Pneumonia Diagnosis with Deep Learning

Ashish Kumar[1], Ruhin Moghal[2]
CSYE7374 38227: Parallel Machine Learning and Artificial Intelligence- Dr. Handan Liu

Abstract—**Pneumonia has been one of the fatal diseases and has the potential to result in severe consequences within a short period of time, due to the flow of fluid in lungs, which leads to drowning. If not acted upon by drugs at the right time, pneumonia may result in death of individuals. Therefore, the early diagnosis is a key factor along the progress of the disease. This paper focuses on the biological progress of pneumonia and its detection by x-ray imaging, overviews the studies conducted on enhancing the level of diagnosis, and presents the methodology and results of parallel computing with deep learning based on various parameters in order to detect the disease. In this study we propose our deep learning architecture for the classification task, which is trained with x-ray images with and without pneumonia.We have implemented two models, VGG16 and DenseNet121. We achieved better accuracy with DenseNet121. Our findings yield an accuracy of 87.10 % using a single GPU without data parallelism and an accuracy of 89.40 % using 4 GPUs with data parallelism. Also, the time required to train the model using 4 GPUs is a lot faster than training on a single GPU.**

Keywords—**Pneumonia, X-ray, Deep Learning, Transfer Learning, Data Parallelism, Pytorch, Cuda, VGG16, DenseNet121, Multi GPU**

I. INTRODUCTION

Globally, 450 million get infected by pneumonia in a year and 4 million people die from the disease. 1 million people each year have to seek care from hospitals and 50 thousand people die from the disease in the United States of America. The numerical difference between the infection rates and death rates show how crucial the early diagnosis of the disease is. Pneumonia is an inflammatory response in the lung sacs called alveoli. It's often caused by bacteria, viruses, fungi and other microbes. As the germs reach the lung, white blood cells act against the germ and inflammation occurs in the sacs. Thus, alveoli get filled with pneumonia fluid and this fluid causes symptoms like coughing, trouble in breathing and fever. If the infection is not acted upon during the early periods of the disease, pneumonia infection can spread throughout the body and result in the death of the individual, as a result of the inability to exchange gas in the lungs.

Today, one of the most conventional medical techniques used to diagnose the disease is chest x-ray. As the concentrated beam of electrons, called x-ray photons, go through the body tissues, an image is produced on the metal surface (photographic film). During diagnosis, expert radiologists correspond white spots on the image to infiltrates identifying an infection, and white areas to the pneumonia fluid in the lungs. However, the limited color scheme of x-ray images consisting of shades of black and white, cause drawbacks when it comes to determining whether there's an infected area in the lungs or not. This is due to the fact that the high intensity of white wavelength occurs on the photographic film when the fluid in the lungs is high enough to be considered as a dense and solid tissue. In other words, the transition from an air filled tissue (normal state of lungs), which is seen in darker shades, to a dense tissue, requires the sufficient amount of fluid to shift the color scheme to lighter colors. This means that for an x-ray film to be considered as pneumonia, the disease must be in its later stages. Thus, the early detection of pneumonia is restricted due to the limited color scheme of x-ray imaging.

Another drawback for the early diagnosis of pneumonia is the human-dependent detection. Expert radiologists need to have sufficiently trained eyes in order to be able to differentiate between the heterogeneous color distribution of air while flowing in the lungs. This may be seen in different colors on the x-ray image taken, yet not be the dense pneumonia fluid. Thus, it's highly significant for a radiologist to be able to tell whether the white spots on the x-ray film actually correspond to the fluid itself. As a result of the error margin of the human eye, there are many cases where the radiologists fail to make the correct diagnosis. In both cases, whether it's a false positive or false negative diagnosis, it has substantial impacts on the human body. Therefore, computational methods in the diagnosis step of the disease are reliable in terms of consistency. In fig 1 and fig 2, different images with and without pneumonia can be seen.

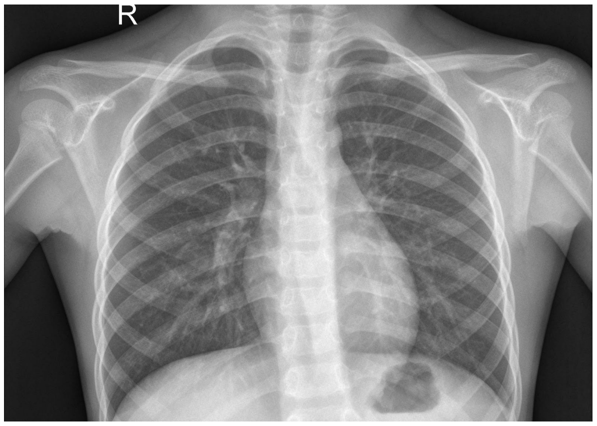


Fig 1: X-ray Image with Pneumonia

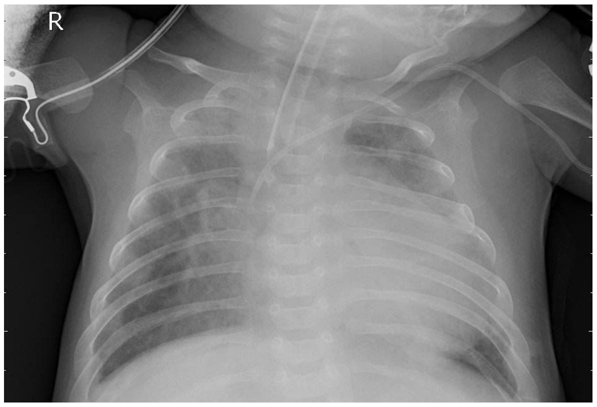


Fig 2: X-ray Image without Pneumonia

The imperceptibility of the healthy versus the pneumonia images can also be witnessed, which portrays the need of well-trained eyes in order to be able to differentiate.

There has been previous studies done regarding pneumonia detection with chest x-rays via machine learning with the use of heat maps, which are images or maps representing the varying temperature or infrared radiation recorded over an area or during a period of time, and differentiation of pulmonary pathology, which is the subspecialty of surgical pathology which deals with the diagnosis and characterization of neoplastic and non-neoplastic diseases of the lungs from normal by using computerized lung sound analysis. Moreover, diagnosing p.carinii pneumonia, which is caused by fungi, with the examination of induced sputum and with indirect immunofluorescence has been used as a method for its specific detection. Aside from using the conventional x-ray imaging, diagnosing lower respiratory tract infection with techniques such as bronchoalveolar lavage, a medical procedure in which a bronchoscope is passed through the mouth or nose into the lungs and fluid is squirted into a small tube, lung biopsy, which is a procedure performed to remove tissue or cells from the body for examination under a microscope, and using lung ultrasonography, which is a technique using echoes of ultrasound pulses to delineate objects or areas of different density in the body to detect neonatal pneumonia, has been done. Neonatal pneumonia is the lung infection in a newborn, which includes lung consolidation with irregular margins and air bronchograms, pleural line abnormalities, and interstitial syndrome [7].

We present a method for classifying pneumonia existence in an x-ray image using deep learning with convolutional neural networks. Deep learning (DL) models continue to grow and the datasets used to train them are increasing in size, leading to longer training times. Therefore, training is being accelerated by deploying DL models across multiple devices (e.g., GPUs/TPUs) in parallel. Data parallelism (DP) is the simplest parallelization strategy (Krizhevsky et al., 2017; Dean et al., 2012; Simonyan & Zisserman, 2014), where replicas of a model are trained on independent devices using independent subsets of data, referred to as mini-batches. Major frameworks (e.g. TensorFlow, PyTorch) support DP using easy-to-use and intuitive APIs. PyTorch has a very useful feature known as data parallelism. Using this feature, PyTorch can distribute computational work among multiple CPU or GPU cores. This feature of PyTorch allows us to use torch.nn.DataParallel to wrap any module and helps us do parallel processing over batch dimension.

In this project, we use parallel computing in the neural network area to reduce computation time. Sequential computing is often not sufficient enough for large scale problems like processing large image datasets and training models in the complex neural networks. Using all cores of the processor can significantly reduce the processing time. We plan to process the images from a pneumonia data set using a CPU, a single GPU and multiple GPUs and then compare the computation time.

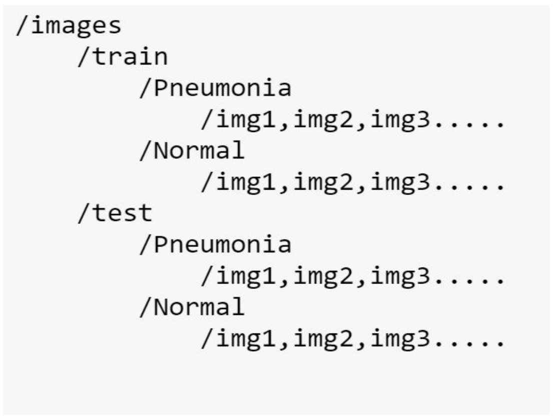
We'll see how to use PyTorch with Data Parallelism to accomplish image classification problems, along the way, learning a little about the library and about the important concept of transfer learning. Therefore, instead of building and training a CNN from scratch, we'll use a pre-built and pre-trained model applying transfer learning. This paper will first explain the methodology in our experimentation, followed by the discussion of the results at hand.

II. SPECIFICATIONS OF DATASET

The dataset was released on a public website, kaggle.com which contained 5840 chest x-ray images of normal and pneumonia patients. Size of the dataset was 1 GB. We have used 4904 images for image training and 936 images for image testing. Link to the dataset is https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia.

A. Data Set Up

With all data science problems, formatting the data correctly will determine the success or failure of the project. Fortunately, the Pneumonia X-ray dataset images were clean and stored in the correct format. If we correctly set up the data directories, PyTorch makes it simple to associate the correct labels with each class. We separated the data into training and testing sets with a 80%, 20% split and then structured the directories as shown in fig 2 below.



```
/images
    /train
        /Pneumonia
            /img1,img2,img3.....
        /Normal
            /img1,img2,img3.....
    /test
        /Pneumonia
            /img1,img2,img3.....
        /Normal
            /img1,img2,img3.....
```

Fig 3: Dataset directory structure

We expect the model to do better on classes with more examples because it can better learn to map features to labels. To deal with the limited number of training examples we used data augmentation during training. Imagenet models need an input size of 224 x 224 so one of the preprocessing steps was to resize the images. Preprocessing was also where we implemented data augmentation for our training data.

B.Data Augmentation

The idea of data augmentation is to artificially increase the number of training images our model sees by applying random transformations to the images. For example, we can randomly rotate or crop the images or flip them horizontally. We wanted our model to distinguish the objects regardless of orientation and data augmentation can also make a model invariant to transformations of the input data.

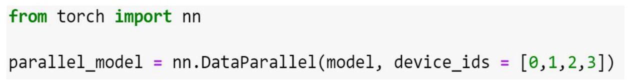
Augmentation is generally only done during training. Each epoch, one iteration through all the training images, a different random transformation was applied to each training image. This means that if we iterate through the data 20 times, our model would see 20 slightly different versions of each image. The overall result was a model that learnt the objects themselves and not how they were presented or artefacts in the image.

III. METHODOLOGY

We classified X-ray images of Pneumonia and normal patients using the Data Parallel method of Pytorch and transfer learning. We implemented transfer learning using pretrained models like vgg16, Resnet50 and densenet121. Transfer Learning is a very useful method as we do not have to build and train a CNN from scratch, we use a pre-built and pre-trained model.

A. Approach to Parallel Machine Learning

There are two ways how we could make use of multiple GPUs.
1. Data Parallelism
Here, we divide batches into smaller batches, and process these smaller batches in parallel on multiple GPU.Data Parallelism in PyTorch is achieved through the nn.DataParallel class. We initialize a nn.DataParallel object with a nn.Module object representing our network, and a list of GPU IDs, across which the batches have to be parallelised. DataParallel is very easy to use, we just add one line to encapsulate the model: device_ids refer to the GPU device number.



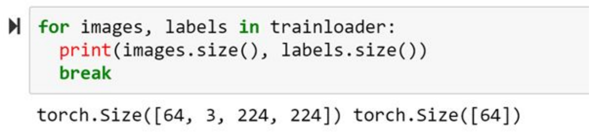
```
from torch import nn

parallel_model = nn.DataParallel(model, device_ids = [0,1,2,3])
```

Fig 4: Code for DataParallel

2. Model Parallelism
Here, we break the neural network into smaller sub networks and then execute these sub networks on different GPUs.

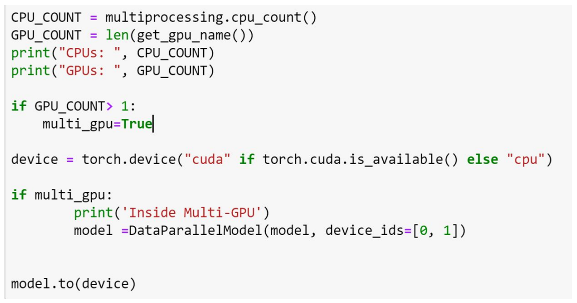
B. Moving tensors around CPU / GPUs

Every Tensor in PyTorch has a to() member function. It's job is to put the tensor on which it's called to a certain device whether it be the CPU or a certain GPU. Input to the to function is a torch.device object which can be initialised with either of the following inputs.
1. cpu for initializing the device for CPU
2. cuda:0 for putting it on GPU number 0.
Generally, whenever we initialise a Tensor, it's put on the CPU. We can move it to the GPU then. We can check whether a GPU is available or not by invoking the torch.cuda.is_available function. We can also move a tensor to a certain GPU by giving it's index as the argument to to function. Importantly, the below piece of code is device agnostic, that is, we don't have to separately change it for it to work on both GPU and the CPU.



```
for images, labels in trainloader:
  print(images.size(), labels.size())
  break
```

```
torch.Size([64, 3, 224, 224]) torch.Size([64])
```



```
CPU_COUNT = multiprocessing.cpu_count()
GPU_COUNT = len(get_gpu_name())
print("CPUs: ", CPU_COUNT)
print("GPUs: ", GPU_COUNT)

if GPU_COUNT> 1:
    multi_gpu=True

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

if multi_gpu:
        print('Inside Multi-GPU')
        model =DataParallelModel(model, device_ids=[0, 1])


model.to(device)
```

Fig 5: torch.cuda.is_available function

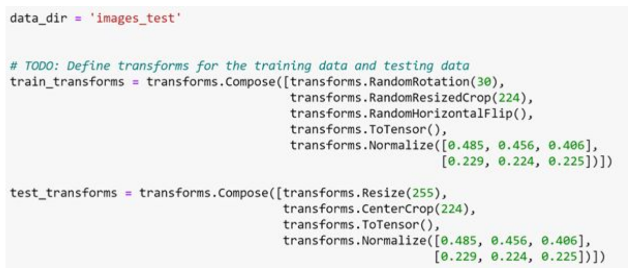
Another way to put tensors on GPUs is to call cuda(n) function on them where n is the index of the GPU. If we just call cuda, then the tensor is placed on GPU 0. The torch.nn.Module class also has to add cuda functions which puts the entire network on a particular device. Unlike, Tensors calling to on the nn.Module object is enough, and there's no need to assign the returned value from the to function.

C. Image Preprocessing

This is the most important step of working with image data. During image preprocessing, we simultaneously prepare the images for our network and apply data augmentation to the training set. Each model will have different input requirements, but if we read through what Imagenet requires, we figure out that our images need to be 224x224 and normalized to a range. To process an image in PyTorch, we use transforms, simple operations applied to arrays. The validation (and testing) transforms are as follows:

1. Resize
2. Center crop to 224 x 224
3. Convert to a tensor
4. Normalize with mean and standard deviation

The end result of passing through these transforms are tensors that can go into our network. The training transformations are similar but with the addition of random augmentations. First up, we define the training and validation transformations. The idea of data augmentation is to artificially increase the number of training images our model sees by applying random transformations to the images. For example, we can randomly rotate or crop the images or flip them horizontally. We want our model to distinguish the objects regardless of orientation and data augmentation can also make a model invariant to transformations of the input data.

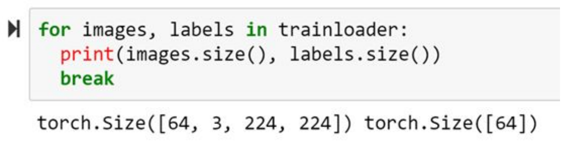


```
data_dir = 'images_test'


# TODO: Define transforms for the training data and testing data
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])
```

Fig 7: Image Preprocessing steps

D. Data Loader

Then, we create datasets and DataLoaders . By using datasets.ImageFolder to make a dataset, PyTorch will automatically associate images with the correct labels provided our directory is set up as above. The datasets are then passed to a DataLoader , an iterator that yields batches of images and labels.



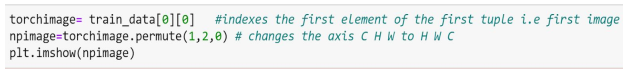
```
for images, labels in trainloader:
  print(images.size(), labels.size())
  break

torch.Size([64, 3, 224, 224]) torch.Size([64])
```

Fig 8: torch.Size

The shape of a batch is (batch_size, color_channels, height, width). During training, validation, and eventually testing, we'll iterate through the DataLoaders, with one pass through the complete dataset comprising one epoch. Every epoch, the training DataLoader will apply a slightly different random transformation to the images for training data augmentation.

E. Displaying an image

The CIFAR10 dataset object returns a tuple containing an image object and a number representing the label of the image. We see from the size of the image data, that each sample is a 3 x 32 x 32 tensor, representing three color values for each of the 322 pixels in the image. It is important to know that this is not quite the same format used for matplotlib. A tensor treats an image in the format of [color, height, width], whereas a numpy image is in the format [height, width, color]. To plot an image, we need to swap axes using the permute() function, or alternatively convert it to a NumPy array and using the transpose function. Note that we do not need to convert the image to a NumPy array, as matplotlib will display the correctly permuted tensor. The following code should make this clear:



```
torchimage= train_data[0][0]   #indexes the first element of the first tuple i.e first image
npimage=torchimage.permute(1,2,0) # changes the axis C H W to H W C
plt.imshow(npimage)
```

Fig 9: Code for Transforming Image

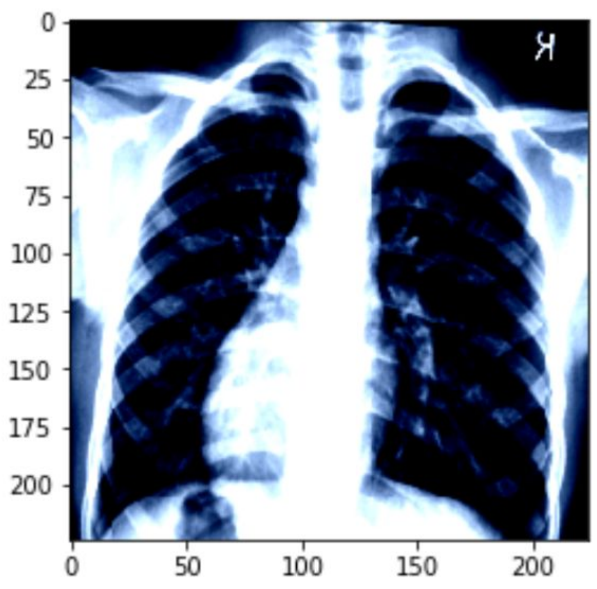
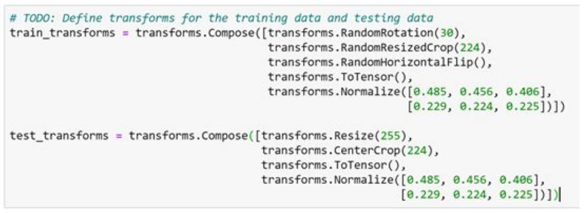


Fig 10: Transformed Image

F. Transforms

The torchvision package includes a number of transforms specifically for Python imaging library images. We can apply multiple transforms to a dataset object using the compose function as follows:



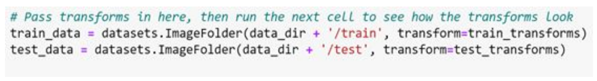
```
# TODO: Define transforms for the training data and testing data
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])
```

Fig 11: Compose Function for multiple transforms

Compose objects are essentially a list of transforms that can then be passed to the dataset as a single variable. It is important to note that the image transforms can only be applied to PIL image data, not tensors. Since transforms in a compose are applied in the order that they are listed, it is important that the ToTensor transform occurs last. If it is placed before the PIL transforms in the Compose list, an error will be generated.

G. ImageFolder

We can see that the main function of the dataset object is to take a sample from a dataset, and the function of DataLoader is to deliver a sample, or a batch of samples, to a deep learning model for evaluation. One of the main things to consider when writing our own dataset object is how do we build a data structure in accessible memory from data that is organized in files on a disk. A common way we might want to organize data is in folders named by class. Let's say that, for this example, we have three folders named Pneumonia and Normal, contained in a parent folder, images. Each of these folders represent the label of the files contained within them. We need to be able to load them while retaining them as separate labels. Happily, there is a class for this, and like most things in PyTorch, it is very easy to use. The class is torchvision.datasets.ImageFolder and it is used as follows:



```
# Pass transforms in here, then run the next cell to see how the transforms look
train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
```

Fig 12 : torchvision.datasets.ImageFolder class

Within the /images/train or /images/test folder, there are two folders, Pneumonia, and Normal, containing images with their folder names indicating labels. Notice that the retrieved labels using DataLoader are represented by integers. Since, in this example, we have two folders, representing two labels, DataLoader returns integers 0 to 1, representing the image labels.

H. Approach to Transfer Learning

The basic premise of transfer learning is simple: take a model trained on a large dataset and transfer its knowledge to a smaller dataset. For image classification with a CNN, we freeze the early convolutional layers of the network and only train the last few layers which make a prediction. The idea is that the convolutional layers extract general, low-level features that are applicable across images such as edges, patterns, gradients and the later layers identify specific features within an image such as eyes or wheels. Thus, we can use a network trained on unrelated categories in a massive dataset (usually Imagenet) and apply it to our own problem because they are universal, low-level features shared between images. The images in the Pneumonia detection dataset are very similar to those in the Imagenet dataset and the knowledge a model learns on Imagenet should easily transfer to this task.

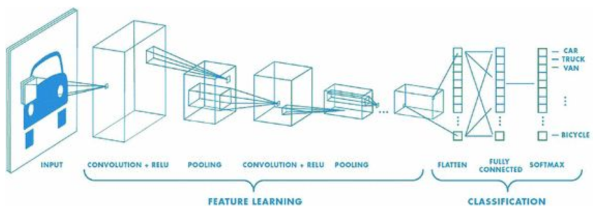
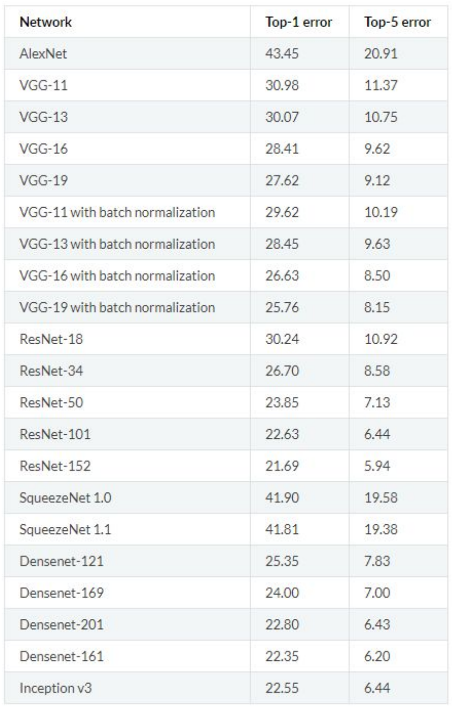


Fig 6: Transfer Learning

Pre-Trained Models for Image Recognition:

With our data in shape, we next turn our attention to the model. For this, we'll use a pre-trained convolutional neural network. PyTorch has a number of models that have already been trained on millions of images from 1000 classes in Imagenet. The complete list of models can be seen here. The performance of these models on Imagenet is shown below:



Network	Top-1 error	Top-5 error
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
VGG-11 with batch normalization	29.62	10.19
VGG-13 with batch normalization	28.45	9.63
VGG-16 with batch normalization	26.63	8.50
VGG-19 with batch normalization	25.76	8.15
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20
Inception v3	22.55	6.44

Fig 15: Performance of pre trained models on Imagenet

For this implementation, we used the VGG-16, densenet121. The process to use a pre-trained model is well-established:

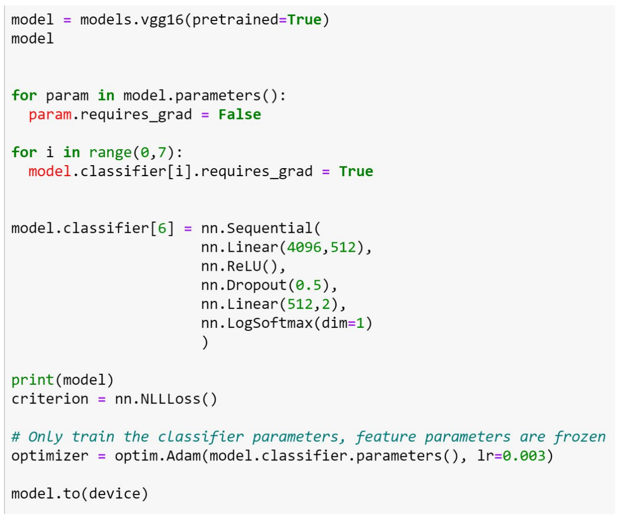
1. Load in pre-trained weights from a network trained on a large dataset
2. Freeze all the weights in the lower (convolutional) layers: the layers to freeze are adjusted depending on similarity of new task to original dataset
3. Replace the upper layers of the network with a custom classifier: the number of outputs must be set equal to the number of classes
4. Train only the custom classifier layers for the task thereby optimizing the model for smaller dataset

Loading in a pre-trained model in PyTorch is simple. Then, we added on our own custom classifier with the following layers:

1. Fully connected with ReLU activation
2. Dropout with 50% chance of dropping
3. Fully connected with log softmax output

When the extra layers are added to the model, they are set to trainable by default (require_grad=True). For the VGG-16, we only changed the last original fully-connected layer. All of the weights in the convolutional layers and the first fully-connected layers are not trainable. Below are the parameters for vgg16:

136,359,234 total parameters.
2,098,690 training parameters.



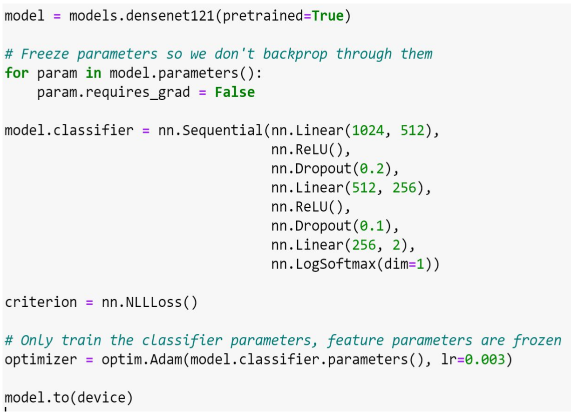
```
model = models.vgg16(pretrained=True)
model


for param in model.parameters():
  param.requires_grad = False

for i in range(0,7):
  model.classifier[i].requires_grad = True


model.classifier[6] = nn.Sequential(
                      nn.Linear(4096,512),
                      nn.ReLU(),
                      nn.Dropout(0.5),
                      nn.Linear(512,2),
                      nn.LogSoftmax(dim=1)
                      )

print(model)
criterion = nn.NLLLoss()

# Only train the classifier parameters, feature parameters are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.003)

model.to(device)
```

Fig 16: VGG-16 Implementation

Densenet :

Densely convolutional networks—in contrast to standard CNNs, where weights propagate through each layer from input to output— each layer, the feature maps for all preceding layers are used as inputs. This results in shorter connections between layers and a network that encourages the reuse of parameters. This results in fewer parameters and strengthens the propagation of features. DenseNet is available in the Densenet121, Densenet169, and Densenet201 variants. We have used densenet121 for our predictions. Below are the parameters for Densenet121:

7,610,498 total parameters.
656,642 training parameters.



```
model = models.densenet121(pretrained=True)

# Freeze parameters so we don't backprop through them
for param in model.parameters():
    param.requires_grad = False

model.classifier = nn.Sequential(nn.Linear(1024, 512),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(512, 256),
                                 nn.ReLU(),
                                 nn.Dropout(0.1),
                                 nn.Linear(256, 2),
                                 nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()

# Only train the classifier parameters, feature parameters are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.003)

model.to(device)
```

Fig 17: Densenet121 Implementation

I. Training the model

Model training in PyTorch is a little more hands-on than in Keras because we have to do the backpropagation and parameter update step ourselves. The main loop iterates over a number of epochs and on each epoch we iterate through the train DataLoader . The DataLoader yields one batch of data and targets which we pass through the model. After each training batch, we calculate the loss, backpropagate the gradients of the loss with respect to the model parameters, and then update the parameters with the optimizer.

When we train a deep learning model, two main operations are performed, Forward Pass and Backward Pass. In forward pass, input is passed through the neural network and after processing the input, an output is generated. Whereas in backward pass, we update the weights of the neural network on the basis of error we get in forward pass.

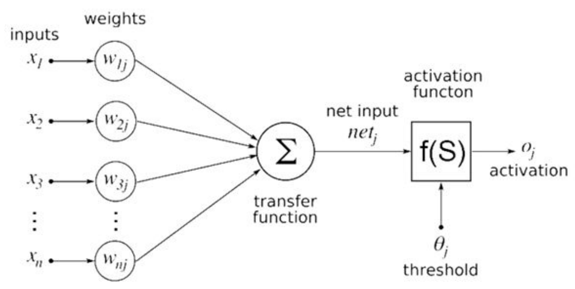


Fig 13: Basic Neural Network

Both of these operations are essentially matrix multiplications. A simple matrix multiplication can be represented by the image below:

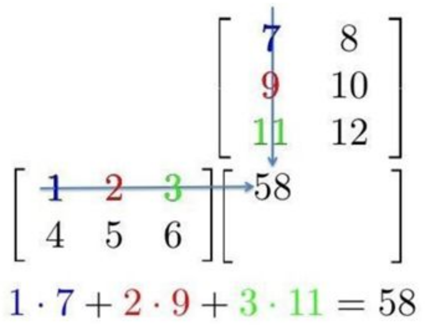


Fig 14: Matrix Multiplication

Here, we can see that each element in one row of the first array is multiplied with one column of the second array. So in a neural network, we can consider the first array as input to the neural network, and the second array can be considered as weights of the network.

J. Training Loss and Optimizer:

The training loss (the error or difference between predictions and true values) is the negative log likelihood (NLL). (The NLL loss in PyTorch expects log probabilities, so we pass in the raw output from the model's final layer.) PyTorch uses automatic differentiation which means that tensors keep track of not only their value, but also every operation (multiply, addition, activation, etc.) which contributes to the value. This means we can compute the gradient for any tensor in the network with respect to any prior tensor.

What this means in practice is that the loss tracks not only the error, but also the contribution to the error by each weight and bias in the model. After we calculate the loss, we can then find the gradients of the loss with respect to each model parameter, a process known as backpropagation. Once we have the gradients, we use them to update the parameters with the optimizer. (If this doesn't sink in at first, don't worry, it takes a little while to grasp! This powerpoint helps to clarify some points.)

The optimizer is Adam, an efficient variant of gradient descent that generally does not require hand-tuning the learning rate. During training, the optimizer uses the gradients of the loss to try and reduce the error ("optimize") of the model output by adjusting the parameters. Only the parameters we added in the custom classifier will be optimized.

```
criterion = nn.NLLLoss()

# Only train the classifier parameters, feature parameters are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.003)
```

Fig 18: Loss and Optimizer function

With the pre-trained model, the custom classifier, the loss, the optimizer, and most importantly, the data, we're ready for training. While training the model on multiple GPUs, one issue can arise with DataParallel: unbalanced GPU usage. Under some settings GPU-1 will be used a lot more than the other GPUs.

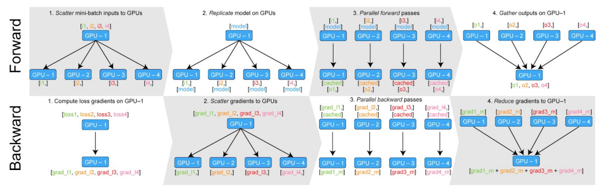

Forward
Backward
1. Scatter mini-batch inputs to GPUs
2. Replicate model on GPUs
3. Parallel forward passes
4. Gather outputs on GPU-1
1. Compute loss gradients on GPU-1
2. Scatter gradients to GPUs
3. Parallel backward passes
4. Reduce gradients to GPU-1

Fig 19: Forward and backward passes with torch.nn.DataParallel

There are two main solution to the imbalanced GPU usage issue:

1. Computing the loss in the forward pass of our model
2. Computing the loss in a parallel fashion.

The first option is the easiest but sometimes we can't use it or it's not practical for various reasons (e.g. our forward pass becomes too complicated and slow because of Python's GIL) so let's talk a bit about the second solution. Along the road we'll learn interesting things about how PyTorch multi-GPU modules work. In that case, the solution is to keep each partial output on its GPU instead of gathering all of them to GPU-1. We well need to distribute our loss criterion computation as well to be able to compute and back propagate our loss. Thankfully for us, Hang Zhang (张航) has open-sourced a nice PyTorch package called PyTorch-Encoding which comprises these custom parallelization functions. We have extracted and slightly adapted this module and we can download here a gist (parallel.py) to include and call from our code. It mainly comprises two modules: *DataParallelModel* and *DataParallelCriterion*. The difference between *DataParallelModel* and *torch.nn.DataParallel* is just that the output of the forward pass (predictions) is not gathered on GPU-1 and is thus a tuple of n_gpu tensors, each tensor being located on a respective GPU. The *DataParallelCriterion* container encapsulates the loss function and takes as input the tuple of n_gpu tensors and the target labels tensor. It computes the loss function in parallel on each GPU, splitting the target label tensor the same way the model input was chunked by DataParallel. Below is an illustration of *DataParallelModel/DataParallelCriterion* internals*:*

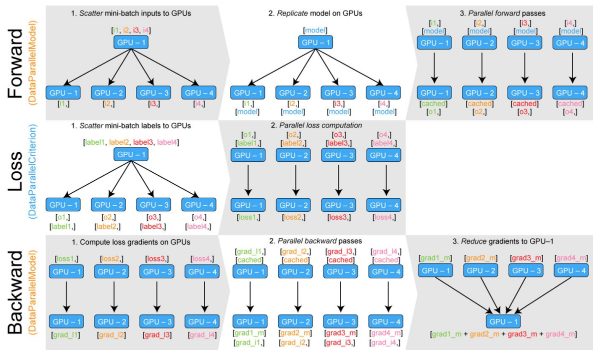

Forward (DataParallelModel)
Loss (DataParallelCriterion)
Backward (DataParallelModel)
1. Scatter mini-batch inputs to GPUs
2. Replicate model on GPUs
3. Parallel forward passes
1. Scatter mini-batch labels to GPUs
2. Parallel loss computation
1. Compute loss gradients on GPUs
2. Parallel backward passes
3. Reduce gradients to GPU-1

Fig 20: Using DataParallelModel and DataParallelCriterion

Here is how to handle two particular cases we may encounter:

1. Our model outputs several tensors: we likely want to disentangle them: output_1, output_2 = zip(*predictions)
2. Sometimes we don't want to use a parallel loss function: gather all the tensors on the cpu: gathered_predictions = parallel.gather(predictions)

K.Saving the Model

When it comes to saving and loading models, there are three core functions to be familiar with:

1. torch.save: Saves a serialized object to disk. This function uses Python's pickle utility for serialization. Models, tensors, and dictionaries of all kinds of objects can be saved using this function.
2. torch.load: Uses pickle's unpickling facilities to deserialize pickled object files to memory. This function also facilitates the device to load the data.
3. torch.nn.Module.load_state_dict: Loads a model's parameter dictionary using a deserialized *state_dict*

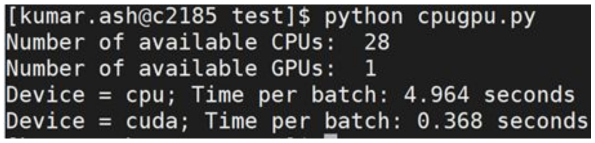
```
checkpoint = {
   'parameters': model.parameters,
   'state_dict': model.state_dict()
}
torch.save(checkpoint,
'models/densenet.pth')
```

IV. RESULTS AND ANALYSIS

A. CPU vs GPU

Computing huge and complex jobs takes up a lot of clock cycles in the CPU. The reason being, CPU takes up the jobs sequentially and it has a fewer number of cores than its counterpart,GPU. But, though GPUs are faster, the time taken to transfer huge amounts of data from CPU to GPU can lead to higher overhead time depending on the architecture of the processors. As CPU takes a lot of time training neural networks, we just ran the code for 1 batch and compared the results with the GPU. We saw there are 28 CPUs available and 1 GPU available on our discovery machine. We used the following command to allocate the gpu node using mobaxterm for accessing machines on Northeastern Discovery:

```
srun -p gpu gres=gpu:1 –pty /bin/bash
module load python/3.6.6
module load cuda/9.2
```



```
[kumar.ash@c2185 test]$ python cpugpu.py
Number of available CPUs:  28
Number of available GPUs:  1
Device = cpu; Time per batch: 4.964 seconds
Device = cuda; Time per batch: 0.368 seconds
```

Fig 21: Time per batch, CPU Vs GPU

We observe that for 1 batch, GPU takes 0.368 seconds while CPU takes 4.964 i.e 1 GPU is almost 14 times faster than 28 CPUs. So, we can save a lot of time for training neural networks for multiple batches and epochs.

B. Single GPU vs Multi GPU

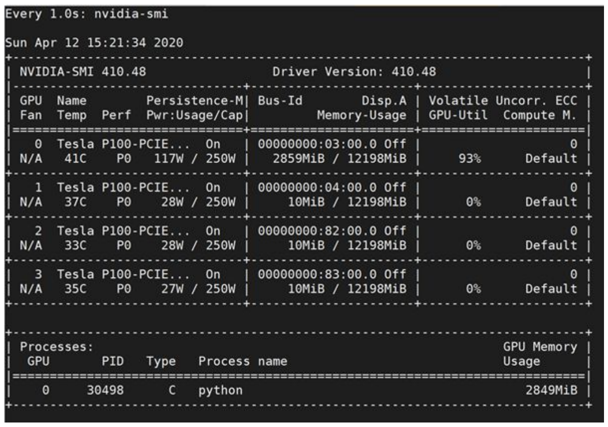
We have implemented Data parallelism with Pytorch to train our model using Multiple GPUs which can further decrease the training time. Data Parallelism in Pytorch is pretty easy to use. Data Parallelism is when we split the mini-batch of samples into multiple smaller mini-batches and run the computation for each of the smaller mini-batches in parallel. Data Parallelism is implemented using torch.nn.DataParallel. One can wrap a Module in DataParallel and it will be parallelized over multiple GPUs in the batch dimension. For training our models, we have used transfer learning using pretrained models Vgg16 and DenseNet121 and compared the results for 1,2 and 4 GPUs.

C. GPU allocation, utilization and training times:

Below is the reservation of nodes for 4 GPUs on Northeastern Discovery:

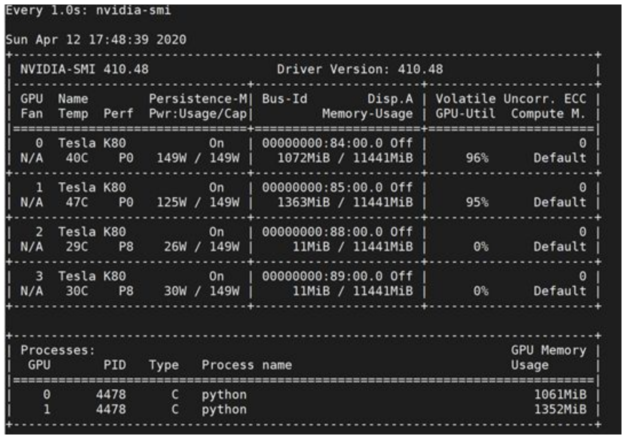
```
srun -p reservation --reservation = CSYE7374_GPU
--gres=gpu:4 --pty --x11 --time=08:00:00 /bin/bash
```

We can see GPU utilization using nvidia-smi command on discovery. Volatile GPU-Util shows the % utilization of GPU



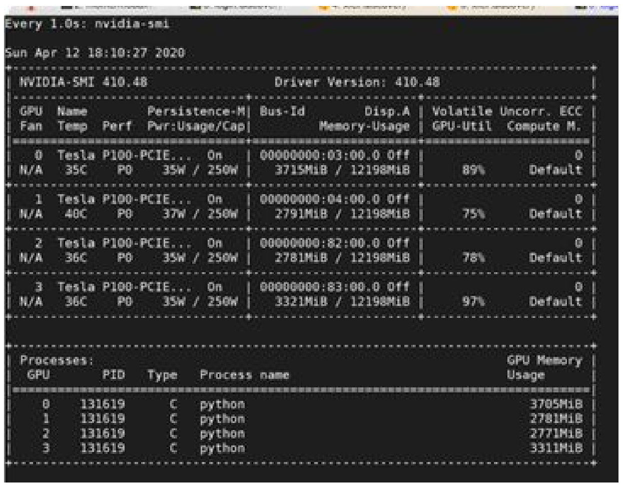
```
Every 1.0s: nvidia-smi

Sun Apr 12 15:21:34 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:03:00.0 Off |                    0 |
| N/A   41C    P0   117W / 250W |   2859MiB / 12198MiB |     93%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla P100-PCIE...  On   | 00000000:04:00.0 Off |                    0 |
| N/A   37C    P0    28W / 250W |     10MiB / 12198MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla P100-PCIE...  On   | 00000000:82:00.0 Off |                    0 |
| N/A   33C    P0    28W / 250W |     10MiB / 12198MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla P100-PCIE...  On   | 00000000:83:00.0 Off |                    0 |
| N/A   35C    P0    27W / 250W |     10MiB / 12198MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     30498      C   python                                      2849MiB |
+-----------------------------------------------------------------------------+
```

Fig 22: Training the model using 1 GPU



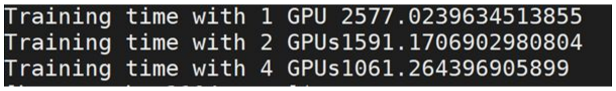
```
Every 1.0s: nvidia-smi

Sun Apr 12 17:48:39 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 00000000:84:00.0 Off |                    0 |
| N/A   40C    P0   149W / 149W |   1072MiB / 11441MiB |     96%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           On   | 00000000:85:00.0 Off |                    0 |
| N/A   47C    P0   125W / 149W |   1363MiB / 11441MiB |     95%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K80           On   | 00000000:88:00.0 Off |                    0 |
| N/A   29C    P8    26W / 149W |     11MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K80           On   | 00000000:89:00.0 Off |                    0 |
| N/A   30C    P8    30W / 149W |     11MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      4478      C   python                                     1061MiB |
|    1      4478      C   python                                     1352MiB |
+-----------------------------------------------------------------------------+
```

Fig 23: Training the model using 2 GPUs



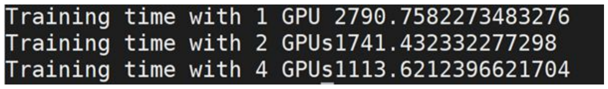
```
Every 1.0s: nvidia-smi

Sun Apr 12 18:10:27 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  On   | 00000000:03:00.0 Off |                    0 |
| N/A   35C    P0    35W / 250W |   3715MiB / 12198MiB |     89%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla P100-PCIE...  On   | 00000000:04:00.0 Off |                    0 |
| N/A   40C    P0    37W / 250W |   2791MiB / 12198MiB |     75%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla P100-PCIE...  On   | 00000000:82:00.0 Off |                    0 |
| N/A   36C    P0    35W / 250W |   2781MiB / 12198MiB |     78%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla P100-PCIE...  On   | 00000000:83:00.0 Off |                    0 |
| N/A   36C    P0    35W / 250W |   3321MiB / 12198MiB |     97%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0    131619      C   python                                     3705MiB |
|    1    131619      C   python                                     2781MiB |
|    2    131619      C   python                                     2771MiB |
|    3    131619      C   python                                     3311MiB |
+-----------------------------------------------------------------------------+
```

Fig 24: Training the model using 4 GPUs



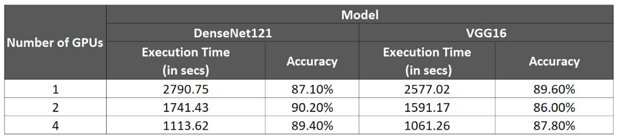
```
Training time with 1 GPU 2577.0239634513855
Training time with 2 GPUs1591.1706902980804
Training time with 4 GPUs1061.264396905899
```

Fig 25: Training times for VGG16 with single and Multi GPUs



```
Training time with 1 GPU 2790.7582273483276
Training time with 2 GPUs1741.432332277298
Training time with 4 GPUs1113.6212396621704
```

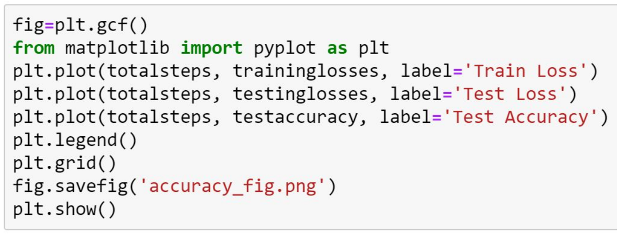
Fig 26: Training times for DenseNet121 with single and Multi GPUs



Number of GPUs	Model			
	DenseNet121		VGG16	
	Execution Time (in secs)	Accuracy	Execution Time (in secs)	Accuracy
1	2790.75	87.10%	2577.02	89.60%
2	1741.43	90.20%	1591.17	86.00%
4	1113.62	89.40%	1061.26	87.80%

Fig 27: Accuracy for models using single and multiple GPUs

We now plot the graph for Train Loss, Test Loss and Test accuracy. We have achieved the accuracy of almost 90% using densenet121.



```python
fig=plt.gcf()
from matplotlib import pyplot as plt
plt.plot(totalsteps, traininglosses, label='Train Loss')
plt.plot(totalsteps, testinglosses, label='Test Loss')
plt.plot(totalsteps, testaccuracy, label='Test Accuracy')
plt.legend()
plt.grid()
fig.savefig('accuracy_fig.png')
plt.show()
```

Fig 28 : Code to plot trainloss, test loss, test accuracy

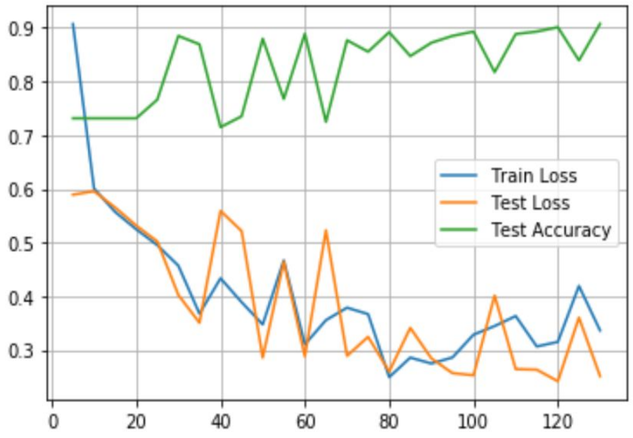


Fig 29: Graph plot for train loss, test loss, test accuracy for DenseNet121

After using multiple GPUs using data parallelism, the execution times reduced drastically as shown below. We used matplotlib library to plot the graph to compare the execution times of 1,2 and 4 GPUs for VGG16 and Densenet121.

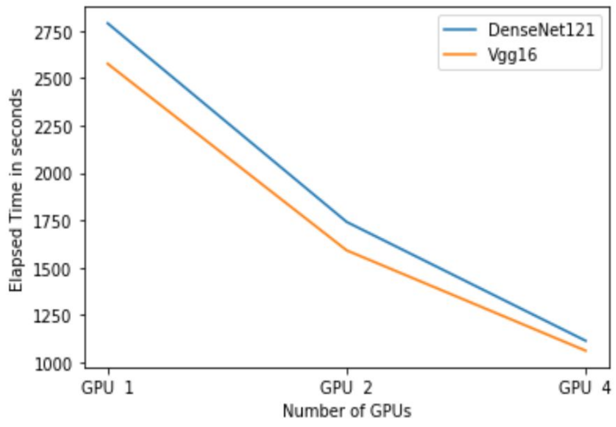
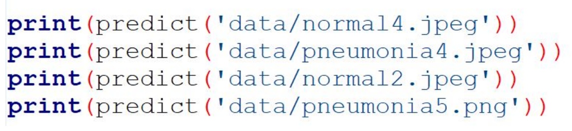


Fig 30: Elapsed Time vs Number of GPUs

C. Model Predictions

Let’s do the prediction now. Our images, which we want to predict, are kept in a data folder and we execute the predict_dense.py file.



```
print(predict('data/normal4.jpeg'))
print(predict('data/pneumonia4.jpeg'))
print(predict('data/normal2.jpeg'))
print(predict('data/pneumonia5.png'))
```

Fig 29: python predcit_dense.py

Input Data to Predict:

We make predictions on the below four images. Two of these are normal x-ray images and two of these are images with pneumonia.

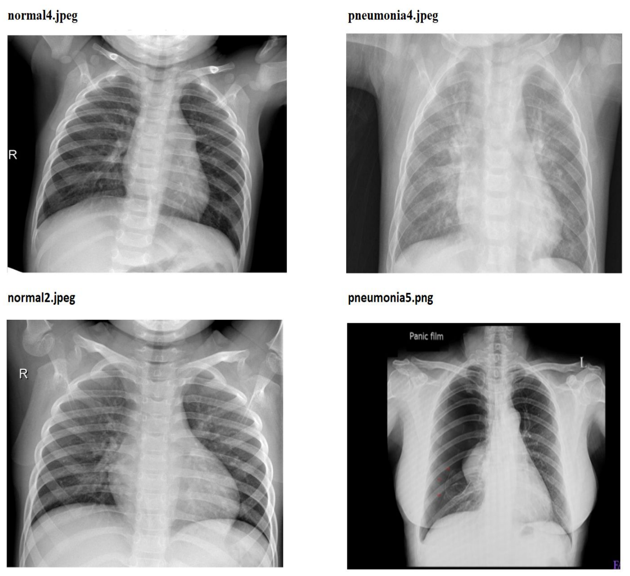
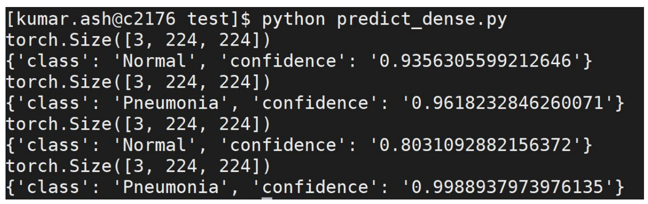


Fig 30: Input x-rays to predict

Output:

We see below that our model is doing pretty good and predicted the results with good accuracy. Confidence is the accuracy of the prediction.



```
[kumar.ash@c2176 test]$ python predict_dense.py
torch.Size([3, 224, 224])
{'class': 'Normal', 'confidence': '0.9356305599212646'}
torch.Size([3, 224, 224])
{'class': 'Pneumonia', 'confidence': '0.9618232846260071'}
torch.Size([3, 224, 224])
{'class': 'Normal', 'confidence': '0.8031092882156372'}
torch.Size([3, 224, 224])
{'class': 'Pneumonia', 'confidence': '0.9988937973976135'}
```

Fig 31: Accuracy achieved for x-ray images

V. CONCLUSION

This project demonstrates the benefits of Data Parallelism using multiple GPUs to overcome computation times. We observe that, end to end training times of our neural networks, using a single CPU, can take several hours. Hence, we can use GPUs for training large neural networks without compromising the accuracy. Using Data Parallelism with pytorch and multiple GPUs can speed up the training process by atlest 250%.

Additional work could include balancing load on multiple GPU machines in which loss can be computed in a parallel fashion.

REFERENCES

[1] CDC Features. Centers for Disease Control and Prevention, Centers for Disease Control and Prevention, 6 Nov. 2017, www.cdc.gov/features/pneumonia/index.html.

[2] Erthal, Jonas. Detecting Pneumonia with Deep Learning: A Soft Introduction to Convolutional Neural Networks. Medium.com, Medium, 9 July 2018, medium.com/datadriveninvestor/detecting-pneumoniawith-deep-learning-a-soft-introduction-to-convolutional-neural-networksb3c6b6c23a88

[3] Rajpurkar, et al. CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning. [1711.05225] CheXNet: RadiologistLevel Pneumonia Detection on Chest X-Rays with Deep Learning, 25 Dec. 2017, arxiv.org/abs/1711.05225.

[4] Ellington, Laura E, et al. Computerised Lung Sound Analysis to Improve the Specificity of Paediatric Pneumonia Diagnosis in ResourcePoor Settings: Protocol and Methods for an Observational Study. BMJ Open, British Medical Journal Publishing Group, 1 Jan. 2012, bmjopen.bmj.com/content/2/1/e000506.short.

[5] Joseph A. Kovacs, and Valerie. Diagnosis of Pneumocystis Carinii Pneumonia: Improved Detection in Sputum with Use of Monoclonal Antibodies — NEJM. New England Journal of Medicine, www.nejm.org/doi/full/10.1056/nejm198803103181001.

[6] Antoni Torres, and Mustafa. INVASIVE DIAGNOSTIC TECHNIQUES FOR PNEUMONIA: PROTECTED SPECIMEN BRUSH, BRONCHOALVEOLAR LAVAGE, AND LUNG BIOPSY METHODS. Infectious Disease Clinics of North America, Elsevier, 9 Apr. 2007, www.sciencedirect.com/science/article/pii/S0891552005702063.

[7] Jing Liu, et al. Lung Ultrasonography for the Diagnosis of Severe Neonatal Pneumonia. Chest, Elsevier, 9 Jan. 2016, www.sciencedirect.com/science/article/pii/S00123692154

[8] Optional: Pytorch Data Parallelism https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

[9] Optional: Data Parallelism https://towaOptional: Data Parallelismrdsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce88270

[10]Towards data Science: Speed up your algorithm https://towardsdatascience.com/speed-up-your-algorithms-part-1-pytorch-56d8a4ae7051

[11] Multi-GPU data and model parallelism https://www.google.co.in/amp/s/glassboxmedicine.com/2020/03/04/multi-gpu-training-in-pytorch-data-and-model-parallelism/amp/

[12] Pytorch Memory Multi GPU debugging https://www.google.co.in/amp/s/blog.paperspace.com/pytorch-memory-multi-gpu-debugging/amp/

[13] Medium: Learn pytorch multi gpu properly https://medium.com/@theaccelerators/learn-pytorch-multi-gpu-properly-3eb976c030ee
Learn PyTorch Multi-GPU properly - The Black Knight

[14] Pytorch-memory-multi-gpu-debugging https://www.google.co.in/amp/s/blog.paperspace.com/pytorch-memory-multi-gpu-debugging/amp/

[15]Training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255

[16] Transfer-learning-from-pre-trained-models https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751

[17]Training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255

[18] Transfer-learning-from-pre-trained-models https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751

[19]Image-classification-with-transfer-learning-and-pytorch https://stackabuse.com/image-classification-with-transfer-learning-and-pytorch/

[20] How-to-scale-training-on-multiple-gpus https://towardsdatascience.com/how-to-scale-training-on-multiple-gpus-dae1041f49d2

[21] mnist_hogwild
https://github.com/pytorch/examples/tree/master/mnist_hogwild

[22]TensorFlow-Multiclass-Image-Classification-using-CNN
https://github.com/MuhammedBuyukkinaci/TensorFlow-Multiclass-Image-Classification-using-CNN-s?files=1

[23]Make-keras-run-on-multi-machine-multi-core-cpu-system
Make Keras run on multi-machine multi-core cpu system

[24]Tensorflow-how-to-optimise-your-input-pipeline-with-queues-and-multi-threading-
https://blog.metaflow.fr/tensorflow-how-to-optimise-your-input-pipeline-with-queues-and-multi-threading-e7c3874157e0

[25]Multithreaded-predictions-with-tensorflow-estimators
https://medium.com/element-ai-research-lab/multithreaded-predictions-with-tensorflow-estimators-eb041861da07

[26] Image-augmentation
https://medium.com/the-artificial-impostor/custom-image-augmentation-with-keras-70595b01aeac

[27] Tips and tricks for gpu and multiprocessing in tensorflow
https://sefiks.com/2019/03/20/tips-and-tricks-for-gpu-and-multiprocessing-in-tensorflow/

[28] Basics of image classification with pytorch
https://heartbeat.fritz.ai/basics-of-image-classification-with-pytorch-2f8973c51864