Machine Learning Engineer Nanodegree Capstone Project

AnnMargaret Tutu
June 27th, 2020

Definition

Project Overview

The novel coronavirus (COVID-19) has brought to light many bottlenecks in the medical testing supply chain. Even after the rapid proliferation of testing availability enabled by the enactment of the Defense Production Act and containment efforts led by state Governors across the country, the sheer volume of *serological* test kits needed — the default protocol for pre-screening upper respiratory viral infections — to effectively contain the virus will likely increase as the nation begins to open back up again.

Serological tests are based on reverse transcription of polymerase chain reaction (rPT-PCR) **[3]** and functions on top of an ecosystem of brand-specific instrumentation (e.g., swabs, reagent, etc.) for collecting and processing samples. Further, lab testing can take days to process, is prone to contamination **[1]** and requires confirmation phases to minimize the number of false positives/negatives. U.S. Medical experts call for testing at a rate in excess of 5 million tests per day [Harvard Public Health] and while new approaches, like pooling, have helped labs to process samples in batches, errors are more costly.

A recent article published by McKinsey **[2]** calls for making the medical supply chain more agile. Specifically, the article advises “digitalizing the medical supply chain”; this will be especially vital in seasons where multiple viral infections circulate concurrently. Digitalization of infectious disease screening has potential to better support other key components of an effective containment strategy, such as contact tracing, and lessen the burden on ICUs during winter months.

Researchers and scientists provide a growing body of literature to support the value proposition of using radiographic imaging more robustly to pre-screen upper respiratory infectious diseases at scale. Together with advances in artificial intelligence (AI) **[8]** — and specifically, machine and deep learning along with transfer learning (TL) using pre-trained deep Convolutional Neural Networks (CNNs) — high-performance, machine aided chest X Ray (CXR) image classification demonstrates a less error-prone solution, for less money, with existing infrastructure in place at most hospitals and medical facilities.

Problem Statement

In their paper, **[4]** present a convolutional neural network (CNN) architecture called *COVIDNet*, capable of differentiating between “normal” CXR images and CXRs with either Pneumonia or COVID-19 present. The network gains increased representational capacity for the COVID-19

classification task from the design pattern used to implement the core module, the PEPX unit — a light-weight residual unit. These units are stacked successively in blocks increasing in size exponentially, each block followed with a proceeding pooling layer. The authors illustrate the network architecture with the following visual:

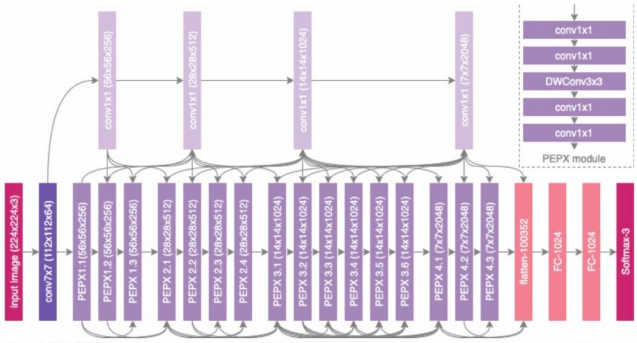


Figure 2. **COVID-Net Architecture.** High architectural diversity and selective long-range connectivity can be observed as it is tailored for COVID-19 case detection from CXR images. The heavy use of a projection-expansion-projection design pattern in the COVID-Net architecture can also be observed, which strikes a strong balance between computational efficiency and representational capacity.

COVID-Net Illustration [(L. Wang and A. Wong., 2020)]

The objective of this project is to incorporate learning from the Udacity Machine Learning Engineer Nanodegree lesson modules together with learnings from a recent course I completed — AI for Medical Diagnosis (AI4MD) offered by deeplearning.ai — to develop, implement and assess an experimental design that meets or exceeds results from the benchmark model **[4]** (described in a later section) on the CovidX dataset (also discussed in more depth later) by leveraging a popular data science library built on top of PyTorch called *fast.ai* **[9]**. The culmination of the work will be a more lightweight set of weights, deployed as an API inference service on the web using the Streamlit framework.

Metrics

To evaluate performance on the validation and test sets, and give special attention to minimizing false negatives, I'll be aggregating the following **metrics**:

Metric	Function
Accuracy	(TP+TN)/(TP+FP+FN+TN)
Precision/PPV	TP/(TP+FP)
Recall/Sensitivity	TP/(TP+FN)
F-score	2*(Recall * Precision) / (Recall + Precision)
Specificity	TN/(TN+FP)

Where:

TP → True Positives
TN → True Negatives
FP → False Positives
FN → False Negatives

This article along with lesson modules from AI4MD helped me wrap my head around these key clinical metrics and the benefit of optimizing for each in scenarios of high prevalence.

Additionally, following training and before deployment, I will perform a quick inference test with the provided labeled test set to justify the experimental design. Fast.ai simplifies the process of applying Test Time Augmentations (TTA) and calculating metrics for test data.

Analysis

Data Exploration

COVIDx Open Data Set

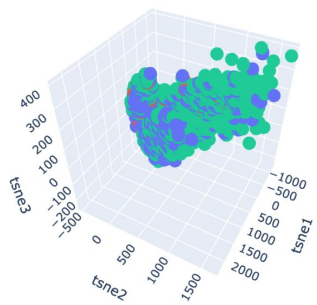
Authors of the paper have also been gracious enough to open-source anonymized patient samples, scraped from various data science platforms on the web, to train and test the model. Further, the project provides detailed instructions and a template Jupiter notebook to preprocess the high-volume dicom data to a more compressed format for classification (e.g., .jpg images from 3D CXR .dem files; a major contribution to the research community, as COVID-19-related imagery is scarce). Roughly ⅓ of the data is openly available on git and ready for download. The other ⅔ of the data is available via the Kaggle competition page for the RSNA Pneumonia Detection Challenge. The data distribution, across 13,870 patient patient case, is as follows:

Split	COVID-19*	Pneumonia	Normal	Total
Training	473	5454	7966	13893
Testing	100	594	885	1579

The CovidNet open-source project provides instructions for sourcing data from various challenges hosted on Kaggle, as well as a template notebook for: (1) generating JPG images (.jpg) from dicom files (.dem), as well as (2) ensuring that patient ids do not appear in *both* the training and testing sets. The AI for Medical Diagnosis illuminates the pitfalls of unique patient ids appearing in both the training and test sets; this leads to increased training bias and performance drops at inference time. The instructions can be found on the readme page for the COVID-Net project, along with other assets, on GitHub.

Exploratory Visualization

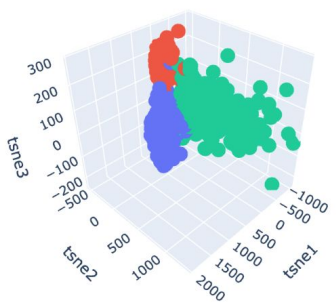
With so many files, I wanted a more compact visualization of label frequency, and to check for potential outliers/mislabeled samples. To gain insight, I projected the image embeddings of both the training and test sets using sklearn's t-SNE to reduce the dimensionality of our features and maximize the probability of similarities:



Train/Valid Set Projection (t-SNE)

	label_id	tsne1	tsne2	tsne3
count	13893.000000	13893.000000	13893.000000	13893.000000
mean	1.358526	-0.118535	-0.068027	-0.071834
std	0.545984	371.060181	306.236084	94.450165
min	0.000000	-1156.285400	-497.206970	-243.185684
25%	1.000000	-203.549957	-259.607788	-78.398254
50%	1.000000	-81.161369	-4.273879	-6.475988
75%	2.000000	108.724785	146.295334	77.567047
max	2.000000	2272.311279	1665.314575	398.048767

Statistical moments of Train/Valid Data/NeighborEmbedding



Test Set Projection (t-SNE)

	label_id	tsne1	tsne2	tsne3
count	1579.000000	1579.000000	1579.000000	1579.000000
mean	1.312856	-0.519371	-0.183029	-0.198344
std	0.584685	439.498016	269.179657	72.817375
min	0.000000	-990.662415	-523.548645	-234.486694
25%	1.000000	-252.217171	-157.574593	-51.469576
50%	1.000000	-141.203369	-52.319725	-9.015071
75%	2.000000	148.781471	104.572750	38.514585
max	2.000000	1945.252319	1357.113037	288.552765

Statistical moments of Test Data/NeighborEmbedding

t-SNE minimizes the Kullback-Leibler (KL) divergence of two distributions: (1) that of the pairwise comparison/similarity probability of original high-dimensional features; and (2) that of the pairwise comparison of features in low-dimensional space. Based on the above visualizations, the lack of variance between labels in training data might have some impact and I tried to keep that in mind when selecting hyper parameters (e.g., epochs/cycle length) as I trained (and retrained) my model.

Moreover, the training pipeline detailed in the following sections incorporates TL to share knowledge about low-level features (e.g., lines, contours, etc.) for the task at hand.

Algorithms and Techniques

The base architecture for this TL pipeline is a *scalable* EfficientNet — specifically, the feature extraction layers from the *EfficientNet-b1* architecture — originally pre-trained on the ImageNet dataset. Authors Tan and Le of the EfficientNet paper arrived at a novel and general rule of thumb: a user-specified coefficient (**Φ**) can be used to manage compound scaling of width, depth, and resolution for a base architecture:

$$
\begin{aligned}
\text{depth: } & d = \alpha^{\phi} \\
\text{width: } & w = \beta^{\phi} \\
\text{resolution: } & r = \gamma^{\phi} \\
\text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\
& \alpha \geq 1, \beta \geq 1, \gamma \geq 1
\end{aligned}
$$

Further, Tan and Le used *Neural Architecture Search (NAS)* to infer a base architecture that reduces the necessary Floating Point Operations (FLOPS) of convolutional operations while still increasing accuracy.

NAS eventually arrived at the base architecture — *EfficientNet-b0* — found within the same search space as the MobileNetV2 architecture and applied the compound scaling mechanism to devise a line of EfficientNets (b0–b7). In their research, the authors demonstrate the performance gains of EfficientNets over most popular ConvNets, both from scratch on ImageNet and other popular open-source training sets typically used for benchmarking purposes:

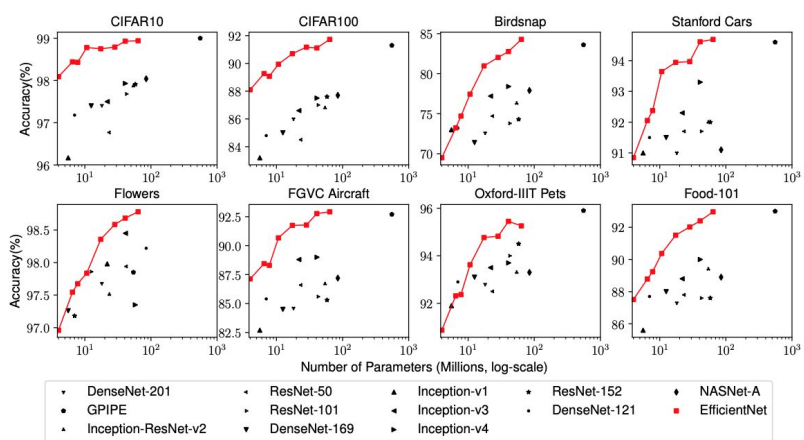
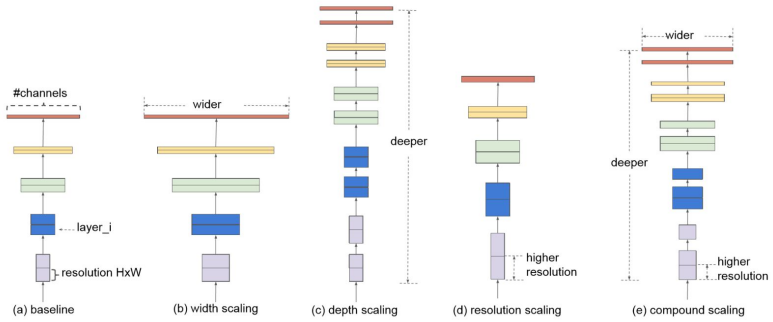


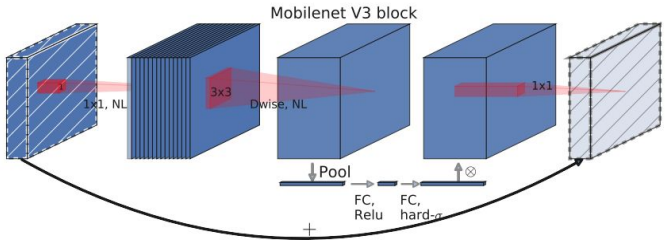
Figure 6. **Model Parameters vs. Transfer Learning Accuracy –** All models are pretrained on ImageNet and finetuned on new datasets.

The following figure illustrates how compound scaling works for EfficientNets **[7]** in contrast to legacy networks, like ResNets.



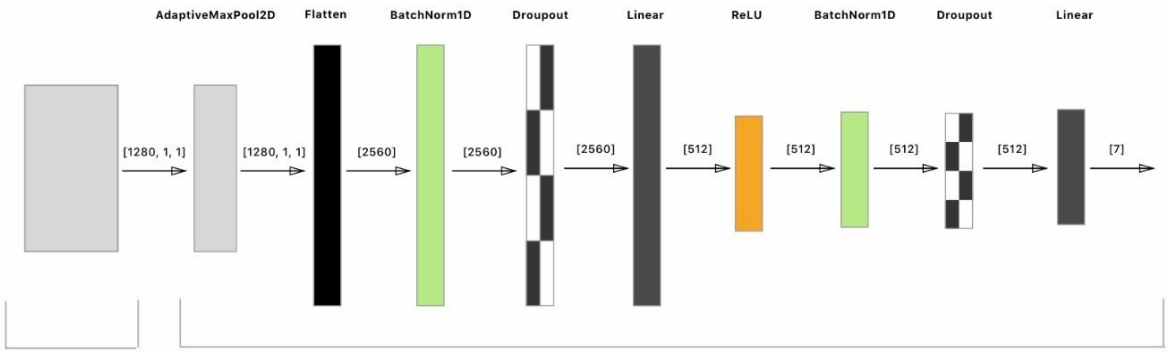
Compound Scaling Illustration [(Tan and Le, 2019)]

Digging deeper into the anatomy of EfficientNets, these networks are mainly comprised of Mobile Inverted Bottleneck Blocks (**MBConv6**) blocks — which were first presented in this **[paper]** — but also incorporate squeeze-and-excitation (SEB) layers into the unit:



MobileNet v2 Block with SEB → MobileNetV3 [(A. Howard et al., 2019)]

Regular residual blocks use skip-connections to forward activation identity up the network with a wide-narrow-wide approach to channel layering, whereas MBConv6 blocks (inverted residual blocks) do the inverse and take a narrow-wide-narrow approach to channel layering, using SEBs. SEBs improve upon how residual blocks weight their channels (typically, weighting all channels equally) by injecting a mini-connected block/network that calculates a weight vector to model interchannel dependencies within the block. I found [this post] very helpful for unpacking the functionality of SEBs. The training pipeline uses the fast.ai library that effectively swaps the original fully-connected layers of the base architecture with with a custom classifier:



As illustrated above, the Fast.ai classifier is a module list that applies an AdaptiveMaxPool2D layer on final features before flattening to a rank-1 tensor and applying batch normalization, non-linearity and dropout to produce the final feature embedding. Results from research presented in **[5]** achieved descent results compared to COVIDNet on the same dataset using far less computational resources; leveraging fast.ai to perform TL with a pretrained ResNet-50 base architecture, cyclical learning rates **[4]**, and human-intervention to implement a progressive resizing strategy (discussed later). To extend the training pipeline of this project, I progressively increase the size of the input images from 240px to 300px.

Methodology

Data Preprocessing

After refining and completing the data generation process, I faced a new issue. Evidenced by the data label distribution table from above, the COVID-19 label is underrepresented in the training data. To address this problem, I utilized this custom PyTorch data sampler to oversample COVID-19 examples in each batch:

```
class ImbalancedDatasetSampler(torch.utils.data.sampler.Sampler):
    """Samples elements randomly from a given list of indices for imbalanced dataset
    Arguments:
        indices (list, optional): a list of indices
        num_samples (int, optional): number of samples to draw
    """

    def __init__(self, dataset, indices=None, num_samples=None):

        # if indices is not provided,
        # all elements in the dataset will be considered
        self.indices = list(range(len(dataset))) \
            if indices is None else indices

        # if num_samples is not provided,
        # draw `len(indices)` samples in each iteration
        self.num_samples = len(self.indices) \
            if num_samples is None else num_samples

        # distribution of classes in the dataset
        label_to_count = {}
        for idx in self.indices:
            label = self._get_label(dataset, idx)
            for l in label:
                if l in label_to_count:
                    label_to_count[l] += 1
                else: label_to_count[l]=1

        # weight for each sample
        weights = [1.0 / min([label_to_count[l] for l in self._get_label(dataset, idx)])
                   for idx in self.indices]
        self.weights = torch.DoubleTensor(weights)

    def _get_label(self, dataset, idx):
        return dataset.y[idx].obj #for category obj

    def __iter__(self):
        return (self.indices[i] for i in torch.multinomial(
            self.weights, self.num_samples, replacement=True))

    def __len__(self):
        return self.num_samples
```

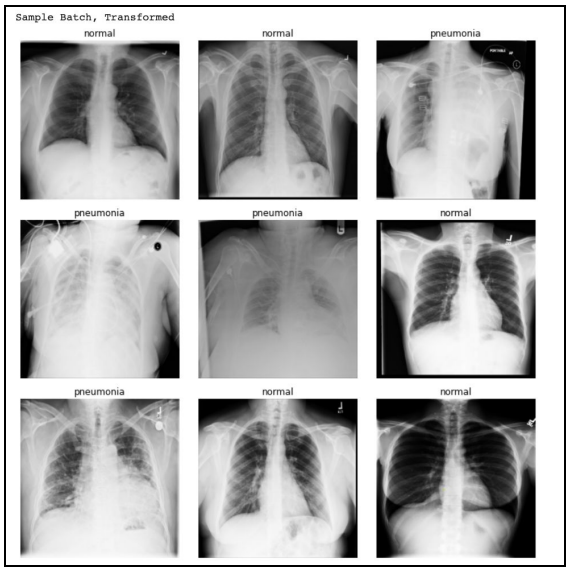
I opted against using the same transforms as the benchmark model. This decision was informed by insights garnered from the AI for Medical Diagnosis; specifically, that with radiographic imaging, minimal transforms are necessary to achieve satisfactory results.

The following fast.ai transforms (effectively wrappers for PyTorch trtransforms) were applied:

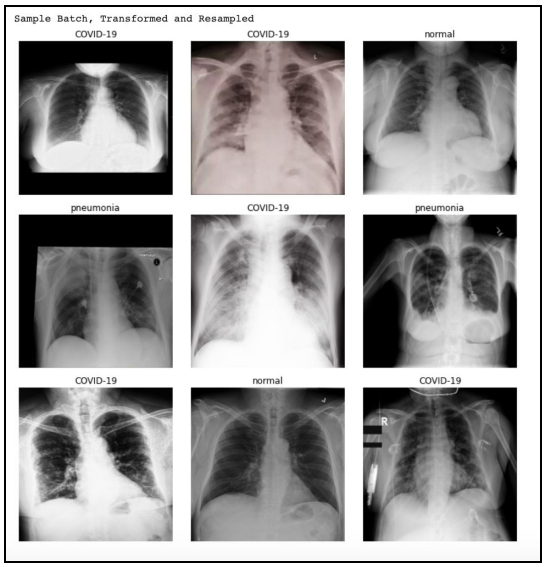
Transform	Parameters
`contrast()`	`scale=(1.5, 1.85), p=0.5`
`crop_pad()`	`size = sz`

Where **scale** is the range of the transform, **p** is the probability of applying the given transform, **sz**, here, is a user-determined value representing the image/input size (in pixels), and **do_rand** indicates whether or not the position of the crop is randomized. If **do_rand** was False, the zoom would be strictly determined by the scale and then center-cropped.

Below are sample transformed batches, before and after label resampling:



Sample Training Batch Before Resampling



Sample Training Batch After Resampling

I discuss automating image augmentation later in the implementation section.

Implementation

The original EfficientNet architecture was implemented in TensorFlow, however in order to train with a fast.ai machine learning pipeline, the project required a PyTorch implementation. The pytorchcv library provides a more robust model zoo than the offerings provided by torchvision, including implementations and weights for the EfficientNet architecture series.

For the **pytorchcv** library implementation, the EfficientNet-b1 blocks are as follows:

```
Model Blocks:
(0) EffiInitBlock: 3   layers (total: 3)
(1) Sequential  : 20  layers (total: 23)
(2) Sequential  : 39  layers (total: 62)
(3) Sequential  : 39  layers (total: 101)
(4) Sequential  : 104 layers (total: 205)
(5) Sequential  : 91  layers (total: 296)
(6) ConvBlock   : 3   layers (total: 299)
(7) AdaptiveAvgPool2d: 1   layers (total: 300)
```

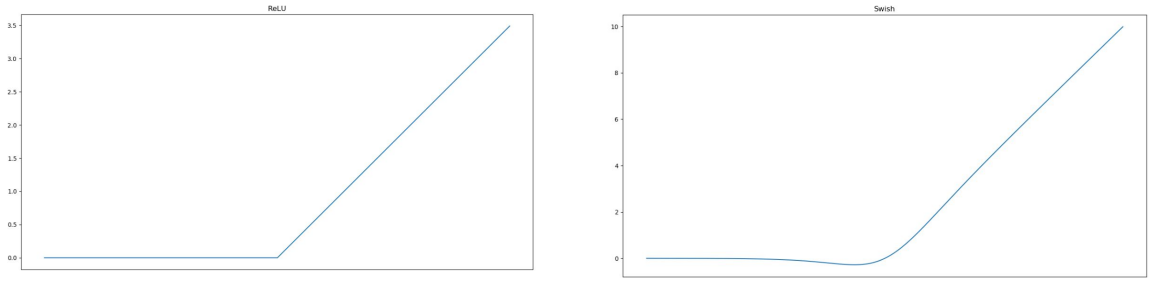
A more granular look at the layers — block by block — of this base model:

```
Model Blocks (Detailed):
---------(0)---------
(0) ConvBlock   : 3   layers (total: 3)
---------(1)---------
(0) EffiDwsConvUnit: 10  layers (total: 10)
(1) EffiDwsConvUnit: 10  layers (total: 20)
---------(2)---------
(0) EffiInvResUnit: 13  layers (total: 13)
(1) EffiInvResUnit: 13  layers (total: 26)
(2) EffiInvResUnit: 13  layers (total: 39)
---------(3)---------
(0) EffiInvResUnit: 13  layers (total: 13)
(1) EffiInvResUnit: 13  layers (total: 26)
(2) EffiInvResUnit: 13  layers (total: 39)
---------(4)---------
(0) EffiInvResUnit: 13  layers (total: 13)
(1) EffiInvResUnit: 13  layers (total: 26)
(2) EffiInvResUnit: 13  layers (total: 39)
(3) EffiInvResUnit: 13  layers (total: 52)
(4) EffiInvResUnit: 13  layers (total: 65)
(5) EffiInvResUnit: 13  layers (total: 78)
(6) EffiInvResUnit: 13  layers (total: 91)
(7) EffiInvResUnit: 13  layers (total: 104)
---------(5)---------
(0) EffiInvResUnit: 13  layers (total: 13)
(1) EffiInvResUnit: 13  layers (total: 26)
(2) EffiInvResUnit: 13  layers (total: 39)
(3) EffiInvResUnit: 13  layers (total: 52)
(4) EffiInvResUnit: 13  layers (total: 65)
(5) EffiInvResUnit: 13  layers (total: 78)
(6) EffiInvResUnit: 13  layers (total: 91)
---------(6)---------
(0) Conv2d      : 1   layers (total: 1)
(1) BatchNorm2d : 1   layers (total: 2)
(2) Swish       : 1   layers (total: 3)
```

Refinement

A key unit of EfficientNets is the Swish Activation. Swish activations function a lot like ReLU Activations (that we often find in architectures like ResNets), in that they mitigate vanishing gradients by setting lagging floating-point values at the end of the pass to 0, effectively canceling them out.

A visualization of the ReLU logic gate verses Swish logic gate:



Non-linearity applied on gradients Relu (left) and Swish (right) activation visualized.

In essence, the Swish Activation is a smoothing function and sets small gradients to 0 more gradually. I found this post really insightful on why researchers are making the switch to Swish, as benchmarks indicate it tends to perform just as well (and sometimes better) than ReLU.

It is important to note that the original EfficientNet architecture from the paper was implemented in Tensorflow. In translation, the pytorchcv library implementation of the Swish Activation function runs into memory issues between progressive resizes, due to the way the activations are hooked/stored in the forward pass. To fix this issue, I patched the Swish Activation module with an update as described in this GitHub issues thread:

```
class SwishAutograd(torch.autograd.Function):
    @staticmethod
    def forward(ctx, i):
        result = i * torch.sigmoid(i)
        ctx.save_for_backward(i)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        i = ctx.saved_variables[0]
        sigmoid_i = torch.sigmoid(i)
        return grad_output * (sigmoid_i * (1 + i * (1 - sigmoid_i)))
```

This implementation uses ~20% less memory and the author of this post runs some quick tests to confirm the memory reduction benefits between this update and the original.

Also, to run a training loop for the progressive resizing strategy, I implemented utility methods to keep code DRY at different stages of the convergence process.

For example, **get_data()** effectively wraps the Fast.ai ImageDataBunch generator — responsible for assembling the pytorch data loaders — and streamlines the process of oversampling the underrepresented class.

```
def get_data(sz, tfs, batch=8):
  """
  Automates the instantiation of DataLoaders with oversampling of
  minority class.
  Args:
    sz --> type:int --> img size.
    tfms --> type:List --> transforms.
    batch --> type:int --> batch size.
  """
  data = ImageDataBunch.from_df(path,
                                labels,
                                folder='data/train',
                                ds_tfms=tfs, fn_col=1,
                                label_col=2,
                                size=sz,
                                resize_method=ResizeMethod.SQUISH,
                                bs=bs)
  data.normalize(imagenet_stats)
  train_ds, val_ds = data.train_ds, data.valid_ds
  sampler = ImbalancedDatasetSampler(train_ds)
  train_dl = DataLoader(train_ds, bs, sampler=sampler, num_workers=4)
  val_dl = DataLoader(val_ds, 2*bs, False, num_workers=4)

  data = ImageDataBunch(train_dl=train_dl,
                        valid_dl=val_dl).normalize(imagenet_stats)
  return data

data = get_data(sz=size, tfs=tfms)
```

get_learner() effectively loads the feature extraction layers of EfficientNet as a Sequential module list into a Fast.ai Learner for the training data:

```
def efficientnet_b1(pretrained=True):
  """
  Returns efficientnet-b1 feature extraction layers, using pytorchcv's
  ptcv_get_model helper method.
  Args:
    pretrained --> type:bool (default: True)
  """
  return ptcv_get_model("efficientnet_b1", pretrained=pretrained).features

#Fastai Learner (Model with built-in Training Tools)
def get_learner(train_data):
  """
  Generates a fastai Learner with passed fastai ImageDataBunch data loaders and
  feature extraction layers from a pretrained EfficientNet-b1 network.
  Uses Half-Tensor Precision.
  Args:
    train_data --> type:ImageDataBunch
  """
  learn = cnn_learner(train_data,
                      efficientnet_b1,
                      metrics=[error_rate, accuracy],
                      callback_fns=[ShowGraph]).to_fp16()
  return learn
```

cnn_learner() is a fast.ai method that generates a *Learner Class* that manages callbacks, metrics, research-backed defaults for hyperparameters (eg., for dropout, momentum, batch normalization, loss function, etc). For more info on this method (and the fast.ai Learner more generally), you can refer to the docs.

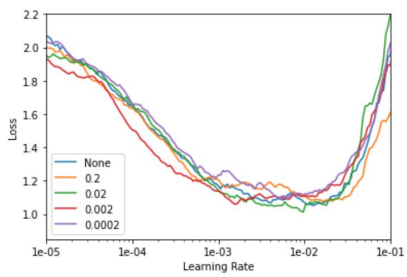
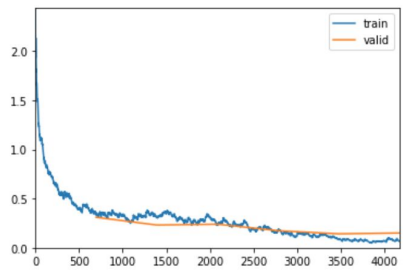
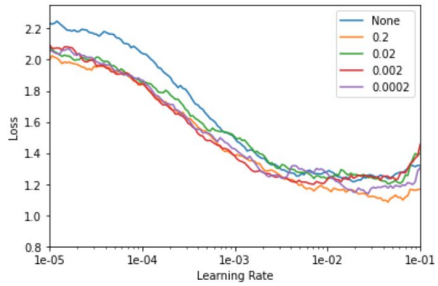
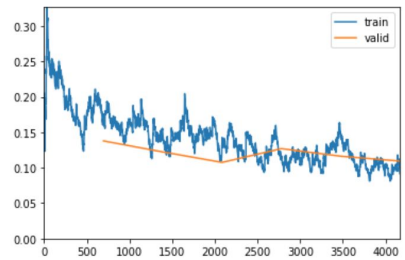
```
Hardware Specs:
Fri Jun 19 04:57:26 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.36.06    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   62C    P0    30W / 250W |     10MiB / 16280MiB |      0%      Default |
|                               |                      |                  ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

For hardware, after upgrading to Google Colab Pro, I gained access to a Tesla P100-PCIE GPU running CUDA 10:

This enabled me to use 4 workers per data loader, and increase the batch size to 16. To start, I used an input size of 450x450 and used a utility function to search for the optimal weight decay and learning rate for each phase.

The following charts visualize the convergence process by phase (where the first chart for each phase shows the hyperparameter search and the second showing the gradual decay of loss):

Stage 1 cycle len=6 lr=3e-3 wd= 2e-2 sz=450 pc_start=.40		
Stage 2 cycle len=6 lr=2e-3 wd=2e-1 sz=600 pc_start=.40		

Stage 1 (top row) retrains feature extraction layers on the COVIDx data (unfrozen) on input size 450x450, and **Stage 2 (bottom row)** fine-tunes the classifier (frozen) on a larger input size, 600x600.

Where ***pc_start*** indicates the inflection point of the learning rate decline for a given cycle in the one-cycle policy.

This approach reached 95% accuracy on validation data with only 9 one-policy-cycles:

Stage 1 Cycles; Floating Point Operations

```
#fit unfrozen, using one-cycle policy:
#appends callback to learners callback list for this run
learn.callbacks.append(TraceMallocMultiColMetric(learn))
learn.unfreeze()
learn.fit_one_cycle(6, slice(3e-3), wd=2e-2, pct_start=.4)
```

epoch	train_loss	valid_loss	error_rate	accuracy	used	max_used	peak	time
0	0.352575	0.309765	0.107991	0.892009	383185	438982	461402	14:03
1	0.307421	0.231492	0.071274	0.928726	233986	438982	306241	09:10
2	0.231832	0.239602	0.087113	0.912887	216335	438982	315788	09:32
3	0.176929	0.173757	0.060835	0.939165	235818	438982	328871	09:20
4	0.083706	0.141832	0.047516	0.952484	262749	438982	347258	09:23
5	0.069053	0.150257	0.046796	0.953204	277072	438982	353537	09:19

Stage 2 Cycles; Floating Point Operations

```
#fit frozen, using one-cycle policy:
gc.collect()
learn.callbacks.append(TraceMallocMultiColMetric(learn))
learn.freeze()
learn.fit_one_cycle(6, max_lr=slice(2e-3), wd=2e-1, pct_start=.4)
```

epoch	train_loss	valid_loss	error_rate	accuracy	used	max_used	peak	time
0	0.174029	0.137810	0.043916	0.956084	290484	367043	382314	11:22
1	0.148875	0.122252	0.043557	0.956443	216236	367043	300005	08:27
2	0.133951	0.107489	0.037437	0.962563	232491	367043	309045	08:31
3	0.113365	0.126921	0.044996	0.955004	220786	367043	322498	08:55
4	0.148248	0.116239	0.035637	0.964363	246157	367043	340826	08:37
5	0.091587	0.109376	0.036357	0.963643	262756	367043	352061	08:31

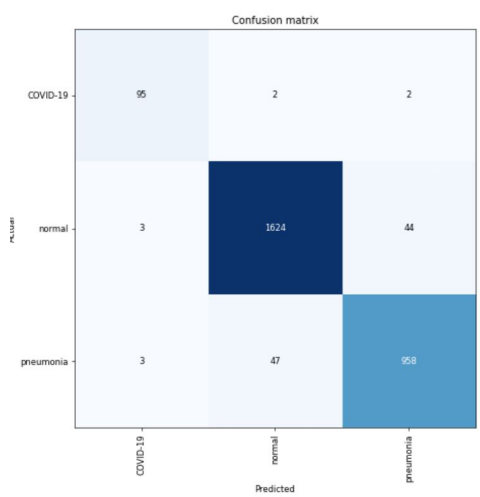
```
sample = torch.randn(1, 3, 450, 450).cuda()
print("FLOPs Count")
count_ops(learn.model, sample, )
-------------------------   ---------
Input size: (1, 3, 450, 450)
2,576,290,976 FLOPs or approx. 2.58 GFLOPs
(2576290976,
```

```
sample2 = torch.randn(1, 3, 600, 600).cuda()
count_ops(learn.model, sample2,)
-------------------------   ---------
Input size: (1, 3, 600, 600)
4,342,704,048 FLOPs or approx. 4.34 GFLOPs
(4342704048,
```

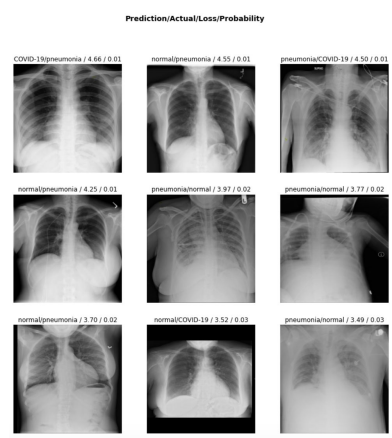
Results

Model Evaluation and Validation

The Fast.ai library provides the **ClassificationInterpetation** class to explore the recorded performance of a Learner instance; generating a confusion matrix along with samples the model struggled to label:



Confusion Matrix for Training/Validation Set



Top Losses (%) Prediction vs. Actual

From the confusion matrix, I was then able to tabulate medical diagnosis metrics to offer a more granular/micro look at training performance by label:

Train/Valid Precision (%)		
Normal	Pneumonia	COVID-19
0.970711	0.954183	0.940594

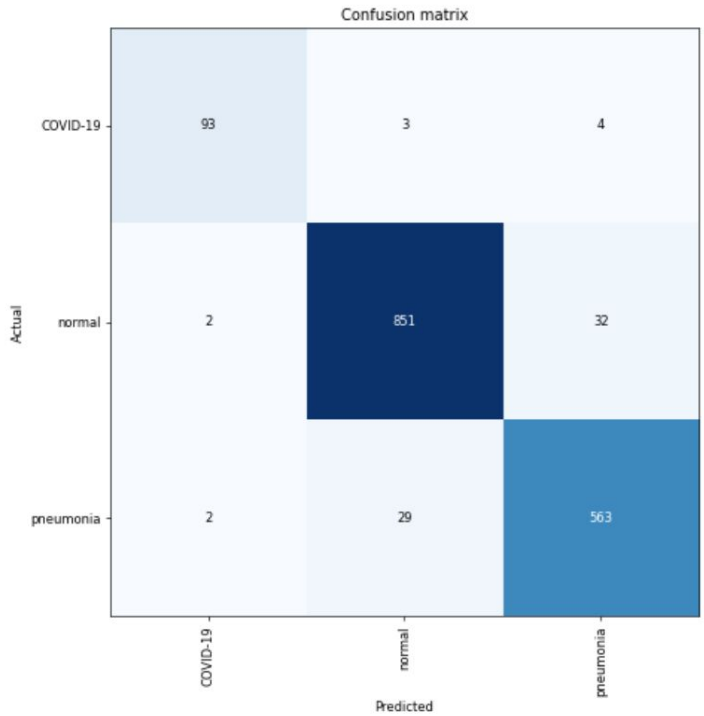
Train/Valid Sensitivity (%)		
Normal	Pneumonia	COVID-19
0.971873	0.950397	0.959596

Train/Valid F1-Score (%)		
Normal	Pneumonia	COVID-19
0.971292	0.950397	0.95

Train/Valid Specificity (%)		
Normal	Pneumonia	COVID-19
0.959268	0.975348	0.997769

Justification/Inference Test

As mentioned in an earlier section, the COVIDx dataset includes a subset of examples for testing. The following confusion matrix summarizes inference test performance:



Confusion Matrix for the Test Set

The following tables highlight that key metrics reach above 90% across the board. This reinforces findings presented in the literature of the benchmark model, that pre-screening CXRs

for viral disease perform better than many of the COVID-19 test kits on the market. More research in this area could yield a complimentary means, powered by AI, to expand testing:

Test Precision (%)		
Normal	Pneumonia	COVID-19
0.96376	0.9399	0.958763

Test Sensitivity (%)		
Normal	Pneumonia	COVID-19
0.961582	0.947811	0.93

Test F1-Score (%)		
Normal	Pneumonia	COVID-19
0.96267	0.947811	0.944162

Test Specificity (%)		
Normal	Pneumonia	COVID-19
0.957895	0.965779	0.997315

Benchmark Comparison

Moreover, I was able to demonstrate that my *FastEfficientCovid* model outperforms, on average, the benchmark model along with other popular CovNet architectures, on a number of fields. For example, fewer Parameters, fewer Floating Point Operations (FLOPs), and higher test time accuracy:

Architecture	**Params (M)**	**MACs (G)**	**Acc. (%)**
VGG-19	20.37	89.63	83.0
ResNet-50	24.97	17.75	90.6
COVID-Net	11.75	7.50	93.3
Fast-Efficient-Covid	**7.8**	**2.58-4.34**	**95.4**

Higher Sensitivity for Pneumonia and COVID-19 detection:

Sensitivity (%)			
Architecture	**Normal**	**Non-COVID19**	**COVID-19**
VGG-19	**98.0**	90.0	58.7
ResNet-50	97.0	92.0	83.0
COVID-Net	95.0	94.0	91.0
Fast-Efficient-Covid	96.2	**94.8**	**93.0**

Higher Precision for Normal and Pneumonia CXRs, while not too far off for COVID-19:

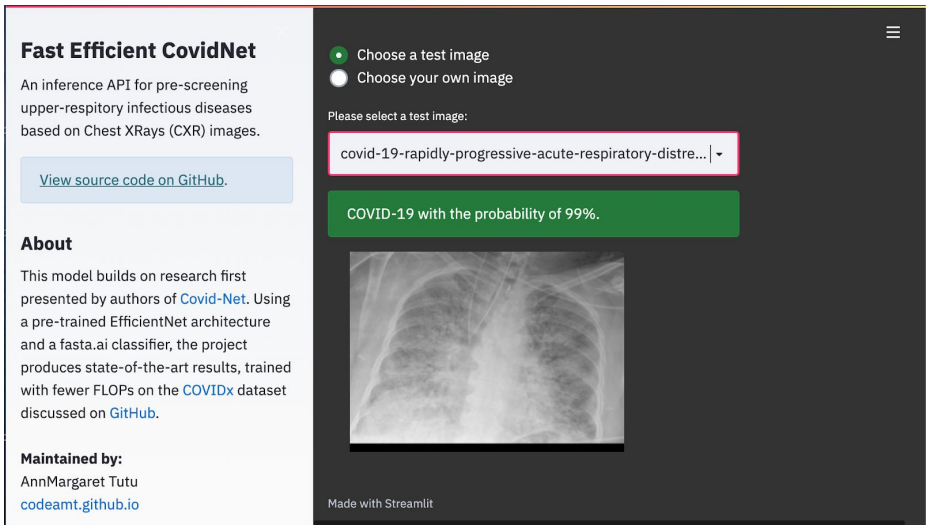
Positive Predictive Value (PPV) / Precision (%)			
Architecture	**Normal**	**Non-COVID19**	**COVID-19**
VGG-19	83.1	75.0	98.4
ResNet-50	88.2	86.8	98.8
COVID-Net	90.5	91.3	**98.9**
Fast-Efficient-Covid	**96.4**	**93.9**	95.9

This is especially significant, given the scarcity of COVID-19 training and test examples for this project.

Deployment

The final stage of my pipeline involves deploying the model as a service to demo the application in production.

I took this as an opportunity to test out a new alternative to the Flask framework gaining great traction with the data science community, called Steamlit. Steamlit is a Python developer tool for building minimal, responsive data apps. It uses a markdown template generator to serve pages and injected Python objects, with Bootstrap-like components on the frontend. I served the static test image files locally, with a second option for pasting a url. On a MacBook Pro, it takes ~4.5 seconds to run CPU inference on one of the examples from a drop down:



EfficientNet serialized model files are much lighter (MB file size, instead of GB), and would be much easier to deploy on Internet of Things (IoT) and/or mobile devices. The code repository also includes a Dockerfile and a requirements.txt file to launch the app as a container/virtual environment or with a cloud provider, as well as a disclaimer that this is for demonstration purposes, and **should not** be used in place of a medical consultation. Authentication flow is lacking with Streamlit (so, for data security reasons, FastAPI might be a better solution for production.

I also created command-line tools for generating the COVIDx dataset and modeling, and a quick-preview notebook that leverages ngrok to launch the app from an iPython Notebook. Last, I deployed the model on a t-micro EC2 instance via Amazon Web Service to provide a more public facing demo.

Conclusion

Reflection

I thoroughly enjoyed working on this project to culminate my learning experience with Udacity and must acknowledge the incredible work from the benchmark model, along with the Fast.ai library, deeplearning.ai's AI for Medicine course, and the breadth of documentation available on the Streamlit framework.

Improvement

Future iterations might include:

- Object-Detection
- Image retouching, using a UNet
- Experimenting with the new medical module in fastai2
- Integrating a cloud storage provider for uploading, saving, sharing capabilities
- Leveraging top-k similarity to build further label consensus
- Orchestration with other containerized services (e.g., for contact tracing) to flesh out a more robust public health tool

References

[1] D. Willman, “Contamination at CDC lab delayed rollout of coronavirus tests.” Apr. 18, 2020. [Online] Available: https://www.washingtonpost.com/investigations/contamination-at-cdc-lab-delayed-rollout-of-coronavirus-tests/2020/04/18/fd7d3824-7139-11ea-aa80-c2470c6b2034_story.html

[2] E. Barriball et al., “Supply-chain recovery in coronavirus times—plan for now and the future.” Mar. 2020. [Online]. Available: https://www.mckinsey.com/business-functions/operations/our-insights/supply-chain-recovery-in-coronavirus-times-plan-for-now-and-the-future

[3] K. Green et al., “What tests could potentially be used for the screening, diagnosis and monitoring of COVID-19 and what are their advantages and disadvantages?” CEBM. Apr. 20, 2020 [Online]. Available: https://www.cebm.net/wp-content/uploads/2020/04/CurrentCOVIDTests_descriptions-FINAL.pdf

[4] L. Wang and A. Wong, “COVID-Net: A Tailored Deep Convolutional Neural Network Design for Detection of COVID19 Cases from Chest Radiography Images,” ArXiv200309871 Cs Eess, Mar. 2020 [Online]. Available: http://arxiv.org/abs/2003.09871.

[5] L. N. Smith, “Cyclical Learning Rates for Training Neural Networks,” ArXiv150601186 Cs, Apr. 2017. [Online]. Available: http://arxiv.org/abs/1506.01186.

[6] M. Farooq, A. Hafeez, “COVID-ResNet: A Deep Learning Framework for Screening of COVID19 from Radiograph,” arXiv:2003.14395, 2020. Available: https://arxiv.org/abs/2003.14395

[7] M. Tan and Q. Lee, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” arXiv:1905.11946, Nov. 23, 2019. Available: https://arxiv.org/pdf/1905.11946.pdf

[8] O. Gozes et al., “Rapid AI Development Cycle for the Coronavirus (COVID-19) Pandemic: Initial Results for Automated Detection & Patient Monitoring using Deep Learning CT Image Analysis,” ArXiv200305037 Cs, Mar. 2020. [Online]. Available: http://arxiv.org/abs/2003.05037.

[9] J. Howard and S. Gugger, “fastai: A Layered API for Deep Learning,” Information, vol. 11, no. 2, p. 108, Feb. 2020, doi: 10.3390/info11020108.