Utilizing Deep Learning on Chest X – Rays to Detect Pneumonia

Definition

Project Overview and Problem Statement

It is estimated that pneumonia is the #1 reason why children have been hospitalized and approximately 1 million adults are hospitalized every year with 50,000 dying from the disease in the United States (American Thoracic Society, 2018). Furthermore, given that pneumonia can occur as a complication in the recently ongoing COVID-19 pandemic leading to a rise in cases worldwide, tools to help quickly diagnose this complication is worth exploring (World Health Organization, 2020). Pneumonia affects the lungs and makes breathing difficult, which can be fatal (American Thoracic Society, 2018).

Pneumonia can be diagnosed through checking x – rays images, such as the images provided by Mooney (2018) on Kaggle. Qualified professionals, such as doctors, would need to view the images and make the diagnosis. The aforementioned resources can be scarce in locations and facilities lacking qualified personnel and during times of high demand. Machine learning, specifically neural networks, can be leveraged to assess chest x – ray images and classify whether the patient the x -ray image belongs to has pneumonia or not.

Metrics

Metrics used to evaluate models would be the following classification metrics: accuracy, precision, recall, and F1. Accuracy will be the metric that shows overall how many cases have been determined correctly compared to all cases or ((true positives + true negatives) / (true positives + true negatives + false positives + false negatives) [Koehrson, 2018]. Recall measures “the ability of a model to find all relevant cases in the dataset” or “true positives / (true positives + false positives)” [Koehrson, 2018]. Precision measures the ability of a “classification model to identify only the relevant data points” or “true positives / (true positives + false negatives)” [Koehrson, 2018]. The F1 score is also considered to score the quality between precision and recall scores, where high and balanced scores are given a higher F1 compared to more extreme precision or recall scores [Koehrson, 2018].

Analysis

Data Exploration and Exploratory Visualization

The data available from the original Kaggle dataset is as follows: (1) the training set for model training has 1,341 normal x-ray images and 3,875 pneumonia x-ray images, (2) the test set for model training model evaluation has 234 normal x-ray images and 390 pneumonia x-ray images, and (3) the validation set for best model evaluation has 8 normal x-ray images and 8 pneumonia x-ray images.

The images were also examined in terms of manual visual exploration. Upon visualization of groups and NORMAL and PNEUMONIA classified images, the raw images seen are RGB 3 channel images, even though the images can be seen as greyscaled images. However, the RGB 3 channel image setting could be kept as is during processing. After going through several iterations of manually viewing the images, it could be manually confirmed that the images consistently show the chest / lung area: this observation is supported by the dataset contributor in that it was noted that “all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans” (Mooney, 2018).

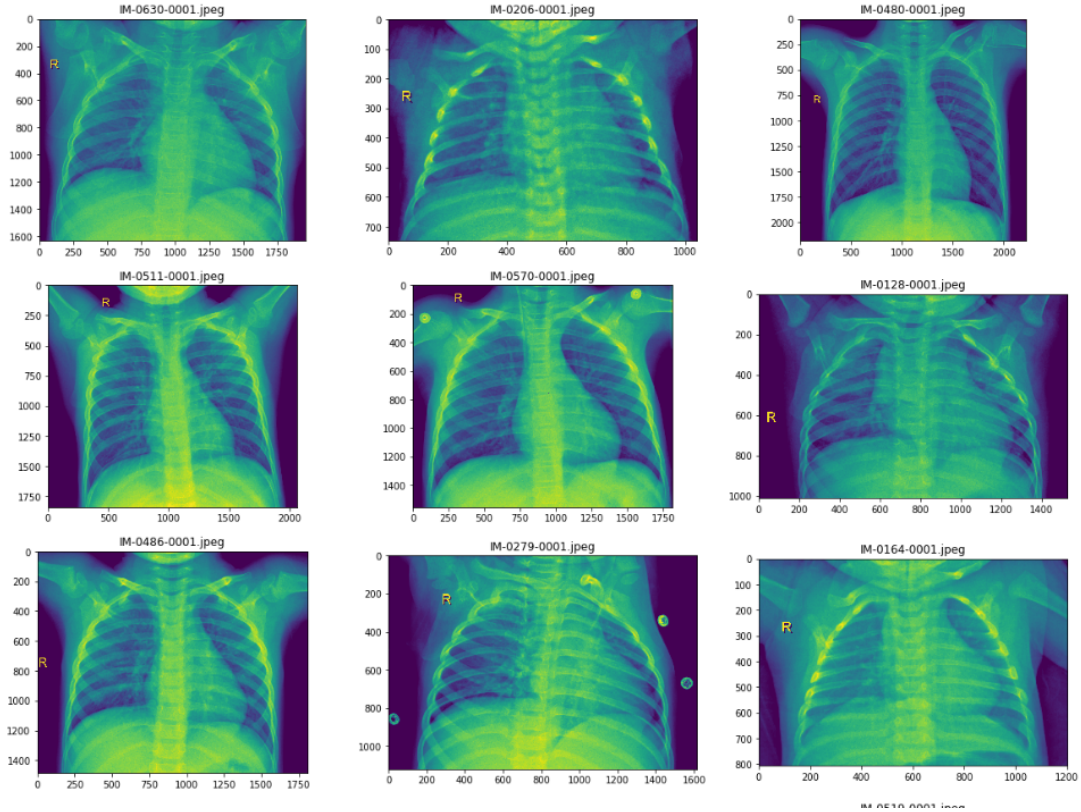

Figure 1. Normal chest x-ray images

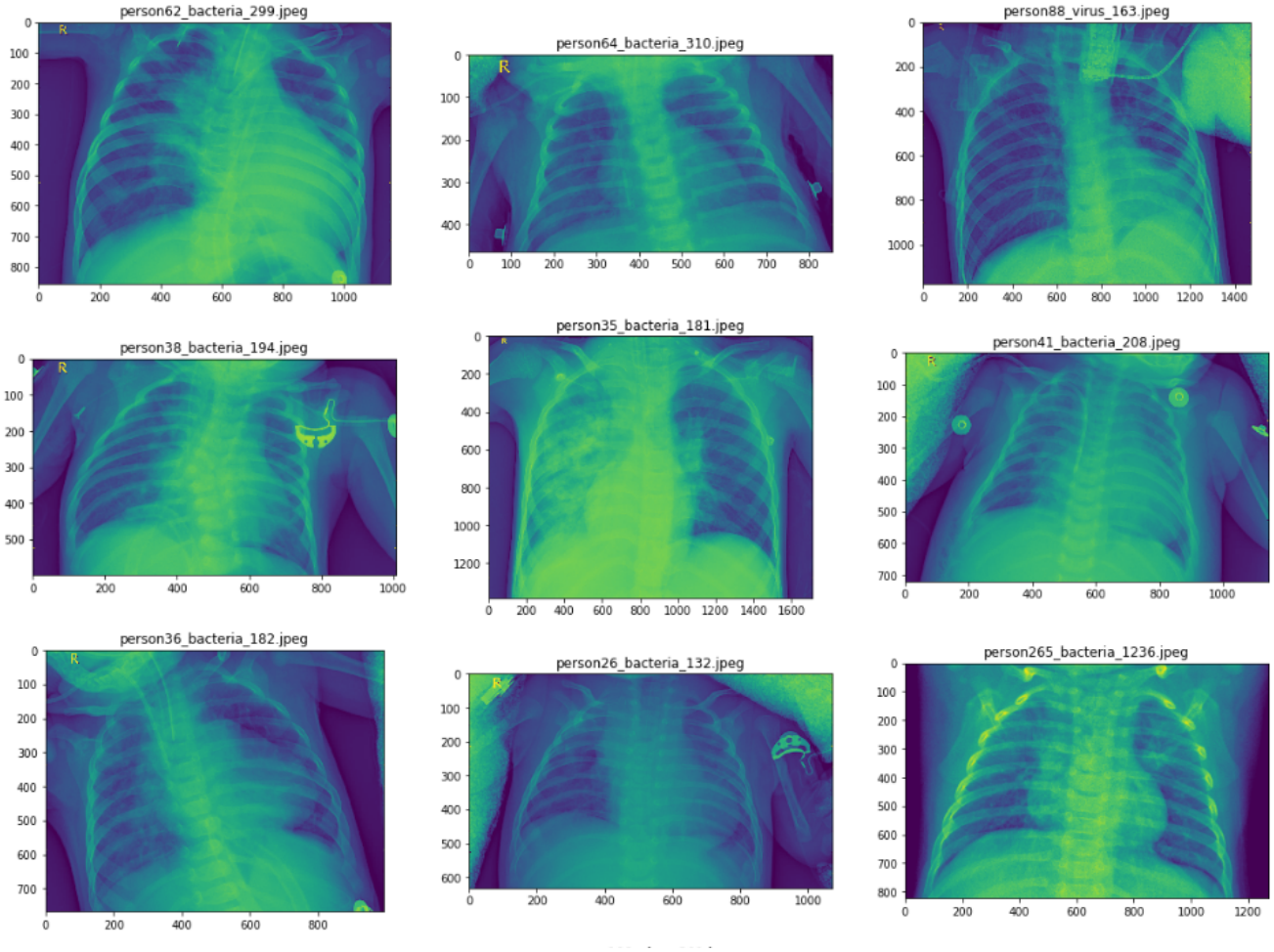

Figure 2. Pneumonia chest x-ray images

Image sizes were examined via a random selection across all images and was found to have varying height and width sizes, so the image sizes will need to be standardized during the preprocessing phase.

```
IM-0234-0001.jpeg size: (1550, 1075)
IM-0553-0001-0003.jpeg size: (2202, 2129)
IM-0256-0001.jpeg size: (1562, 1086)
IM-0160-0001.jpeg size: (1750, 1314)
IM-0304-0001.jpeg size: (1472, 978)
IM-0649-0001.jpeg size: (1512, 1005)
IM-0620-0001-0001.jpeg size: (1832, 1472)
IM-0441-0001.jpeg size: (1472, 1171)
IM-0505-0001-0001.jpeg size: (1400, 1221)
IM-0543-0001.jpeg size: (1778, 1318)
IM-0622-0001-0002.jpeg size: (1626, 1268)
IM-0129-0001.jpeg size: (1384, 1167)
IM-0337-0001.jpeg size: (1584, 1237)
IM-0648-0001.jpeg size: (1604, 1279)
IM-0482-0001.jpeg size: (1252, 1028)
IM-0549-0001-0001.jpeg size: (1696, 1349)
IM-0647-0001.jpeg size: (1214, 994)
IM-0626-0001-0001.jpeg size: (1754, 1178)
IM-0491-0001.jpeg size: (2304, 2086)
IM-0266-0001.jpeg size: (1848, 1495)
```

Figure 3. Normal chest x-ray image file sizes

```
person36_bacteria_182.jpeg size: (992, 768)
person119_virus_225.jpeg size: (920, 512)
person22_bacteria_76.jpeg size: (1128, 1040)
person264_bacteria_1233.jpeg size: (1064, 696)
person258_bacteria_1210.jpeg size: (1056, 776)
person34_bacteria_176.jpeg size: (1380, 801)
person61_bacteria_296.jpeg size: (1264, 1071)
person58_bacteria_276.jpeg size: (1248, 766)
person96_virus_179.jpeg size: (1624, 1304)
person62_bacteria_298.jpeg size: (1372, 948)
person132_virus_266.jpeg size: (1496, 1008)
person266_bacteria_1237.jpeg size: (952, 688)
person88_virus_163.jpeg size: (1472, 1176)
person88_virus_165.jpeg size: (1520, 1176)
person9_bacteria_40.jpeg size: (1248, 728)
person88_virus_163.jpeg size: (1472, 1176)
person143_virus_289.jpeg size: (1368, 992)
person63_bacteria_306.jpeg size: (1048, 736)
person81_virus_153.jpeg size: (1472, 1048)
person258_bacteria_1209.jpeg size: (1056, 856)
```

Figure 4. Pneumonia chest x-ray image file sizes

A manual examination of an image that was resized indicates that no major features are lost during the resizing when resized from 1504 by 1174 to 224 by 224.

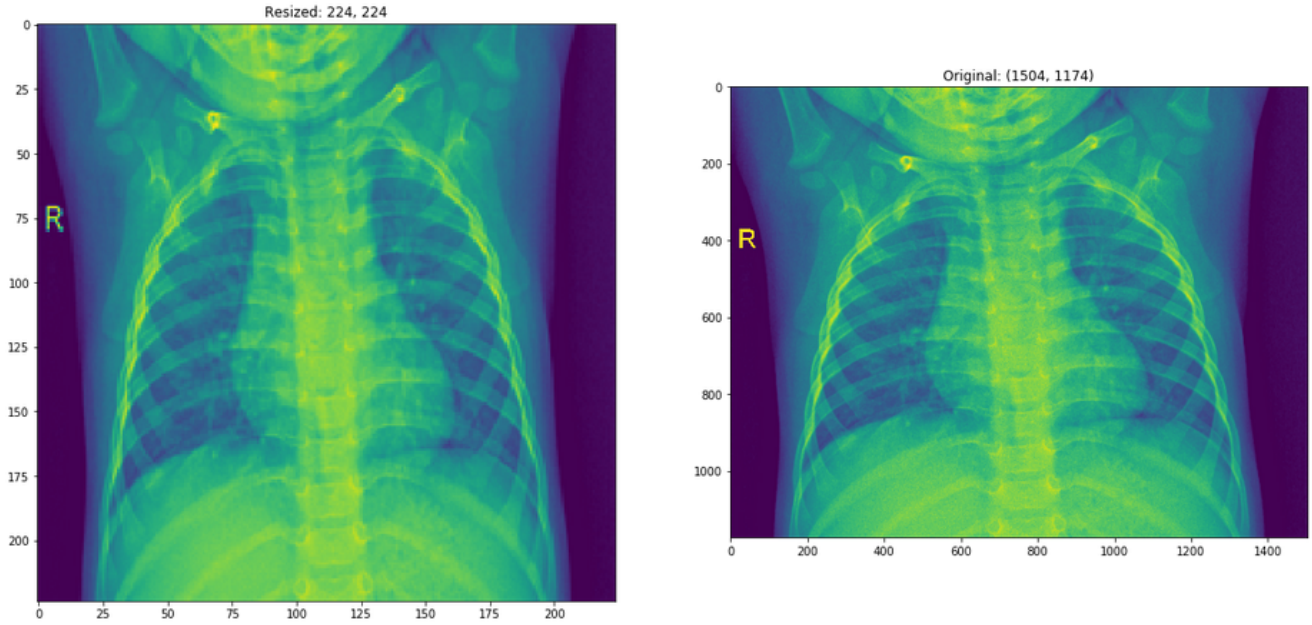

Figure 5. Manually examining resized images

Algorithms and Techniques

Various deep learning neural network architectures were used to classify whether an x-ray image was NORMAL or PNEUMONIA. As guided by the Udacity material on convolutional neural networks and work done in the MNIST dataset multi-layer perceptrons, convolutional neural networks, and transfer learning / pre-trained models available via PyTorch were explored in this classification problem (Udacity, n.d.; LeCun, n.d.; Torch Contributors, “torchvision.models”, 2019). Generally, convolution neural network models would perform better than multi-layer perceptron models, but, given that the data has been vetted for data quality issues and the images have been shown to be relatively consistent, performance improvement between MLP and CNN algorithms on the data might not be as significant (Udacity, n.d., “MLP vs CNN”). A state-of-the-art convolutional neural network with a predefined architecture and hyperparameters that has been pre-trained on an extensive set of image data with many categories would also employed via the transfer learning technique on this dataset (Udacity, n.d., “Transfer Learning”).

CNN architectures include convolutional layers that create filters that each derive a specific feature of an image, which can translate to a specific pattern that can apply to all images in a dataset (Udacity, n.d., “Convolutional Layer”). Since convolutional layers allow for patterns to be found in an image and numerically represent those patterns as a feature vector for each image in the network, it is particularly helpful for datasets that have similar objects in varying locations as long as the features of the object to classify are distinct (Udacity, n.d., “Convolutional Layer”). In the case of PyTorch API, the convolutional layer is created by *Conv2d* method with the *out_channels* parameter defining how many filters are created to highlight a specific feature across all the images from the layer and *kernel_size* parameter defining the size the filter vector for each image would be (Udacity, n.d., “Convolutional Layers in PyTorch”). Multiple convolutional layers can be helpful in that the output filters from the proceeding convolutional layer can be used to further extract filters in the next convolutional layer, which can yield more informative patterns in the following convolutional layer’s

output. Convolutional layers are integrated into a neural network architecture prior to linear layers because convolutional layers are used to create specialized filters that result to additional feature vectors for each image that are fed into the network [Udacity, n.d., “Convolutional Layers (Part 2)”]. Those features from the convolutional layer are used to calculate linear weights and are used in the final classification decision.

By comparison, MLP architectures only leverages the linear calculations of the image’s numeric vector that represents the image and makes predictions using linearly calculated weights across the linear layers, without attempting to create additional feature vectors that represent different patterns within an image, which is done via convolutional layers (Udacity, n.d., “MLP vs CNN”). There is a possibility that datasets with consistent spatial orientation of the object and image resolution clarity might not show too much performance improvement between MLP and CNN architectures since such data consistency works well with MLP structures as well (Udacity, n.d., “MLP vs CNN”).

Benchmark

The benchmark model was adapted from the Udacity convolutional neural network course and was the first and relatively simpler model introduced to provide a contrast to the more complex convolutional neural network (Udacity, n.d.). The exact configuration of the multi-layer model using the PyTorch API is as follows (Udacity., “Multi-Layer Perceptron, MNIST”, 2019):

```
“MLPNet(
  (fc1): Linear(in_features=150528, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=2, bias=True)
  (dropout): Dropout(p=0.2)
)

Loss function (categorical cross-entropy)
```

```
Optimizer (stochastic gradient descent) and learning rate = 0.01”
```

Methodology

Data Preprocessing

Steps were taken to standardize training, test, and validation data to ensure data consistency across all stages of model training, testing, and validation. As noted in an aforementioned section, data preprocessing is needed to standardize image sizes. Transformations to the data are adapted from a Kaggle kernel utilizing the source dataset and are the following: (1) image resizing to 224 by 224, (2) normalization to 0.5 [Salvation23, 2020; Torch Contributors., “torchvision.transforms”, 2019]. Consideration for the benefit of image augmentation on the model performance is also given in regards to the following augmentation features: (1) random horizontal flip of images and (2) random rotation of images by 10 degrees – these changes would only apply to the training dataset [Udacity., “cifar10_cnn_augmentation.ipynb”, 2018; Torch Contributors., “torchvision.transforms”, 2019]. Image augmentation might not be as helpful for this dataset given that some quality control from the dataset publishers have been completed and images are fairly consistent in what is shown in the x-ray and the general area the chest and lungs appear on the image.

Implementation

The implementation of the metrics, algorithms, and techniques can be seen in the ‘Data_Processing_And_Training.ipynb’. In regards to the datasets used the following was implemented and adjusted due to limited compute resources: (1) training dataset – only 500 images were used from the normal and pneumonia classifications due to compute resource considerations for model training, (2) test dataset – all data provided was used because the image numbers were under 500 per classification folder, (3) validation dataset – all data was used since 8 images were available in each classification folder. The c*alculate_metrics* function adapted from Salvation23’s Kaggle kernel is used

to assess model performance across testing and validation datasets via the metrics accuracy, precision, recall, and f1 (Salvation23., 2020; PyTorch, “Training a Classifier.”, 2017). Given that the PyTorch API is used across all models, the *train* method is used to train all models. The implementation of the algorithms and techniques used is as follows:

Model	Architecture	Change / Reference

Baseline – MLP – 1 Hidden Layer	`Net(`
	`  (fc1): Linear(in_features=150528, out_features=512, bias=True)`
	`  (fc2): Linear(in_features=512, out_features=512, bias=True)`
	`  (fc3): Linear(in_features=512, out_features=2, bias=True)`
	`  (dropout): Dropout(p=0.2, inplace=False)`
	`)`
		Baseline model taken from an introductory model. (Udacity., “Multi-Layer Perceptron...”, 2019)

MLP – Image Augmentation	`Net(`
	`  (fc1): Linear(in_features=150528, out_features=512, bias=True)`
	`  (fc2): Linear(in_features=512, out_features=512, bias=True)`
	`  (fc3): Linear(in_features=512, out_features=2, bias=True)`
	`  (dropout): Dropout(p=0.2, inplace=False)`
	`)`
		Change made to data pre-processing step. (Udacity., “cifar10_cnn_augmentation...”, 2018)

MLP – 2	`Net(` `  (fc1): Linear(in_features=150528, out_features=256,`	Added

Hidden Layers	bias=True)
	  (fc2): Linear(in_features=256, out_features=512, bias=True)
	  (fc3): Linear(in_features=512, out_features=64, bias=True)
	  (fc4): Linear(in_features=64, out_features=2, bias=True)
	  (dropout): Dropout(p=0.2, inplace=False)
	)
		another hidden layer to the baseline MLP model.

CNN – 3 Convolution layers	Net(
	  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
	  (fc1): Linear(in_features=50176, out_features=512, bias=True)
	  (fc2): Linear(in_features=512, out_features=2, bias=True)
	  (dropout): Dropout(p=0.2, inplace=False)
	)
		Implemented CNN for this dataset adapted from Udacity sample for CNN introduction. (Udacity., “cifar10_cnn_augmentation”, 2018)

CNN – Learning rate change +	Net(
	  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1,
		Changing the learning rate and increasing

Increased epochs	1), padding=(1, 1))
	  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
	  (fc1): Linear(in_features=50176, out_features=512, bias=True)
	  (fc2): Linear(in_features=512, out_features=2, bias=True)
	  (dropout): Dropout(p=0.2, inplace=False)
	)
		the epochs to adjust to the smaller learning rate changes.

CNN – 4 Convolution layers	Net(
	  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
	  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
	  (fc1): Linear(in_features=12544, out_features=512, bias=True)
	  (fc2): Linear(in_features=512, out_features=2, bias=True)
	  (drop): Dropout(p=0.2, inplace=False)
	)
		Adding more complexity to the CNN model. Adapted from Salvation23's (2020) Kaggle kernel.

CNN –	VGG(	Leveraging a

Transfer learning - Vgg16	(features): Sequential((0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (1): ReLU(inplace=True) (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (3): ReLU(inplace=True) (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (6): ReLU(inplace=True) (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (8): ReLU(inplace=True) (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (11): ReLU(inplace=True) (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (13): ReLU(inplace=True) (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (15): ReLU(inplace=True) (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	peer-reviewed CCN that has more convolutional layers, pooling layers, etc. via transfer learning. (Simonyan & Zisserman, 2014; Torch Contributors, “torchvision. models”, 2019)

```
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3),
stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096,
bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096,
bias=True)
    (4): ReLU(inplace=True)
```

```
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=2,
bias=True)
  )
)
```

Figure 6. Details of algorithm / technique implementation.

Overall, depending on the changes made per training and testing iteration for each algorithmic variation the workflow for implementation consisted of transforming the train, test, validation data via the torchvision.transforms api, defining the neural network model architecture via the torch.nn api, training the model via the *train* method, and calculate the metrics via the *calculate_metrics* method (Torch Contributors., “torchvision.transforms”, 2019; Torch Contributors, “torch.nn.”, 2019;).

Refinement

As shown in Figure 6, the complexity of the neural network was increased from the initial linear (Linear) layers at *Baseline – MLP – 1 Hidden Layer* to include more complexity in the form of layers that included convolutional (Conv2d) layers and accompanying transformations of those layers, such as pooling (MaxPool2d) and layers to avert overfitting (ReLU) [Udacity., “Machine Learning Engineer Nanodegree...”, n.d.]. Refinement from the baseline model to the final model included changes to the architecture as well as attempts at preprocessing and parameter changes along the way. From “Baseline – MLP – 1 Hidden Layer” to “MLP – Image Augmentation”, a change was added to the preprocessing step to see whether image augmentation significantly improves the baseline model. The progression to “MLP – 2 Hidden Layers” involved adding one more linear layer to the baseline model. The progression to “CNN – 3 Convolution layers” involved adding aspects of a convolutional neural network to the baseline multi-layer perceptron. The progression to “CNN – Learning rate change + Increased epochs” involved changing parameters on the previous CNN model to see if there would be

performance improvement. The progression to “CNN – 3 Convolution layers” involved adding additional complexity to the model per Salvation23’s (2020) Kaggle kernel and the initial CNN model. The progression to “CNN – Transfer learning – Vgg16” involved utilizing a more complex, peer-reviwed CNN model and pre-trained weights derived from the ImageNet Challenge 2014 data (Simonyan & Zisserman, 2014).

Results

Model Evaluation and Validation

The results in Figure 7 were obtained from the running the models on data shown in ‘Data_Preprocessing_And_Training.ipynb’.

Model Description	*Learning Rate*	*Epochs*	**Accuracy**	**F1**	**Precision**	**Recall**
Baseline – MLP – 1 Hidden Layer	0.01	10	0.81	0.86	0.78	0.97
MLP – Image Augmentation	0.01	10	0.83	0.87	0.86	0.87
MLP – 2 Hidden Layers	0.01	10	0.80	0.86	0.78	0.97
CNN – 3 Convolution layers	0.01	10	0.82	0.86	0.88	0.83
CNN – Learning rate change + Increased epochs	0.001	20	0.64	0.61	0.95	0.45
CNN – 4 Convolution layers	0.01	10	0.63	0.77	0.63	1.00
CNN – Transfer learning - Vgg16	0.01	10	0.83	0.88	0.80	0.97

Figure 7. Testing data results that was leveraged in training parameters

In reference to the baseline MLP model, image augmentation and adding an additional layer was not found to significantly improve the predictive performance of the model on the data. There was no performance improvement in adding one more linear later in the *MLP – 2 Hidden Layers* performance. While image augmentation improved precision, recall performance was decreased, resulting in a one percentage performance increase in the F1 score based on precision and recall.

The initial CNN model showed slight improvement in accuracy and precision, which indicates that false positives were less likely with the CNN model. Attempts to add an additional convolution layer or edit parameters such as learning rate and epochs did not produce performance improvement. The additional convolutional layer did not help in this case given the relative consistency of the images and ended yielding more false positive predictions as indicated by the lower precision. The number of epochs used for training was not large and thus might not have been able to take advantage of the finer grained learning rate of 0.001 vs 0.01. Given the 10x decrease of the learning rate ad provided more compute resources, the number of epochs that might have been more appropriate to assess the learning rate change might be 100 rather than 20.

Of the CNN models, the *CNN – Transfer learning – Vgg16* model performed the best compared to the baseline: reasons for this includes the weights pretrained from the diverse ImageNet dataset and the peer-reviewed architecture that has been shown to generalize effectively to other image datasets (Simonyan & Zisserman, 2014). There was not significant improvement when using CNN models compared to MLP models due to the good image quality and consistent image feature orientation (i.e., the alignment and placement of the chest and lungs) [Udacity, n.d., “MLP vs CNN”].

Justification

The final dataset that was not seen by the model during training ans testing was the validation dataset as shown in Figure 8 in which the baseline and the best performing model was selected.

Model Description	**Accuracy**	**F1**	**Precision**	**Recall**
Baseline – MLP – 1 Hidden Layer	0.50	0.67	0.50	1.0
CNN – Transfer learning - Vgg16	0.88	0.88	0.8	1.0

Figure 8. Final validation data results

It is clear that the final CNN model vastly improved upon the baseline model predictions in every measure. The better precision, which indicated the reduction of false positives, contributed to better accuracy and a F1 score, indicating that precision and recall improved in the final model.

References

American Thoracic Society. (2018). Top 20 Pneumonia Facts—2018. Retrieved from https://www.thoracic.org/patients/patient-resources/resources/top-pneumonia-facts.pdf

Koehrsen, W. (2018, March 10). Beyond Accuracy: Precision and Recall. Retrieved from https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c

LeCun. Y. (n.d.). The MNIST database of handwritten digits. Retrieved from http://yann.lecun.com/exdb/mnist/

Mooney, P. (2018, March 24). Chest X-Ray Images (Pneumonia). Retrieved from https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia

Salvation23. (2020, February 20). XRay: CNN: pyTorch. Retrieved from https://www.kaggle.com/salvation23/xray-cnn-pytorch

Simonyan, K., & Zisserman, A. (2014, September 4). Very Deep Convolutional Networks for Large-Scale Image Recognition. Retrieved from https://arxiv.org/pdf/1409.1556.pdf

PyTorch. (2017). Training a Classifier. Retrieved from https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Torch Contributors. (2019). torchvision.models. Retrieved from https://pytorch.org/docs/stable/torchvision/models.html

Torch Contributors. (2019). torch.nn. Retrieved from https://pytorch.org/docs/stable/nn.html

Torch Contributors. (2019). torchvision.transforms. Retrieved from https://pytorch.org/docs/stable/torchvision/transforms.html

Udacity. (2018, November 30). cifar10_cnn_augmentation.ipynb. Retrieved from

https://github.com/udacity/deep-learning-v2-pytorch/blob/master/convolutional-neural-networks/cifar-cnn/cifar10_cnn_augmentation.ipynb

Udacity. (2019, May 3). Multi-Layer Perceptron, MNIST. Retrieved from https://github.com/udacity/deep-learning-v2-pytorch/blob/master/convolutional-neural-networks/mnist-mlp/mnist_mlp_solution.ipynb

Udacity. (2019, May 8). Transfer Learning. Retrieved from https://github.com/udacity/deep-learning-v2-pytorch/blob/master/transfer-learning/Transfer_Learning_Solution.ipynb

Udacity. (n.d.). Machine Learning Engineer Nanodegree - 2. Additional Materials: Convolutional Neural Networks.

World Health Organization. (2020). Coronavirus. Retrieved from https://www.who.int/health-topics/coronavirus