A comparison of the number of parameters in different models can be found in Table 2.

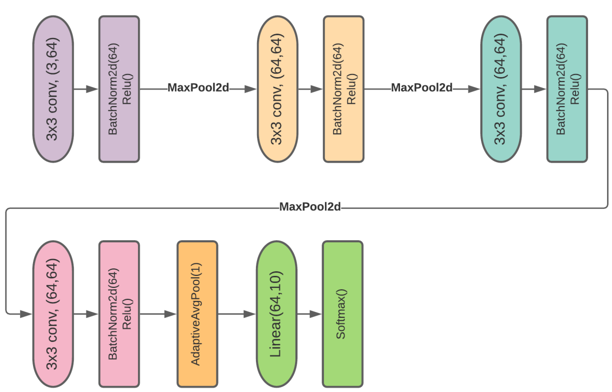


Figure 3: ConvNet Architecture - Challenge 1

3 VGG Blocks Our second proposed architecture is based on the VGG architecture proposed by Simonyan and Zisserman. It is composed of 3 VGG-blocks. Each of these block is composed of 2 stacked convolution layer batch-normalized between each convolution and a max pool layer. Finally, unlike typical VGG-architecture, we use 2 linear layers for the classification. The architecture is represented in Figure 4. This architecture has a total of 4,646,922 weight parameters (see Table 2).

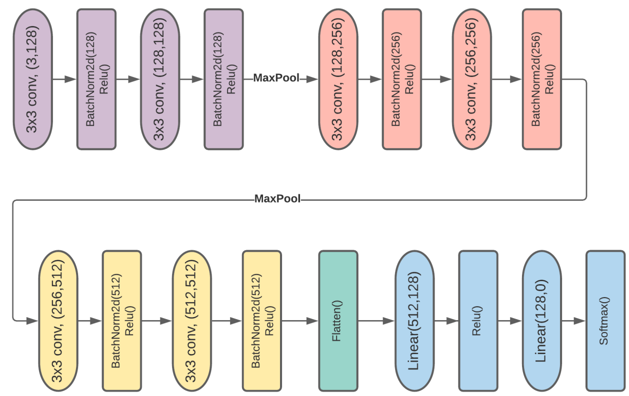


Figure 4: 3 VGG block Architecture - Challenge 1

Methods

Our general approach to FSL in challenge 1 is augment data as much as possible and prevent over-fitting. A shared data augmentation method has been found to be effective with both architectures. The data augmentation makes use of the following *torchvision* transforms:

- transforms.RandomHorizontalFlip()
- transforms.RandomCrop(size=[32,32], padding=4,fill=128)
- CIFAR10Policy()
- transforms.ToTensor()
- transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

where *CIFAR10Policy()* is an implementation of the CIFAR-10 augmentation policy learned by AutoAugment. By combining these augmentation policy, we are able to artificially expand our training space state search. Since by randomly augmenting the data we presumably encounter multiple new training examples each epoch, we use a warm-restart scheduler (from torch.optim.lr_scheduler.CosineAnnealingLR) to bring back our learning rate to its maximum value on each epoch. For all experiments we ran, we used Stochastic Gradient Descent as the optimizer with Cross-Entropy as the loss function. For each architecture, a manual binary search of hyper-parameters was conducted for the initial learning rate and weight decay. For each parameter run, 10 runs over different random partitions were averaged in order to evaluate their impact on the accuracy despite high variability. The result of the parameter search on the learning rate for both architectures can be found in Table 1. The weight decay was found to be best left to the initial value for both methods. While adding dropout layers was considered and evaluated, their impact on the accuracy was overall negative. We hypothesize that given the relatively low number of learnable parameters and large data augmentation, such dropout layers were not warranted to prevent over-fitting and were instead an unnecessary hurdle to the learning. Each of the models were trained using a batch size of 128. An overview of all hyper-parameters can be found in Table 4 in the Appendix.

Results

Table 1: Models' Accuracy over 10 Random Partitions

Learning Rate	4Block ConvNet Acc.	3Block VGG Acc.
0.01	26.64 $\pm$1.84	31.65 $\pm$ 3.27
0.05	32.63 $\pm$3.29	**34.86** $\pm$ 2.65
0.1	32.70 $\pm$2.31	33.37 $\pm$ 2.45
0.2	**34.91** $\pm$2.3	32.83 $\pm$ 2.41
0.25	33.19 $\pm$2.70	32.49 $\pm$ 2.52
0.15	33.03 $\pm$3.83	33.93 $\pm$ 2.31
0.175	33.35 $\pm$2.44	33.34 $\pm$ 2.40
0.1875	33.39 $\pm$3.33	32.60 $\pm$ 3.15

Our overall results in terms of accuracy are shown in Table 1. We remark that both models' accuracy under the best learning rates are very similar, well within their standard deviation. This goes to show that the 4 block ConvNet, despite having much less learnable parameters (40x less!), performs at a similar level as the VGG-like architecture. The simpler network architecture is also faster to train over 150 epochs. Under their best learning rates and on a Google Colab GPU (Tesla P-100), the 3 Block VGG took $8.11s \pm 0.08$, the 4 block ConvNet took $7.29s \pm 0.05$ and the baseline $2.47s \pm 0.04$, making the former respectively 3.28x and 2.95x slower than the baseline. Finally, we also point out that there is a discrepancy between our best result on the final random-partition of the test set of CIFAR-10 and our best score on Codalab (0.309), which was done with the 4 block ConvNet. This might be explicable by a weaker generalization on another test set, if it is the case the one is used for the evaluation on Codalab. Henceforth, we believe that the next steps to be taken in order to better our performances would not necessarily be towards bettering our network architecture but better fine-tuning the data augmentation, scheduling and regularization. Such improvements may not only ameliorate performances over a training of 150 epochs, but also possibly allow for a longer training (since it would potentially reduce any over-fitting side effect). Further supporting this intuition is that initial tests using deeper networks such as a Wide-ResNet on this challenge performed significantly worse than our simple 4 block ConvNet, although an extensive tuning was not conducted. Finally we briefly mention methods that have been considered, but not explored to their fullest

GLICO data augmentation While initially thinking that augmenting the data of an FSL problem with a generative model presented a circularity issue (one would assume training such a model with so few samples is as a hard a problem as the original FSL problem), the GLICO (Azuri and Weinshall 2020) architecture seems very promising for the purpose of augmenting few-samples data-sets in the context of a FSL problem. Due to time constraints (and having found this method late into the work), it has been set aside as a future endeavour.

Table 2: Number of Parameters of proposed architectures and other architectures for reference

Model	Number of Parameters
4 block ConvNet	113,738
3 VGG block	4,646,922
AlexNet	60M
VGG16	138M

Challenge 2

Overview

In Challenge 2, we were asked to reconsider challenge 1, but this time with the ability to "use external data or models not trained on CIFAR-10." With the addition of this constraint, our new approach was to experiment with transfer learning using the VGG-11 and ResNet-18 models pretrained on Imagenet. Using these two models provided by the Pytorch Torchvision Model Repository, the goal was to then finetune the final linear layer of each model on the few-sample CIFAR-10 data as was previously described in our taxonomy of FSL methods. By starting with pretrained feature layers instead of training from scratch, we hope a significantly higher test-accuracy is achievable in similar training time by making use of the pretrained weights on a similar task as CIFAR-10.

Methodology

The approach we took was the same for both of the chosen pretrained models. We first downloaded the model from the PyTorch model repository and reshaped the final layer such that the number of outputs matched the number of target dataset classes, which in this case was 10. This means that we are removing the final fully-connected layer and using the rest of the pretrained network as fixed feature extractor for our CIFAR-10 dataset. For all experiments we ran, we used a Stochastic Gradient Descent as the optimizer with Cross-Entropy as the loss function. Finally we trained each model for 20 epochs with 5 different learning rates ranging from 0.001 to 0.05.

Architectures Explored

Our choice of model from which to operate transfer learning is two-fold: firstly, each of them have well documented, robust perfomance on ImageNet, secondly, both of them are much deeper than networks explored in Challenge 1 while staying easy to handle on limited hardware.

ResNet-18 ResNet was first introduced by He et al. in 2015 and became an extremely successful architecture for image classification and related tasks. As discussed before, what makes this architecture so successful is the introduction of the residual blocks which help to solve the problem of vanishing gradients in deep neural networks. The ResNet-18 architecture consists of 18 layers, upon which the skip-connections are added, creating 8 residual blocks within the network. While larger ResNet architectures such as ResNet-34 or ResNet-50 were also viable options, we have chosen the smaller ResNet-18 due to the low class count in CIFAR-10. By choosing a relatively smaller architecture, we can also achieve lower parameter count and lower computational expense. As per He et al., ResNet-18 has a top-1 error rate of 27.88% on ImageNet, for an accuracy of 72.12%. ResNet18 has about 11M trainable parameters, which is nearly 27.5x as many as our best performing architecture in Challenge 1. While this is quite a high parameter count, in this case it is not an issue due to the fact that we are using a pretrained model as opposed to training from scratch. An overview of the ResNet-18 structure is shown in Table 5 in the Appendix.

VGG-11 bn The VGG-11 architecture makes use of VGG blocks as previously described, where each block is 2 convolutional layer stacked followed by a maxpool. Like the ResNet-18, we have chosen a relatively smaller version of the available VGG pretrained network (for instance VGG-16 or VGG-19) due to the low class count of CIFAR-10 and computational cost. As per PyTorch documentation [1], VGG-11 batch normalized has a top-1 error rate of 26.70% on ImageNet, for an accuracy of 73.3%. An overview of the VGG-11 bn architecture is shown in Table 6.

Results

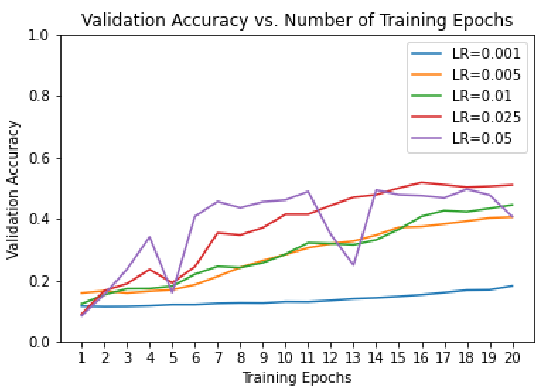


Figure 5: ResNet-18 Finetuning Accuracies - Challenge 2

We observe that in general, transfer learning from the VGG-11 model was more successful. Looking at figure 6 above, we see that when finetuning the VGG model, we were consistently reaching top-accuracy in only a few epochs. This means that while we trained for 20 epochs, with a higher learning rate it would only be necessary to train for 5-10 epochs to achieve similar results. Under a single final testing regime where the learning rate = 0.05 (incidentally also our best learning rate for our VGG-Like architecture in Challenge 1), the VGG-11 model was able to attain a mean validation accuracy over three runs of 60.8%. We also observe that despite being higher than our results in Challenge 1, our best result here fall short of performances of a single model such as ResNet18 or VGG-11 trained on a large set of either CIFAR-10 or ImageNet. A difficulty encountered in this challenge was the relative lengthy and resource consuming process of hyper-parameter tuning. Nevertheless, our method is quite simple and fast to train given the right hardware. Although embedding learning has been considered, initial tests provided poor accuracy and it was decided not to further explore this method given time and resource constraints.

[1] https://pytorch.org/hub/pytorch_vision_vgg/

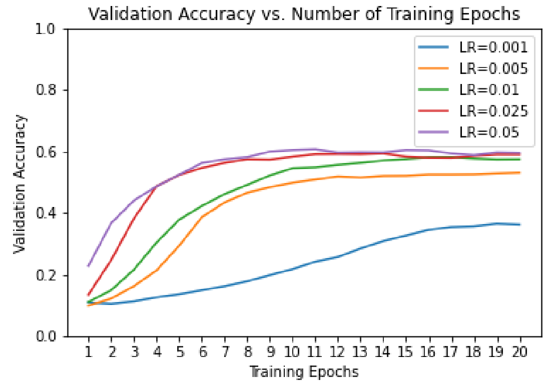


Figure 6: VGG-11 Finetuning Accuracies - Challenge 2

Table 3: Mean Top-Accuracies After 20 Epochs

Learning Rate	VGG-11	ResNet-18
0.001	36.15	18.15
0.005	53.0	40.55
0.01	58.35	44.50
0.025	58.90	51.0
0.05	60.80	49.60

References

Azuri, I., and Weinshall, D. 2020. Generative latent implicit conditional optimization when learning from small sample.

Cubuk, E. D.; Zoph, B.; Mane, D.; Vasudevan, V.; and Le, Q. V. 2019. Autoaugment: Learning augmentation policies from data.

Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; Uszkoreit, J.; and Houlsby, N. 2020.

An image is worth 16x16 words: Transformers for image recognition at scale.

Hasanpour, S. H.; Rouhani, M.; Fayyaz, M.; and Sabokrou, M. 2018. Lets keep it simple, using simple architectures to outperform deeper and more complex architectures.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep residual learning for image recognition.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Krizhevsky, A., et al. 2009. Learning multiple layers of features from tiny images.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25:1097–1105.

Miller, E.; Matsakis, N.; and Viola, P. 2000. Learning from one example through shared densities on transforms. volume 1, 464 – 471 vol.1.

Simonyan, K., and Zisserman, A. 2015. Very deep convolutional networks for large-scale image recognition.

Wang, Y.; Yao, Q.; Kwok, J. T.; and Ni, L. M. 2020. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)* 53(3):1–34.

Appendix

Table 4: Challenge 1 Models Best Hyper-parameters'

Hyper param.	4Block ConvNet Acc.	3Block VGG Acc.
Batch size	128	128
Optimizer	SGD	SGD
Learning rate	0.2	0.05
Weight decay	0.0005	0.0005
Momentum	0.9	0.9
Number of epochs	150	150
Scheduler	CosineAnnealingLR	CosineAnnealingLR

Table 5: ResNet-18 Architecture

Layer Name	Parameters
conv1	7x7, 64, stride 2
max pool	3x3, stride 2
conv2.1	3x3, 64
conv2.2	3x3, 64
conv3.1	3x3, 128
conv3.2	3x3, 128
conv4.1	3x3, 256
conv4.2	3x3, 256
conv5.1	3x3, 256
conv5.2	3x3, 256
average pool, fc, softmax	

Table 6: VGG-11 Architecture

Layer Name	Parameters
conv	3x3, 64
max pool	3x3
conv	3x3, 128
max pool	3x3
conv	3x3, 256
conv	3x3, 256
max pool	3x3
conv	3x3, 512
conv	3x3, 512
max pool	3x3
conv	3x3, 512
conv	3x3, 512
max pool	3x3
FC	4096
FC	4096
FC	1000
softmax	

Challenge1

May 3, 2021

[11]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from numpy.random import RandomState
import numpy as np
import torch
import torch.optim as optim
from torch.utils.data import Subset
from torchvision import datasets, transforms
import math
from PIL import Image, ImageEnhance, ImageOps
import random
import timeit
```

Define functions to return datasets & dataloaders and execute train and test

[12]:
```
def load_data_set(transform_train, transform_test):
  # Final submission, use partitions of the test data set
  cifar_data_train = datasets.CIFAR10(root='.',train=False,␣
↪transform=transform_train, download=True)
  cifar_data_test = datasets.CIFAR10(root='.',train=False,␣
↪transform=transform_val, download=True)

  return cifar_data_train, cifar_data_test

def get_data_loaders(train_data,test_data, random_seed, batch_size):
    prng = RandomState(random_seed)
    random_permute = prng.permutation(np.arange(0, 1000))
    indx_train = np.concatenate([np.where(np.array(train_data.targets) ==␣
↪classe)[0][random_permute[0:10]] for classe in range(0, 10)])
    indx_val = np.concatenate([np.where(np.array(test_data.targets) ==␣
↪classe)[0][random_permute[10:210]] for classe in range(0, 10)])


    train_data = Subset(train_data, indx_train)
    val_data = Subset(test_data, indx_val)
```

```
    print('Num Samples For Training %d Num Samples For Val %d'%(train_data.
↪indices.shape[0],val_data.indices.shape[0]))

    train_loader = torch.utils.data.DataLoader(train_data,
                                               batch_size=batch_size,
                                               shuffle=True)

    val_loader = torch.utils.data.DataLoader(val_data,
                                             batch_size=batch_size,
                                             shuffle=False)
    return train_loader, val_loader

def train(model, device, train_loader, optimizer, epoch, display):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        #scheduler.step()
    if display:
      print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
          epoch, batch_idx * len(data), len(train_loader.dataset),
          100. * batch_idx / len(train_loader), loss.item()))

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target, size_average=False).
↪item() # sum up batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max␣
↪log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.
↪format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    return 100. * correct / len(test_loader.dataset)
```

Define the AutoAugment policy. This cannot be imported as of now.

```
[13]: class CIFAR10Policy(object):
    '''
    Unofficial implementation of the CIFAR10 Augmentation Policies learned by␣
↪AutoAugment,
    described in this Google AI Blogpost (https://ai.googleblog.com/2018/06/
↪improving-deep-learning-performance.html).

    The implementation is taken from this repository (https://github.com/
↪DeepVoltaire/AutoAugment)
    '''
    def __init__(self, fillcolor=(128, 128, 128)):
        self.policies = [
            SubPolicy(0.1, "invert", 7, 0.2, "contrast", 6, fillcolor),
            SubPolicy(0.7, "rotate", 2, 0.3, "translateX", 9, fillcolor),
            SubPolicy(0.8, "sharpness", 1, 0.9, "sharpness", 3, fillcolor),
            SubPolicy(0.5, "shearY", 8, 0.7, "translateY", 9, fillcolor),
            SubPolicy(0.5, "autocontrast", 8, 0.9, "equalize", 2, fillcolor),

            SubPolicy(0.2, "shearY", 7, 0.3, "posterize", 7, fillcolor),
            SubPolicy(0.4, "color", 3, 0.6, "brightness", 7, fillcolor),
            SubPolicy(0.3, "sharpness", 9, 0.7, "brightness", 9, fillcolor),
            SubPolicy(0.6, "equalize", 5, 0.5, "equalize", 1, fillcolor),
            SubPolicy(0.6, "contrast", 7, 0.6, "sharpness", 5, fillcolor),

            SubPolicy(0.7, "color", 7, 0.5, "translateX", 8, fillcolor),
            SubPolicy(0.3, "equalize", 7, 0.4, "autocontrast", 8, fillcolor),
            SubPolicy(0.4, "translateY", 3, 0.2, "sharpness", 6, fillcolor),
            SubPolicy(0.9, "brightness", 6, 0.2, "color", 8, fillcolor),
            SubPolicy(0.5, "solarize", 2, 0.0, "invert", 3, fillcolor),

            SubPolicy(0.2, "equalize", 0, 0.6, "autocontrast", 0, fillcolor),
            SubPolicy(0.2, "equalize", 8, 0.6, "equalize", 4, fillcolor),
            SubPolicy(0.9, "color", 9, 0.6, "equalize", 6, fillcolor),
            SubPolicy(0.8, "autocontrast", 4, 0.2, "solarize", 8, fillcolor),
            SubPolicy(0.1, "brightness", 3, 0.7, "color", 0, fillcolor),

            SubPolicy(0.4, "solarize", 5, 0.9, "autocontrast", 3, fillcolor),
            SubPolicy(0.9, "translateY", 9, 0.7, "translateY", 9, fillcolor),
            SubPolicy(0.9, "autocontrast", 2, 0.8, "solarize", 3, fillcolor),
            SubPolicy(0.8, "equalize", 8, 0.1, "invert", 3, fillcolor),
            SubPolicy(0.7, "translateY", 9, 0.9, "autocontrast", 1, fillcolor)
        ]


    def __call__(self, img):
```

```
        policy_idx = random.randint(0, len(self.policies) - 1)
        return self.policies[policy_idx](img)

    def __repr__(self):
        return "AutoAugment CIFAR10 Policy"

class SubPolicy(object):
    '''
    Unofficial implementation of the CIFAR10 Augmentation Policies learned by␣
↪AutoAugment,
    described in this Google AI Blogpost (https://ai.googleblog.com/2018/06/
↪improving-deep-learning-performance.html).

    The implementation is taken from this repository (https://github.com/
↪DeepVoltaire/AutoAugment)
    '''
    def __init__(self, p1, operation1, magnitude_idx1, p2, operation2,␣
↪magnitude_idx2, fillcolor=(128, 128, 128)):
        ranges = {
            "shearX": np.linspace(0, 0.3, 10),
            "shearY": np.linspace(0, 0.3, 10),
            "translateX": np.linspace(0, 150 / 331, 10),
            "translateY": np.linspace(0, 150 / 331, 10),
            "rotate": np.linspace(0, 30, 10),
            "color": np.linspace(0.0, 0.9, 10),
            "posterize": np.round(np.linspace(8, 4, 10), 0).astype(np.int),
            "solarize": np.linspace(256, 0, 10),
            "contrast": np.linspace(0.0, 0.9, 10),
            "sharpness": np.linspace(0.0, 0.9, 10),
            "brightness": np.linspace(0.0, 0.9, 10),
            "autocontrast": [0] * 10,
            "equalize": [0] * 10,
            "invert": [0] * 10
        }

        # from https://stackoverflow.com/questions/5252170/
↪specify-image-filling-color-when-rotating-in-python-with-pil-and-setting-expand
        def rotate_with_fill(img, magnitude):
            rot = img.convert("RGBA").rotate(magnitude)
            return Image.composite(rot, Image.new("RGBA", rot.size, (128,) *␣
↪4), rot).convert(img.mode)

        func = {
            "shearX": lambda img, magnitude: img.transform(
                img.size, Image.AFFINE, (1, magnitude * random.choice([-1, 1]),␣
↪0, 0, 1, 0),
```

```
                 Image.BICUBIC, fillcolor=fillcolor),
             "shearY": lambda img, magnitude: img.transform(
                 img.size, Image.AFFINE, (1, 0, 0, magnitude * random.
↪choice([-1, 1]), 1, 0),
                 Image.BICUBIC, fillcolor=fillcolor),
             "translateX": lambda img, magnitude: img.transform(
                 img.size, Image.AFFINE, (1, 0, magnitude * img.size[0] * random.
↪choice([-1, 1]), 0, 1, 0),
                 fillcolor=fillcolor),
             "translateY": lambda img, magnitude: img.transform(
                 img.size, Image.AFFINE, (1, 0, 0, 0, 1, magnitude * img.size[1]␣
↪* random.choice([-1, 1])),
                 fillcolor=fillcolor),
             "rotate": lambda img, magnitude: rotate_with_fill(img, magnitude),
             "color": lambda img, magnitude: ImageEnhance.Color(img).enhance(1 +␣
↪magnitude * random.choice([-1, 1])),
             "posterize": lambda img, magnitude: ImageOps.posterize(img,␣
↪magnitude),
             "solarize": lambda img, magnitude: ImageOps.solarize(img,␣
↪magnitude),
             "contrast": lambda img, magnitude: ImageEnhance.Contrast(img).
↪enhance(
                 1 + magnitude * random.choice([-1, 1])),
             "sharpness": lambda img, magnitude: ImageEnhance.Sharpness(img).
↪enhance(
                 1 + magnitude * random.choice([-1, 1])),
             "brightness": lambda img, magnitude: ImageEnhance.Brightness(img).
↪enhance(
                 1 + magnitude * random.choice([-1, 1])),
             "autocontrast": lambda img, magnitude: ImageOps.autocontrast(img),
             "equalize": lambda img, magnitude: ImageOps.equalize(img),
             "invert": lambda img, magnitude: ImageOps.invert(img)
         }

         self.p1 = p1
         self.operation1 = func[operation1]
         self.magnitude1 = ranges[operation1][magnitude_idx1]
         self.p2 = p2
         self.operation2 = func[operation2]
         self.magnitude2 = ranges[operation2][magnitude_idx2]


     def __call__(self, img):
         if random.random() < self.p1: img = self.operation1(img, self.
↪magnitude1)
```

```
        if random.random() < self.p2: img = self.operation2(img, self.
↪magnitude2)
        return img
```

[18]:
```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])

transform_val = transforms.Compose([transforms.ToTensor(), normalize]) #careful␣
↪to keep this one same

transform_train = transforms.Compose([
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomCrop(size=[32,32],␣
↪padding=4, fill=128),
                                      CIFAR10Policy(),
                                      transforms.ToTensor(),
                                      normalize])
##### Load Cifar Data
cifar_data_train,cifar_data_test = load_data_set(transform_train,transform_val)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

Define all our models!

[19]:
```
class ConvNet(nn.Module):

    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2))

        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2))
```

```
        self.layer4 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU())

        self.avgpool = nn.AdaptiveAvgPool2d(1)

        self.classifier = nn.Linear(64, 10)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
↪nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = out.view(out.size(0), -1)

        out = self.classifier(out)
        return out
```

[20]:
```
class VGG_Net(torch.nn.Module):
    def __init__(self, init_weights=False):
        super(VGG_Net, self).__init__()

        self.VGG_block1 = nn.Sequential(
        nn.Conv2d(3, 128,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=128),
        nn.ReLU(inplace=True),
        nn.Conv2d(128, 128,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=128),
        nn.ReLU(inplace=True),
        nn.MaxPool2d((2,2)),
        nn.Dropout(0.2)
        )

        self.VGG_block2 = nn.Sequential(
        nn.Conv2d(128, 256,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=256),
        nn.ReLU(inplace=True),
```

```
        nn.Conv2d(256, 256,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=256),
        nn.ReLU(inplace=True),
        nn.MaxPool2d((2,2)),
        nn.Dropout(0.2)
        )

        self.VGG_block3 = nn.Sequential(
        nn.Conv2d(256, 512,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=512),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512,  kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=512),
        nn.ReLU(inplace=True),
        nn.Dropout(0.2)
        )
        self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(512,128),
        nn.ReLU(),
        nn.Linear(128,10))

        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        x = self.VGG_block1(x)
        x = self.VGG_block2(x)
        x = self.VGG_block3(x)
        x = self.classifier(x)
        return x

    # Somehow this seems to make accuracy worst
    def _initialize_weights(self):
      for m in self.modules():
          if isinstance(m, nn.Conv2d):
              nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
↪nonlinearity='relu')
          elif isinstance(m, nn.BatchNorm2d):
              nn.init.constant_(m.weight, 1)
              nn.init.constant_(m.bias, 0)
```

Final test on 4 block ConvNet

[21]:
```
accs = []
times = []
EPOCHS = 150
```

```python
learning_rate = 0.2
for seed in [1,2,3,4,5,6,7,8,9,10]:
  train_loader, val_loader =␣
 ↪get_data_loaders(cifar_data_train,cifar_data_test,seed,128)

  model = ConvNet()
  model.to(device)
  optimizer = torch.optim.SGD(model.parameters(),
                              lr=learning_rate, momentum=0.9,
                              weight_decay=0.0005)
  duration = learning_rate * (0.005 ** 3)
  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, EPOCHS,␣
 ↪duration, -1)
  start_time = timeit.default_timer()
  for epoch in range(EPOCHS):
    display_bool  = epoch%10==0
    train(model, device, train_loader, optimizer, epoch, display=display_bool)
    scheduler.step()
  elapsed = timeit.default_timer() - start_time
  times.append(elapsed)
  accs.append(test(model, device, val_loader))

accs = np.array(accs)
times = np.array(times)
print('Acc over 10 instances: %.2f +- %.2f'%(accs.mean(),accs.std()))
print('Training time over 10 instances: %.2f +- %.2f'%(times.mean(),times.
 ↪std()))
```

```
Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.328881
Train Epoch: 10 [0/100 (0%)]    Loss: 2.071114
Train Epoch: 20 [0/100 (0%)]    Loss: 1.804456
Train Epoch: 30 [0/100 (0%)]    Loss: 1.675893
Train Epoch: 40 [0/100 (0%)]    Loss: 1.446929
Train Epoch: 50 [0/100 (0%)]    Loss: 1.339385
Train Epoch: 60 [0/100 (0%)]    Loss: 1.156281
Train Epoch: 70 [0/100 (0%)]    Loss: 1.228063
Train Epoch: 80 [0/100 (0%)]    Loss: 1.076820
Train Epoch: 90 [0/100 (0%)]    Loss: 0.767182
Train Epoch: 100 [0/100 (0%)]   Loss: 0.816218
Train Epoch: 110 [0/100 (0%)]   Loss: 0.860247
Train Epoch: 120 [0/100 (0%)]   Loss: 0.627976
Train Epoch: 130 [0/100 (0%)]   Loss: 0.723445
Train Epoch: 140 [0/100 (0%)]   Loss: 0.554375
```

```
/usr/local/lib/python3.7/dist-packages/torch/nn/_reduction.py:42: UserWarning:
size_average and reduce args will be deprecated, please use reduction='sum'
instead.
```

```
  warnings.warn(warning.format(ret))

Test set: Average loss: 2.2185, Accuracy: 665/2000 (33.25%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.302792
Train Epoch: 10 [0/100 (0%)]    Loss: 2.089259
Train Epoch: 20 [0/100 (0%)]    Loss: 1.747485
Train Epoch: 30 [0/100 (0%)]    Loss: 1.601876
Train Epoch: 40 [0/100 (0%)]    Loss: 1.527862
Train Epoch: 50 [0/100 (0%)]    Loss: 1.490015
Train Epoch: 60 [0/100 (0%)]    Loss: 1.329051
Train Epoch: 70 [0/100 (0%)]    Loss: 1.032516
Train Epoch: 80 [0/100 (0%)]    Loss: 1.009270
Train Epoch: 90 [0/100 (0%)]    Loss: 0.857287
Train Epoch: 100 [0/100 (0%)]   Loss: 0.867318
Train Epoch: 110 [0/100 (0%)]   Loss: 0.805307
Train Epoch: 120 [0/100 (0%)]   Loss: 0.654153
Train Epoch: 130 [0/100 (0%)]   Loss: 0.640020
Train Epoch: 140 [0/100 (0%)]   Loss: 0.659104

Test set: Average loss: 2.2462, Accuracy: 659/2000 (32.95%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.321607
Train Epoch: 10 [0/100 (0%)]    Loss: 2.075875
Train Epoch: 20 [0/100 (0%)]    Loss: 1.747978
Train Epoch: 30 [0/100 (0%)]    Loss: 1.671037
Train Epoch: 40 [0/100 (0%)]    Loss: 1.437350
Train Epoch: 50 [0/100 (0%)]    Loss: 1.325777
Train Epoch: 60 [0/100 (0%)]    Loss: 1.183336
Train Epoch: 70 [0/100 (0%)]    Loss: 0.975102
Train Epoch: 80 [0/100 (0%)]    Loss: 0.945320
Train Epoch: 90 [0/100 (0%)]    Loss: 0.788731
Train Epoch: 100 [0/100 (0%)]   Loss: 0.777531
Train Epoch: 110 [0/100 (0%)]   Loss: 0.838944
Train Epoch: 120 [0/100 (0%)]   Loss: 0.677615
Train Epoch: 130 [0/100 (0%)]   Loss: 0.685333
Train Epoch: 140 [0/100 (0%)]   Loss: 0.619270

Test set: Average loss: 2.2205, Accuracy: 695/2000 (34.75%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.342828
Train Epoch: 10 [0/100 (0%)]    Loss: 1.925964
Train Epoch: 20 [0/100 (0%)]    Loss: 1.723845
Train Epoch: 30 [0/100 (0%)]    Loss: 1.600801
```

```
Train Epoch: 40 [0/100 (0%)]    Loss: 1.428645
Train Epoch: 50 [0/100 (0%)]    Loss: 1.170405
Train Epoch: 60 [0/100 (0%)]    Loss: 1.144084
Train Epoch: 70 [0/100 (0%)]    Loss: 1.228412
Train Epoch: 80 [0/100 (0%)]    Loss: 0.858264
Train Epoch: 90 [0/100 (0%)]    Loss: 0.770229
Train Epoch: 100 [0/100 (0%)]   Loss: 0.820433
Train Epoch: 110 [0/100 (0%)]   Loss: 0.786189
Train Epoch: 120 [0/100 (0%)]   Loss: 0.647721
Train Epoch: 130 [0/100 (0%)]   Loss: 0.588705
Train Epoch: 140 [0/100 (0%)]   Loss: 0.588794

Test set: Average loss: 2.2159, Accuracy: 715/2000 (35.75%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.326327
Train Epoch: 10 [0/100 (0%)]    Loss: 1.996611
Train Epoch: 20 [0/100 (0%)]    Loss: 1.671213
Train Epoch: 30 [0/100 (0%)]    Loss: 1.432489
Train Epoch: 40 [0/100 (0%)]    Loss: 1.376742
Train Epoch: 50 [0/100 (0%)]    Loss: 1.179493
Train Epoch: 60 [0/100 (0%)]    Loss: 1.143770
Train Epoch: 70 [0/100 (0%)]    Loss: 0.872340
Train Epoch: 80 [0/100 (0%)]    Loss: 0.918779
Train Epoch: 90 [0/100 (0%)]    Loss: 0.973632
Train Epoch: 100 [0/100 (0%)]   Loss: 0.893715
Train Epoch: 110 [0/100 (0%)]   Loss: 0.707223
Train Epoch: 120 [0/100 (0%)]   Loss: 0.598746
Train Epoch: 130 [0/100 (0%)]   Loss: 0.799662
Train Epoch: 140 [0/100 (0%)]   Loss: 0.730001

Test set: Average loss: 2.5009, Accuracy: 648/2000 (32.40%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.341781
Train Epoch: 10 [0/100 (0%)]    Loss: 1.932215
Train Epoch: 20 [0/100 (0%)]    Loss: 1.625332
Train Epoch: 30 [0/100 (0%)]    Loss: 1.532589
Train Epoch: 40 [0/100 (0%)]    Loss: 1.330155
Train Epoch: 50 [0/100 (0%)]    Loss: 1.269889
Train Epoch: 60 [0/100 (0%)]    Loss: 1.020123
Train Epoch: 70 [0/100 (0%)]    Loss: 1.072837
Train Epoch: 80 [0/100 (0%)]    Loss: 0.870932
Train Epoch: 90 [0/100 (0%)]    Loss: 0.865527
Train Epoch: 100 [0/100 (0%)]   Loss: 0.822930
Train Epoch: 110 [0/100 (0%)]   Loss: 0.715871
Train Epoch: 120 [0/100 (0%)]   Loss: 0.627241
Train Epoch: 130 [0/100 (0%)]   Loss: 0.718668
```

```
Train Epoch: 140 [0/100 (0%)]   Loss: 0.502314

Test set: Average loss: 2.1354, Accuracy: 751/2000 (37.55%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.333476
Train Epoch: 10 [0/100 (0%)]    Loss: 2.107597
Train Epoch: 20 [0/100 (0%)]    Loss: 1.679236
Train Epoch: 30 [0/100 (0%)]    Loss: 1.615218
Train Epoch: 40 [0/100 (0%)]    Loss: 1.352714
Train Epoch: 50 [0/100 (0%)]    Loss: 1.241841
Train Epoch: 60 [0/100 (0%)]    Loss: 1.058603
Train Epoch: 70 [0/100 (0%)]    Loss: 1.115418
Train Epoch: 80 [0/100 (0%)]    Loss: 0.944458
Train Epoch: 90 [0/100 (0%)]    Loss: 0.851638
Train Epoch: 100 [0/100 (0%)]   Loss: 0.804358
Train Epoch: 110 [0/100 (0%)]   Loss: 0.541244
Train Epoch: 120 [0/100 (0%)]   Loss: 0.604858
Train Epoch: 130 [0/100 (0%)]   Loss: 0.712817
Train Epoch: 140 [0/100 (0%)]   Loss: 0.614281

Test set: Average loss: 2.4578, Accuracy: 624/2000 (31.20%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.324522
Train Epoch: 10 [0/100 (0%)]    Loss: 2.113816
Train Epoch: 20 [0/100 (0%)]    Loss: 1.846896
Train Epoch: 30 [0/100 (0%)]    Loss: 1.563382
Train Epoch: 40 [0/100 (0%)]    Loss: 1.492098
Train Epoch: 50 [0/100 (0%)]    Loss: 1.342601
Train Epoch: 60 [0/100 (0%)]    Loss: 1.073540
Train Epoch: 70 [0/100 (0%)]    Loss: 1.199202
Train Epoch: 80 [0/100 (0%)]    Loss: 1.117634
Train Epoch: 90 [0/100 (0%)]    Loss: 0.942020
Train Epoch: 100 [0/100 (0%)]   Loss: 0.774025
Train Epoch: 110 [0/100 (0%)]   Loss: 0.880215
Train Epoch: 120 [0/100 (0%)]   Loss: 0.739152
Train Epoch: 130 [0/100 (0%)]   Loss: 0.672294
Train Epoch: 140 [0/100 (0%)]   Loss: 0.730151

Test set: Average loss: 2.2828, Accuracy: 674/2000 (33.70%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.344118
Train Epoch: 10 [0/100 (0%)]    Loss: 1.842051
Train Epoch: 20 [0/100 (0%)]    Loss: 1.629363
Train Epoch: 30 [0/100 (0%)]    Loss: 1.579953
Train Epoch: 40 [0/100 (0%)]    Loss: 1.562980
```

```
Train Epoch: 50 [0/100 (0%)]    Loss: 1.318424
Train Epoch: 60 [0/100 (0%)]    Loss: 1.177732
Train Epoch: 70 [0/100 (0%)]    Loss: 1.060522
Train Epoch: 80 [0/100 (0%)]    Loss: 0.912594
Train Epoch: 90 [0/100 (0%)]    Loss: 0.871638
Train Epoch: 100 [0/100 (0%)]   Loss: 0.773438
Train Epoch: 110 [0/100 (0%)]   Loss: 0.752173
Train Epoch: 120 [0/100 (0%)]   Loss: 0.701181
Train Epoch: 130 [0/100 (0%)]   Loss: 0.584612
Train Epoch: 140 [0/100 (0%)]   Loss: 0.649081

Test set: Average loss: 2.2710, Accuracy: 645/2000 (32.25%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.374857
Train Epoch: 10 [0/100 (0%)]    Loss: 2.006327
Train Epoch: 20 [0/100 (0%)]    Loss: 1.827475
Train Epoch: 30 [0/100 (0%)]    Loss: 1.663929
Train Epoch: 40 [0/100 (0%)]    Loss: 1.402316
Train Epoch: 50 [0/100 (0%)]    Loss: 1.451772
Train Epoch: 60 [0/100 (0%)]    Loss: 1.333224
Train Epoch: 70 [0/100 (0%)]    Loss: 1.143562
Train Epoch: 80 [0/100 (0%)]    Loss: 1.102418
Train Epoch: 90 [0/100 (0%)]    Loss: 1.144685
Train Epoch: 100 [0/100 (0%)]   Loss: 0.937401
Train Epoch: 110 [0/100 (0%)]   Loss: 0.987714
Train Epoch: 120 [0/100 (0%)]   Loss: 0.985414
Train Epoch: 130 [0/100 (0%)]   Loss: 0.875858
Train Epoch: 140 [0/100 (0%)]   Loss: 0.731074

Test set: Average loss: 2.1360, Accuracy: 711/2000 (35.55%)

Acc over 10 instances: 33.93 +- 1.84
Training time over 10 instances: 7.05 +- 0.20
```

Final test on 4 block VGG_Net

[23]:
```
accs = []
times = []
EPOCHS = 150
learning_rate = 0.05
for seed in [1,2,3,4,5,6,7,8,9,10]:
  train_loader, val_loader =␣
 ↪get_data_loaders(cifar_data_train,cifar_data_test,seed,128)

  model = VGG_Net()
  model.to(device)
  optimizer = torch.optim.SGD(model.parameters(),
```

```
                              lr=learning_rate, momentum=0.9,
                              weight_decay=0.0005)
  duration = learning_rate * (0.005 ** 3)
  scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, EPOCHS,␣
↪duration, -1)
  start_time = timeit.default_timer()
  for epoch in range(EPOCHS):
    display_bool  = epoch%10==0
    train(model, device, train_loader, optimizer, epoch, display=display_bool)
    scheduler.step()
  elapsed = timeit.default_timer() - start_time
  times.append(elapsed)
  accs.append(test(model, device, val_loader))

accs = np.array(accs)
times = np.array(times)
print('Acc over 10 instances: %.2f +- %.2f'%(accs.mean(),accs.std()))
print('Training time over 10 instances: %.2f +- %.2f'%(times.mean(),times.
↪std()))
```

```
Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.321645
Train Epoch: 10 [0/100 (0%)]    Loss: 1.977180
Train Epoch: 20 [0/100 (0%)]    Loss: 1.677113
Train Epoch: 30 [0/100 (0%)]    Loss: 1.697210
Train Epoch: 40 [0/100 (0%)]    Loss: 1.402687
Train Epoch: 50 [0/100 (0%)]    Loss: 1.323851
Train Epoch: 60 [0/100 (0%)]    Loss: 1.154911
Train Epoch: 70 [0/100 (0%)]    Loss: 0.956166
Train Epoch: 80 [0/100 (0%)]    Loss: 0.621303
Train Epoch: 90 [0/100 (0%)]    Loss: 0.594957
Train Epoch: 100 [0/100 (0%)]   Loss: 0.726405
Train Epoch: 110 [0/100 (0%)]   Loss: 0.642328
Train Epoch: 120 [0/100 (0%)]   Loss: 0.580420
Train Epoch: 130 [0/100 (0%)]   Loss: 0.662785
Train Epoch: 140 [0/100 (0%)]   Loss: 0.523455
```

```
/usr/local/lib/python3.7/dist-packages/torch/nn/_reduction.py:42: UserWarning:
size_average and reduce args will be deprecated, please use reduction='sum'
instead.
  warnings.warn(warning.format(ret))
```

```

Test set: Average loss: 2.4527, Accuracy: 644/2000 (32.20%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.312232
Train Epoch: 10 [0/100 (0%)]    Loss: 2.001851
```

```
Train Epoch: 20 [0/100 (0%)]    Loss: 1.697603
Train Epoch: 30 [0/100 (0%)]    Loss: 1.463761
Train Epoch: 40 [0/100 (0%)]    Loss: 1.408589
Train Epoch: 50 [0/100 (0%)]    Loss: 1.154686
Train Epoch: 60 [0/100 (0%)]    Loss: 0.978711
Train Epoch: 70 [0/100 (0%)]    Loss: 0.954801
Train Epoch: 80 [0/100 (0%)]    Loss: 0.828743
Train Epoch: 90 [0/100 (0%)]    Loss: 0.782550
Train Epoch: 100 [0/100 (0%)]   Loss: 0.761894
Train Epoch: 110 [0/100 (0%)]   Loss: 0.802367
Train Epoch: 120 [0/100 (0%)]   Loss: 0.484565
Train Epoch: 130 [0/100 (0%)]   Loss: 0.898101
Train Epoch: 140 [0/100 (0%)]   Loss: 0.803677

Test set: Average loss: 2.6416, Accuracy: 659/2000 (32.95%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.322818
Train Epoch: 10 [0/100 (0%)]    Loss: 2.054726
Train Epoch: 20 [0/100 (0%)]    Loss: 1.718863
Train Epoch: 30 [0/100 (0%)]    Loss: 1.679921
Train Epoch: 40 [0/100 (0%)]    Loss: 1.187498
Train Epoch: 50 [0/100 (0%)]    Loss: 1.190835
Train Epoch: 60 [0/100 (0%)]    Loss: 1.268718
Train Epoch: 70 [0/100 (0%)]    Loss: 1.086241
Train Epoch: 80 [0/100 (0%)]    Loss: 0.679833
Train Epoch: 90 [0/100 (0%)]    Loss: 1.091842
Train Epoch: 100 [0/100 (0%)]   Loss: 0.791955
Train Epoch: 110 [0/100 (0%)]   Loss: 0.687368
Train Epoch: 120 [0/100 (0%)]   Loss: 0.555208
Train Epoch: 130 [0/100 (0%)]   Loss: 0.749968
Train Epoch: 140 [0/100 (0%)]   Loss: 0.660940

Test set: Average loss: 2.3304, Accuracy: 752/2000 (37.60%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.327940
Train Epoch: 10 [0/100 (0%)]    Loss: 2.059176
Train Epoch: 20 [0/100 (0%)]    Loss: 1.815549
Train Epoch: 30 [0/100 (0%)]    Loss: 1.348611
Train Epoch: 40 [0/100 (0%)]    Loss: 1.309313
Train Epoch: 50 [0/100 (0%)]    Loss: 1.190574
Train Epoch: 60 [0/100 (0%)]    Loss: 0.871405
Train Epoch: 70 [0/100 (0%)]    Loss: 0.769270
Train Epoch: 80 [0/100 (0%)]    Loss: 0.894676
Train Epoch: 90 [0/100 (0%)]    Loss: 0.750949
Train Epoch: 100 [0/100 (0%)]   Loss: 0.683270
Train Epoch: 110 [0/100 (0%)]   Loss: 0.513948
```

```
Train Epoch: 120 [0/100 (0%)]   Loss: 0.614650
Train Epoch: 130 [0/100 (0%)]   Loss: 0.649457
Train Epoch: 140 [0/100 (0%)]   Loss: 0.513803

Test set: Average loss: 2.4566, Accuracy: 711/2000 (35.55%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.312976
Train Epoch: 10 [0/100 (0%)]    Loss: 1.940813
Train Epoch: 20 [0/100 (0%)]    Loss: 1.709565
Train Epoch: 30 [0/100 (0%)]    Loss: 1.326833
Train Epoch: 40 [0/100 (0%)]    Loss: 1.267784
Train Epoch: 50 [0/100 (0%)]    Loss: 1.157953
Train Epoch: 60 [0/100 (0%)]    Loss: 1.053111
Train Epoch: 70 [0/100 (0%)]    Loss: 0.900621
Train Epoch: 80 [0/100 (0%)]    Loss: 0.766811
Train Epoch: 90 [0/100 (0%)]    Loss: 0.924690
Train Epoch: 100 [0/100 (0%)]   Loss: 0.740567
Train Epoch: 110 [0/100 (0%)]   Loss: 0.824353
Train Epoch: 120 [0/100 (0%)]   Loss: 0.450384
Train Epoch: 130 [0/100 (0%)]   Loss: 0.536934
Train Epoch: 140 [0/100 (0%)]   Loss: 0.591570

Test set: Average loss: 2.5445, Accuracy: 653/2000 (32.65%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.322575
Train Epoch: 10 [0/100 (0%)]    Loss: 2.125710
Train Epoch: 20 [0/100 (0%)]    Loss: 1.750500
Train Epoch: 30 [0/100 (0%)]    Loss: 1.264611
Train Epoch: 40 [0/100 (0%)]    Loss: 1.424606
Train Epoch: 50 [0/100 (0%)]    Loss: 1.250599
Train Epoch: 60 [0/100 (0%)]    Loss: 0.939110
Train Epoch: 70 [0/100 (0%)]    Loss: 0.848734
Train Epoch: 80 [0/100 (0%)]    Loss: 0.734567
Train Epoch: 90 [0/100 (0%)]    Loss: 0.844955
Train Epoch: 100 [0/100 (0%)]   Loss: 0.807697
Train Epoch: 110 [0/100 (0%)]   Loss: 0.619958
Train Epoch: 120 [0/100 (0%)]   Loss: 0.660349
Train Epoch: 130 [0/100 (0%)]   Loss: 0.590745
Train Epoch: 140 [0/100 (0%)]   Loss: 0.688726

Test set: Average loss: 2.3699, Accuracy: 754/2000 (37.70%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.340241
Train Epoch: 10 [0/100 (0%)]    Loss: 2.074024
Train Epoch: 20 [0/100 (0%)]    Loss: 1.748350
```

```
Train Epoch: 30 [0/100 (0%)]    Loss: 1.547470
Train Epoch: 40 [0/100 (0%)]    Loss: 1.206303
Train Epoch: 50 [0/100 (0%)]    Loss: 1.158733
Train Epoch: 60 [0/100 (0%)]    Loss: 0.867207
Train Epoch: 70 [0/100 (0%)]    Loss: 0.914654
Train Epoch: 80 [0/100 (0%)]    Loss: 0.821548
Train Epoch: 90 [0/100 (0%)]    Loss: 0.676306
Train Epoch: 100 [0/100 (0%)]   Loss: 0.563781
Train Epoch: 110 [0/100 (0%)]   Loss: 0.591518
Train Epoch: 120 [0/100 (0%)]   Loss: 0.606856
Train Epoch: 130 [0/100 (0%)]   Loss: 0.635555
Train Epoch: 140 [0/100 (0%)]   Loss: 0.650764

Test set: Average loss: 2.6262, Accuracy: 623/2000 (31.15%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.334088
Train Epoch: 10 [0/100 (0%)]    Loss: 2.050284
Train Epoch: 20 [0/100 (0%)]    Loss: 1.814426
Train Epoch: 30 [0/100 (0%)]    Loss: 1.600226
Train Epoch: 40 [0/100 (0%)]    Loss: 1.482042
Train Epoch: 50 [0/100 (0%)]    Loss: 1.223956
Train Epoch: 60 [0/100 (0%)]    Loss: 1.231164
Train Epoch: 70 [0/100 (0%)]    Loss: 0.999632
Train Epoch: 80 [0/100 (0%)]    Loss: 0.858637
Train Epoch: 90 [0/100 (0%)]    Loss: 0.773526
Train Epoch: 100 [0/100 (0%)]   Loss: 0.797117
Train Epoch: 110 [0/100 (0%)]   Loss: 0.689044
Train Epoch: 120 [0/100 (0%)]   Loss: 0.818884
Train Epoch: 130 [0/100 (0%)]   Loss: 0.486521
Train Epoch: 140 [0/100 (0%)]   Loss: 0.611566

Test set: Average loss: 2.4197, Accuracy: 698/2000 (34.90%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]     Loss: 2.295387
Train Epoch: 10 [0/100 (0%)]    Loss: 2.019333
Train Epoch: 20 [0/100 (0%)]    Loss: 1.872334
Train Epoch: 30 [0/100 (0%)]    Loss: 1.485014
Train Epoch: 40 [0/100 (0%)]    Loss: 1.253632
Train Epoch: 50 [0/100 (0%)]    Loss: 1.111021
Train Epoch: 60 [0/100 (0%)]    Loss: 1.410300
Train Epoch: 70 [0/100 (0%)]    Loss: 0.987970
Train Epoch: 80 [0/100 (0%)]    Loss: 0.917053
Train Epoch: 90 [0/100 (0%)]    Loss: 0.703128
Train Epoch: 100 [0/100 (0%)]   Loss: 0.650320
Train Epoch: 110 [0/100 (0%)]   Loss: 0.592226
Train Epoch: 120 [0/100 (0%)]   Loss: 0.835847
```

```
Train Epoch: 130 [0/100 (0%)]	Loss: 0.644251
Train Epoch: 140 [0/100 (0%)]	Loss: 0.527865

Test set: Average loss: 2.3659, Accuracy: 698/2000 (34.90%)

Num Samples For Training 100 Num Samples For Val 2000
Train Epoch: 0 [0/100 (0%)]	Loss: 2.304602
Train Epoch: 10 [0/100 (0%)]	Loss: 1.981912
Train Epoch: 20 [0/100 (0%)]	Loss: 1.682592
Train Epoch: 30 [0/100 (0%)]	Loss: 1.609216
Train Epoch: 40 [0/100 (0%)]	Loss: 1.353532
Train Epoch: 50 [0/100 (0%)]	Loss: 1.097617
Train Epoch: 60 [0/100 (0%)]	Loss: 1.016961
Train Epoch: 70 [0/100 (0%)]	Loss: 0.893169
Train Epoch: 80 [0/100 (0%)]	Loss: 1.042039
Train Epoch: 90 [0/100 (0%)]	Loss: 0.920236
Train Epoch: 100 [0/100 (0%)]	Loss: 0.678921
Train Epoch: 110 [0/100 (0%)]	Loss: 0.650951
Train Epoch: 120 [0/100 (0%)]	Loss: 0.716926
Train Epoch: 130 [0/100 (0%)]	Loss: 0.724038
Train Epoch: 140 [0/100 (0%)]	Loss: 0.507296

Test set: Average loss: 2.2778, Accuracy: 738/2000 (36.90%)

Acc over 10 instances: 34.65 +- 2.22
Training time over 10 instances: 7.40 +- 0.05
```

[24]:

```python
model = ConvNet()
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print(f'Number of parameters in the 4 block convnet: {params}')
model = VGG_Net()
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print(f'Number of parameters in the 3 block VGG_Net: {params}')
```

```
Number of parameters in the 4 block convnet: 113738
Number of parameters in the 3 block VGG_Net: 4646922
```

[48]:

```
```

Challenge2_resnet_finetune

May 3, 2021

[1]:
```
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Subset
import numpy as np
from numpy.random import RandomState
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
```

[2]:
```
# Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inception]
model_name = "resnet"

num_classes = 10

num_epochs = 20

feature_extract = True
```

[3]:
```
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())
            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
```

[4]:
```
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

[5]:
```
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)
        num_ftrs = model_ft.fc.in_features
```

```
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "vgg":
        """ VGG11_bn
        """
        model_ft = models.vgg11_bn(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    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)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to
/root/.cache/torch/hub/checkpoints/resnet18-5c106cde.pth
```

```
HBox(children=(FloatProgress(value=0.0, max=46827520.0), HTML(value='')))
```

[6]:
```
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])
    ]),
}

##### Cifar Data
cifar_data_train = datasets.CIFAR10(root='.',train=False,␣
 ↪transform=data_transforms["train"], download=True)
```

```
#We need two copies of this due to weird dataset api
cifar_data_test = datasets.CIFAR10(root='.',train=False,␣
↪transform=data_transforms["val"], download=True)

seed=0
prng = RandomState(seed)
random_permute = prng.permutation(np.arange(0, 1000))
indx_train = np.concatenate([np.where(np.array(cifar_data_train.targets) ==␣
↪classe)[0][random_permute[0:10]] for classe in range(0, 10)])
indx_val = np.concatenate([np.where(np.array(cifar_data_test.targets) ==␣
↪classe)[0][random_permute[10:210]] for classe in range(0, 10)])


train_data = Subset(cifar_data_train, indx_train)
val_data = Subset(cifar_data_test, indx_val)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=128,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(val_data,
                                         batch_size=128,
                                         shuffle=False)

dataloaders_dict = {"train":train_loader, "val":val_loader}

# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz

HBox(children=(FloatProgress(value=0.0, max=170498071.0), HTML(value='')))


Extracting ./cifar-10-python.tar.gz to .
Files already downloaded and verified
Device: cuda:0
```

[7]:

```
# 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)

optimizer_ft = optim.SGD(params_to_update, lr=0.05, momentum=0.9)
```

```
Params to learn:
         fc.weight
         fc.bias
```

[8]:
```
criterion = nn.CrossEntropyLoss()

model_ft, hist = train_model(model_ft, dataloaders_dict, criterion,␣
 ↪optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
Epoch 0/19
----------
train Loss: 2.5242 Acc: 0.0700
val Loss: 2.5310 Acc: 0.0850

Epoch 1/19
----------
train Loss: 2.4719 Acc: 0.1500
val Loss: 2.4966 Acc: 0.1535

Epoch 2/19
----------
train Loss: 2.5039 Acc: 0.2400
val Loss: 2.2523 Acc: 0.2355

Epoch 3/19
----------
train Loss: 2.2105 Acc: 0.2600
val Loss: 1.8955 Acc: 0.3410

Epoch 4/19
----------
```

```
train Loss: 1.7238 Acc: 0.5100
val Loss: 2.3710 Acc: 0.1600

Epoch 5/19
----------
train Loss: 2.1212 Acc: 0.1500
val Loss: 1.8747 Acc: 0.4075

Epoch 6/19
----------
train Loss: 1.8863 Acc: 0.4900
val Loss: 2.1997 Acc: 0.4560

Epoch 7/19
----------
train Loss: 1.9853 Acc: 0.6200
val Loss: 2.3796 Acc: 0.4355

Epoch 8/19
----------
train Loss: 2.2908 Acc: 0.5100
val Loss: 2.1852 Acc: 0.4550

Epoch 9/19
----------
train Loss: 1.8797 Acc: 0.6800
val Loss: 1.8977 Acc: 0.4610

Epoch 10/19
----------
train Loss: 1.7121 Acc: 0.6200
val Loss: 1.4710 Acc: 0.4885

Epoch 11/19
----------
train Loss: 1.0902 Acc: 0.7400
val Loss: 1.9581 Acc: 0.3485

Epoch 12/19
----------
train Loss: 1.0881 Acc: 0.6200
val Loss: 2.8084 Acc: 0.2495

Epoch 13/19
----------
train Loss: 1.5558 Acc: 0.4900
val Loss: 1.4513 Acc: 0.4935
```

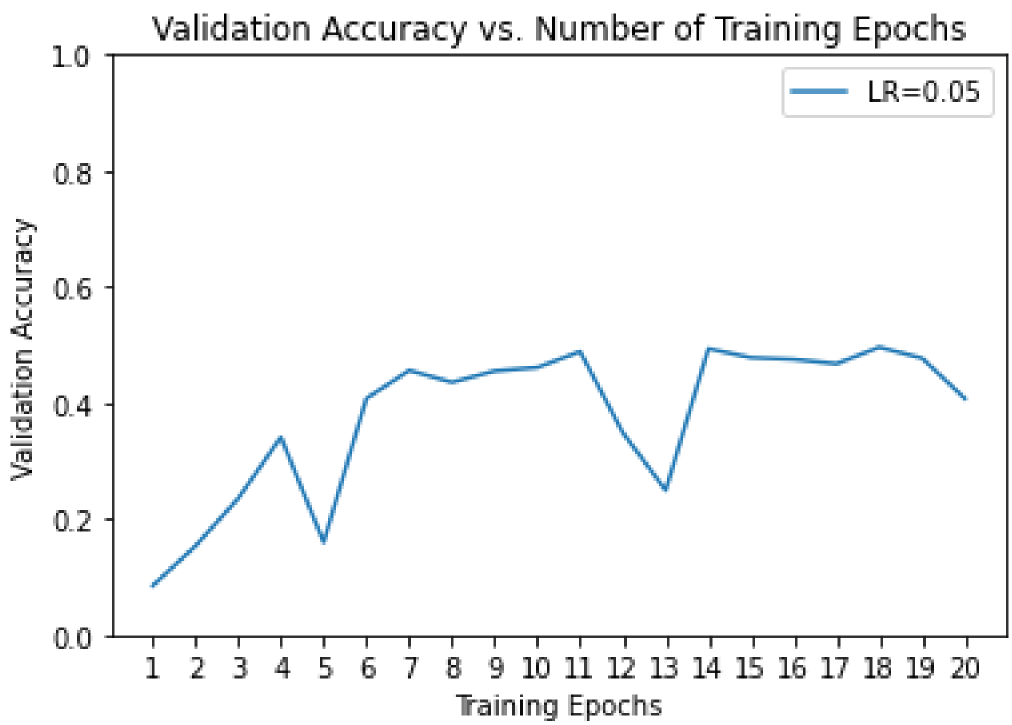
```
Epoch 14/19
----------
train Loss: 0.9629 Acc: 0.6600
val Loss: 1.5718 Acc: 0.4780

Epoch 15/19
----------
train Loss: 1.0089 Acc: 0.7100
val Loss: 1.6915 Acc: 0.4750

Epoch 16/19
----------
train Loss: 1.0584 Acc: 0.6900
val Loss: 1.7664 Acc: 0.4675

Epoch 17/19
----------
train Loss: 1.3072 Acc: 0.6300
val Loss: 1.6319 Acc: 0.4960

Epoch 18/19
----------
train Loss: 0.9496 Acc: 0.7500
val Loss: 1.6153 Acc: 0.4770

Epoch 19/19
----------
train Loss: 0.8080 Acc: 0.7600
val Loss: 1.9857 Acc: 0.4075

Training complete in 2m 31s
Best val Acc: 0.496000
```

[9]:

```
torch.save(model_ft.state_dict(), './resnet_ft_05.pth')

ohist = []

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

print(ohist)

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1), ohist, label="LR=0.05")
```

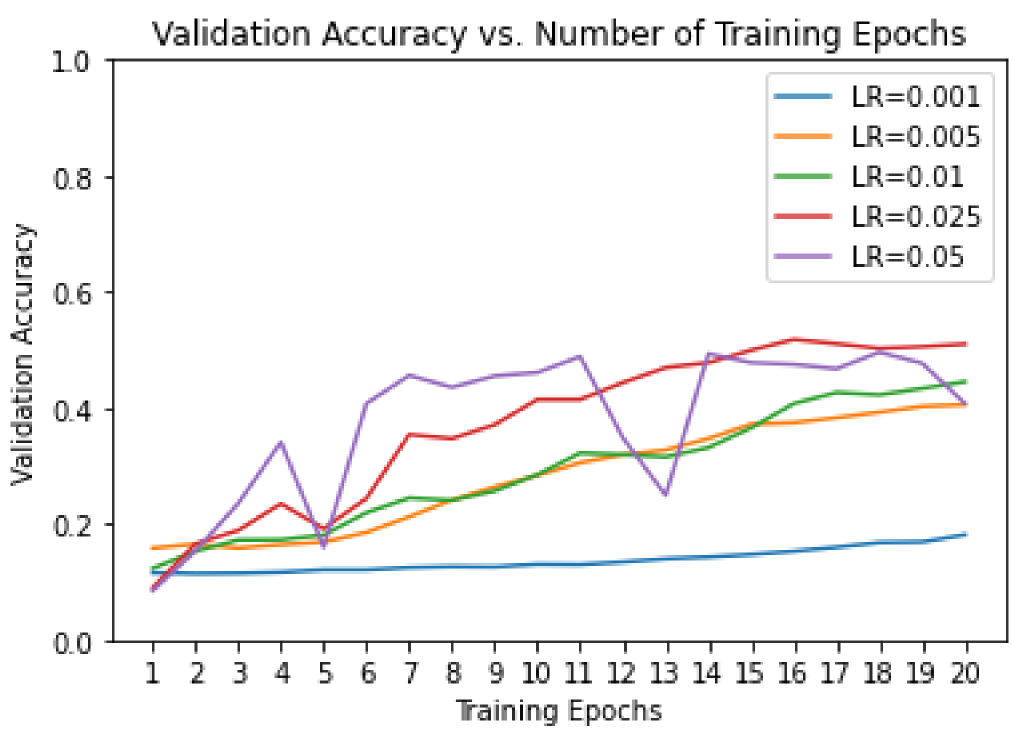
```
plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.savefig('resnet_finetuned_05.png')
plt.show()
```

```
[array(0.085), array(0.1535), array(0.2355), array(0.341), array(0.16),
array(0.4075), array(0.456), array(0.4355), array(0.455), array(0.461),
array(0.4885), array(0.3485), array(0.2495), array(0.4935), array(0.478),
array(0.475), array(0.4675), array(0.496), array(0.477), array(0.4075)]
```



[10]:
```
accs_001 = [0.116, 0.1145, 0.115, 0.117, 0.121, 0.121, 0.125, 0.1265, 0.126, 0.
↪1305, 0.13, 0.1345, 0.14, 0.143, 0.1475, 0.153, 0.16, 0.168, 0.169, 0.1815]
accs_005 = [0.1585, 0.1655, 0.1585, 0.1645, 0.169, 0.1855, 0.2125, 0.2415, 0.
↪264, 0.284, 0.3055, 0.319, 0.3275, 0.347, 0.372, 0.3745, 0.383, 0.3925, 0.
↪4025, 0.4055]
accs_01 = [0.1235, 0.153, 0.1725, 0.173, 0.181, 0.2195, 0.245, 0.241, 0.257, 0.
↪285, 0.322, 0.3195, 0.3155, 0.3315, 0.3655, 0.4075, 0.4265, 0.4225, 0.4335,␣
↪0.445]
accs_025 = [0.09, 0.166, 0.189, 0.235, 0.192, 0.244, 0.354, 0.347, 0.371, 0.
↪4145, 0.4145, 0.4435, 0.4695, 0.478, 0.4995, 0.518, 0.5105, 0.5025, 0.5055,␣
↪0.51]
```

```
accs_05 = [0.085, 0.1535, 0.2355, 0.341, 0.16, 0.4075, 0.456, 0.4355, 0.455, 0.
↪461, 0.4885, 0.3485, 0.2495, 0.4935, 0.478, 0.475, 0.4675, 0.496, 0.477, 0.
↪4075]

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1), accs_001, label="LR=0.001")
plt.plot(range(1,num_epochs+1), accs_005, label="LR=0.005")
plt.plot(range(1,num_epochs+1), accs_01, label="LR=0.01")
plt.plot(range(1,num_epochs+1), accs_025, label="LR=0.025")
plt.plot(range(1,num_epochs+1), accs_05, label="LR=0.05")


plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.savefig('resnet_ft_all.png')
plt.show()
```



[ ]:

Challenge2_vgg_finetune

May 3, 2021

[1]:
```python
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Subset
import numpy as np
from numpy.random import RandomState
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
```

[2]:
```python
# Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inception]
model_name = "vgg"

num_classes = 10

num_epochs = 20

feature_extract = True
```

[3]:
```python
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())
            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
```

[4]:
```
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

[5]:
```
def initialize_model(model_name, num_classes, feature_extract,␣
 ↪use_pretrained=True):
    model_ft = None
    input_size = 0
    model_ft = models.vgg11_bn(pretrained=use_pretrained)
    set_parameter_requires_grad(model_ft, feature_extract)
    num_ftrs = model_ft.classifier[6].in_features
    model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
    input_size = 224

    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)
```

```
Downloading: "https://download.pytorch.org/models/vgg11_bn-6002323d.pth" to
/root/.cache/torch/hub/checkpoints/vgg11_bn-6002323d.pth
```

```
HBox(children=(FloatProgress(value=0.0, max=531503671.0), HTML(value='')))
```

[6]:
```
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])
    ]),
}

##### Cifar Data
cifar_data_train = datasets.CIFAR10(root='.',train=False,␣
 ↪transform=data_transforms["train"], download=True)

#We need two copies of this due to weird dataset api
cifar_data_test = datasets.CIFAR10(root='.',train=False,␣
 ↪transform=data_transforms["val"], download=True)

seed=0
prng = RandomState(seed)
random_permute = prng.permutation(np.arange(0, 1000))
indx_train = np.concatenate([np.where(np.array(cifar_data_train.targets) ==␣
 ↪classe)[0][random_permute[0:10]] for classe in range(0, 10)])
indx_val = np.concatenate([np.where(np.array(cifar_data_test.targets) ==␣
 ↪classe)[0][random_permute[10:210]] for classe in range(0, 10)])


train_data = Subset(cifar_data_train, indx_train)
val_data = Subset(cifar_data_test, indx_val)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=128,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(val_data,
                                         batch_size=128,
```

```
                                                   shuffle=False)

dataloaders_dict = {"train":train_loader, "val":val_loader}

# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz

HBox(children=(FloatProgress(value=0.0, max=170498071.0), HTML(value='')))

Extracting ./cifar-10-python.tar.gz to .
Files already downloaded and verified
Device: cpu
```

[7]:

```
# 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)

optimizer_ft = optim.SGD(params_to_update, lr=0.005, momentum=0.9)
```

```
Params to learn:
	 classifier.6.weight
	 classifier.6.bias
```

[ ]:
```
criterion = nn.CrossEntropyLoss()

model_ft, hist = train_model(model_ft, dataloaders_dict, criterion,␣
 ↪optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))
```

```
Epoch 0/19
----------
train Loss: 2.3693 Acc: 0.0800
val Loss: 2.2863 Acc: 0.1300

Epoch 1/19
----------
train Loss: 2.3396 Acc: 0.0900
val Loss: 2.2614 Acc: 0.1710

Epoch 2/19
----------
train Loss: 2.2991 Acc: 0.1000
val Loss: 2.2283 Acc: 0.2130

Epoch 3/19
----------
train Loss: 2.2389 Acc: 0.1600
val Loss: 2.1877 Acc: 0.2700

Epoch 4/19
----------
train Loss: 2.1857 Acc: 0.2900
val Loss: 2.1415 Acc: 0.3310

Epoch 5/19
----------
train Loss: 2.1548 Acc: 0.3000
val Loss: 2.0901 Acc: 0.3825

Epoch 6/19
----------
train Loss: 2.1001 Acc: 0.3500
val Loss: 2.0355 Acc: 0.4320

Epoch 7/19
----------
train Loss: 2.0421 Acc: 0.4500
val Loss: 1.9809 Acc: 0.4690

Epoch 8/19
----------
```

```
train Loss: 1.9902 Acc: 0.3800
val Loss: 1.9267 Acc: 0.4890

Epoch 9/19
----------
train Loss: 1.9179 Acc: 0.4200
val Loss: 1.8743 Acc: 0.4935

Epoch 10/19
----------
train Loss: 1.8494 Acc: 0.5000
```

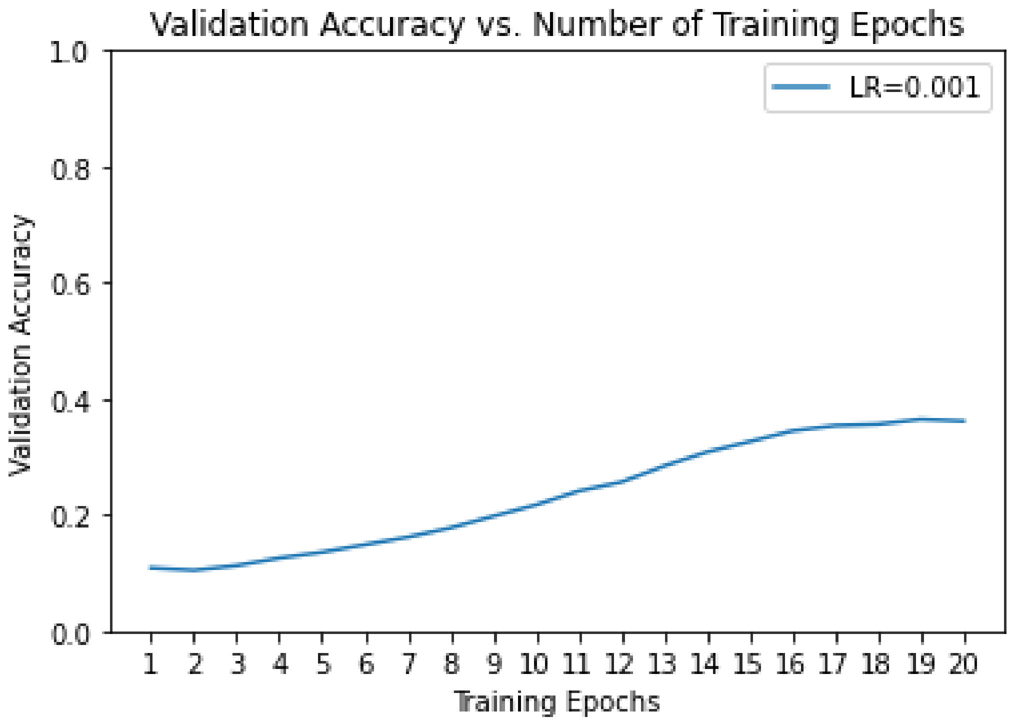
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-8-4b3739047485> in <module>
      1 criterion = nn.CrossEntropyLoss()
      2
----> 3 model_ft, hist = train_model(model_ft, dataloaders_dict, criterion,␣
 ↪optimizer_ft, num_epochs=num_epochs, is_inception=(model_name=="inception"))

<ipython-input-3-391d22e8ca1a> in train_model(model, dataloaders, criterion,␣
 ↪optimizer, num_epochs, is_inception)
     43                         loss = loss1 + 0.4*loss2
     44                     else:
---> 45                         outputs = model(inputs)
     46                         loss = criterion(outputs, labels)
     47

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\modules\mo
 ↪py in _call_impl(self, *input, **kwargs)
    720             result = self._slow_forward(*input, **kwargs)
    721         else:
--> 722             result = self.forward(*input, **kwargs)
    723         for hook in itertools.chain(
    724                 _global_forward_hooks.values(),

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\ision\models\
 ↪py in forward(self, x)
     40
     41     def forward(self, x):
---> 42         x = self.features(x)
     43         x = self.avgpool(x)
     44         x = x.view(x.size(0), -1)

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\modules\mo
 ↪py in _call_impl(self, *input, **kwargs)
    720             result = self._slow_forward(*input, **kwargs)
```

```
    721         else:
--> 722             result = self.forward(*input, **kwargs)
    723         for hook in itertools.chain(
    724                 _global_forward_hooks.values(),

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\modules\co
 ↪py in forward(self, input)
    115     def forward(self, input):
    116         for module in self:
--> 117             input = module(input)
    118         return input
    119 

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\modules\mo
 ↪py in _call_impl(self, *input, **kwargs)
    720             result = self._slow_forward(*input, **kwargs)
    721         else:
--> 722             result = self.forward(*input, **kwargs)
    723         for hook in itertools.chain(
    724                 _global_forward_hooks.values(),

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\modules\ba
 ↪py in forward(self, input)
    134             self.running_mean if not self.training or self.
 ↪track_running_stats else None,
    135             self.running_var if not self.training or self.
 ↪track_running_stats else None,
--> 136             self.weight, self.bias, bn_training,␣
 ↪exponential_average_factor, self.eps)

    137 
    138 

~\AppData\Local\Continuum\anaconda3\envs\PytorchPruning\lib\site-packages\torch\nn\functional
 ↪py in batch_norm(input, running_mean, running_var, weight, bias, training,␣
 ↪momentum, eps)
   2014     return torch.batch_norm(
   2015         input, weight, bias, running_mean, running_var,
-> 2016         training, momentum, eps, torch.backends.cudnn.enabled
   2017     )
   2018 

KeyboardInterrupt:
```

0.1 See the report for the accurate final picture. A run was started by mistake and I unfortunately do not have time to re-train

[ ]:
```
torch.save(model_ft.state_dict(), './vgg_ft_005.pth')

ohist = []

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

print(ohist)

plt.title("Validation Accuracy vs. Number of Training Epochs")
plt.xlabel("Training Epochs")
plt.ylabel("Validation Accuracy")
plt.plot(range(1,num_epochs+1), ohist, label="LR=0.005")


plt.ylim((0,1.))
plt.xticks(np.arange(1, num_epochs+1, 1.0))
plt.legend()
plt.savefig('vgg_finetuned_005.png')
plt.show()
```

```
[array(0.1085), array(0.1045), array(0.1125), array(0.1255), array(0.1355),
array(0.1485), array(0.1615), array(0.1775), array(0.1975), array(0.217),
array(0.241), array(0.2565), array(0.284), array(0.308), array(0.326),
array(0.3445), array(0.3535), array(0.356), array(0.3645), array(0.3615)]
```


Validation Accuracy vs. Number of Training Epochs
LR=0.001
Validation Accuracy
Training Epochs
1.0
0.8
0.6
0.4
0.2
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

[ ]: