Breast Cancer Diagnosis Using Deep Transfer Learning

Abhishek Bais - Wasae Qureshi - Subarna Chowdhury Soma
Team: AWS

Abstract

According to the American Cancer Society, after skin cancer, breast cancer is the most commonly found cancer that affects American women. So much so, 13% of the American women population run the risk of getting it in their lifetime. In 2021, it is estimated that 43,600 women will die from breast cancer. Breast cancer is of many types. Amongst them, Invasive Ductal Carcinoma (IDC) is the most common type of breast cancer. To determine whether the IDC is benign or malignant, pathologists focus on small regions of breast tissue which contain the IDC.

With rapid advancements in Artificial Intelligence (AI) and Deep Learning (DL), particularly in the field of medical imaging, it is now possible to create "**Automated Disease Detection Systems".** These systems provide necessary tools to assist doctors in making reliable, and accurate diagnosis of cancer. Consistent results delivered in quick time goes a long way in saving lives.

In the last few years, Convolutional Neural Network (CNN), has emerged as the goto algorithm in the area of computer vision and image processing. They extract features from images automatically. Detecting breast cancer features in images accurately, however, requires the ability to extract a very large number of image features. Training from scratch can be a bottleneck as a large corpus of labelled images may not be easily available. It can also turn out to be extremely expensive, both in terms of compute resources and time spent in training the system. It is here that "**Deep Transfer Learning**" offers incredible promise. Here, knowledge acquired by prior, elaborate training on similar domains is leveraged to help solve new problems.

In this study we compare and contrast a wide array of ensembled models with pre-trained CNNs and multiple Deep Neural Network (DNN) layers to find a model with highest accuracy of predicting malignant cancer given an image of the breast tissue. Additionally, we also ensemble a model with CNN and Long short-term memory (**LSTM**), which is a

kind of artificial recurrent neural network (RNN) and compared and contrasted its results with the CNN and DNN ensemble. We performed this study on a subset of the Breast Histopathology Images dataset, details of which are described in the **Data** section of this report.

After performing a series of hyper-tuning experiments and evaluating on a range of metrics, such as test accuracy, size of the model, precision, recall, and AUC/ROC, we found that the best model for Breast Cancer detection is an ensemble of **VGG16, a pre trained CNN model, that is 16 layers deep and trained on more than a million images from the ImageNet database** and multiple DNN layers. The details of these experiments are described in the **Experiments** section of this report. The best model had a training accuracy of 84.8%, a validation accuracy of 82.6% with a precision of 85%, a recall of 85% and a validation AUC of 90%. The system achieved this accuracy after 20 epochs of training which we believe can be further improved with more compute resources at our disposal.

Introduction

In the USA and the world, Breast Cancer continues to be one of the leading causes of death among women. The American Cancer Society estimates that about 281,550 new cases of invasive breast cancer will be diagnosed in American women in 2021 alone.

Scientists have found many contributing factors that induce uncontrolled cell mutation causing breast cancer. Some of these factors are obesity, inhaling birth control pills, irregular menstrual cycles, exposure to radiation and estrogen hormone imbalance. Early symptoms include soreness in the breast, skin irritation, redness, swelling of tissue, erosion of nipples and watery discharge. Diagnosing cancerous manifestation early and accurately is imperative to saving lives.

Traditional methods of detecting breast cancer early include mammographic detection techniques based on computer-aided detection (CAD) methods. These tools rely on manually extracted features, a difficult task, requires experts on the job, and are extremely expensive.

The advent of **Convolutional Neural Networks (CNNs) and transfer learning techniques** solve some of these problems. Here, knowledge gained by training models on millions of images of related domains, on TPUs and GPUs is leveraged to solve new problems.

Today, CNNs with transfer learning are used in two different ways.

1. Pre-trained CNNs are frozen and used to extract features.
2. Pre-trained CNNs have some layers unfrozen and fine-tuned on new images while the rest of the layers are kept frozen.

In this study we compare and contrast both transfer learning techniques with CNNs.

An ensemble model with pre-trained CNN with Deep Neural Network (DNN) layers is then built and used to classify the medical images, here identifying whether the image of the breast tissue corresponds to a benign or malignant cancerous patient. **The broad architecture of an ensemble model with pre-trained CNN with DNN layers is shown in Fig 1.0. below.**

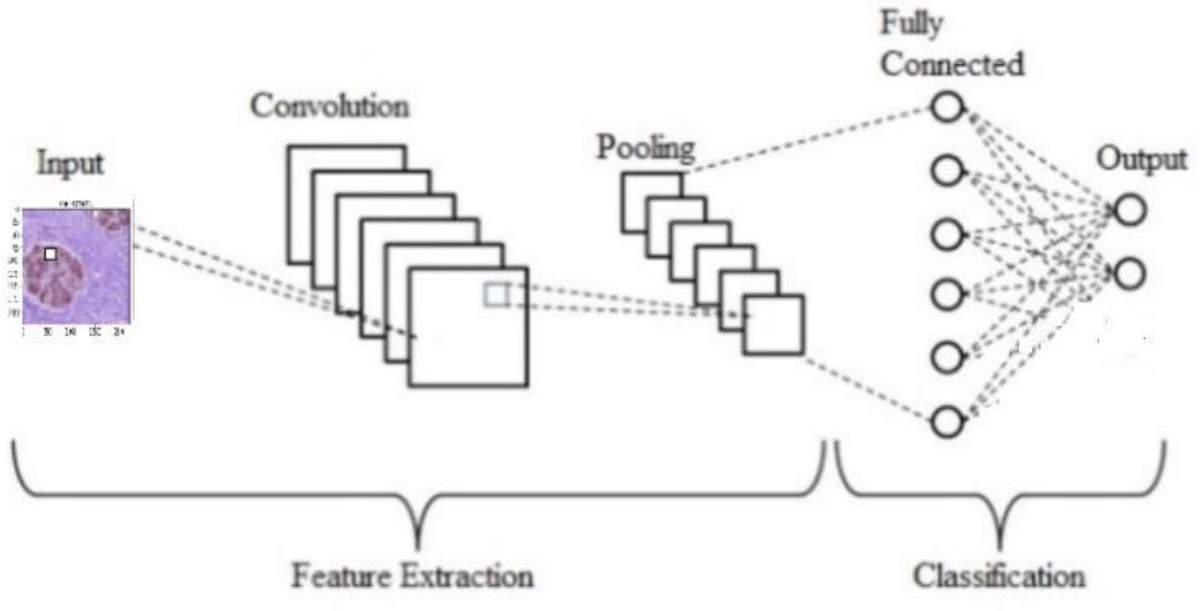


Fig 1.0. Convolutional Neural Network (CNN) with Deep Neural Network (DNN) layers

Such an architecture offers a significant advantage over traditional mammographic detection techniques due to their ability to automatically apply millions of extracted image features from related domains and apply them to solve new problems. CNNs in particular due to their weight sharing and local connectivity characteristics are uniquely positioned to act as feature extractors. Combining low-level features with higher level features, they are able to obtain subtle details hidden in images, helping classify medical

images accurately, distinguishing between benign and malignant breast cancer manifestations.

In our study we compared and contrasted a wide array of pre-trained CNN models. These include **DenseNet201, InceptionResNetV2, NASNetMobile, ResNet50V2, ResNet152V2 and VGG16.** Additionally, we hyper-parameter tuned the best model, evaluated on a range of metrics further and trained it further.

Another ensemble model we evaluated was a **hybrid model of pre-trained CNN with Long Short Term Memory (LSTMS). The broad architecture of an ensemble hybrid model with pre-trained CNN with LSTM layers is shown in Fig 2.0. below.**

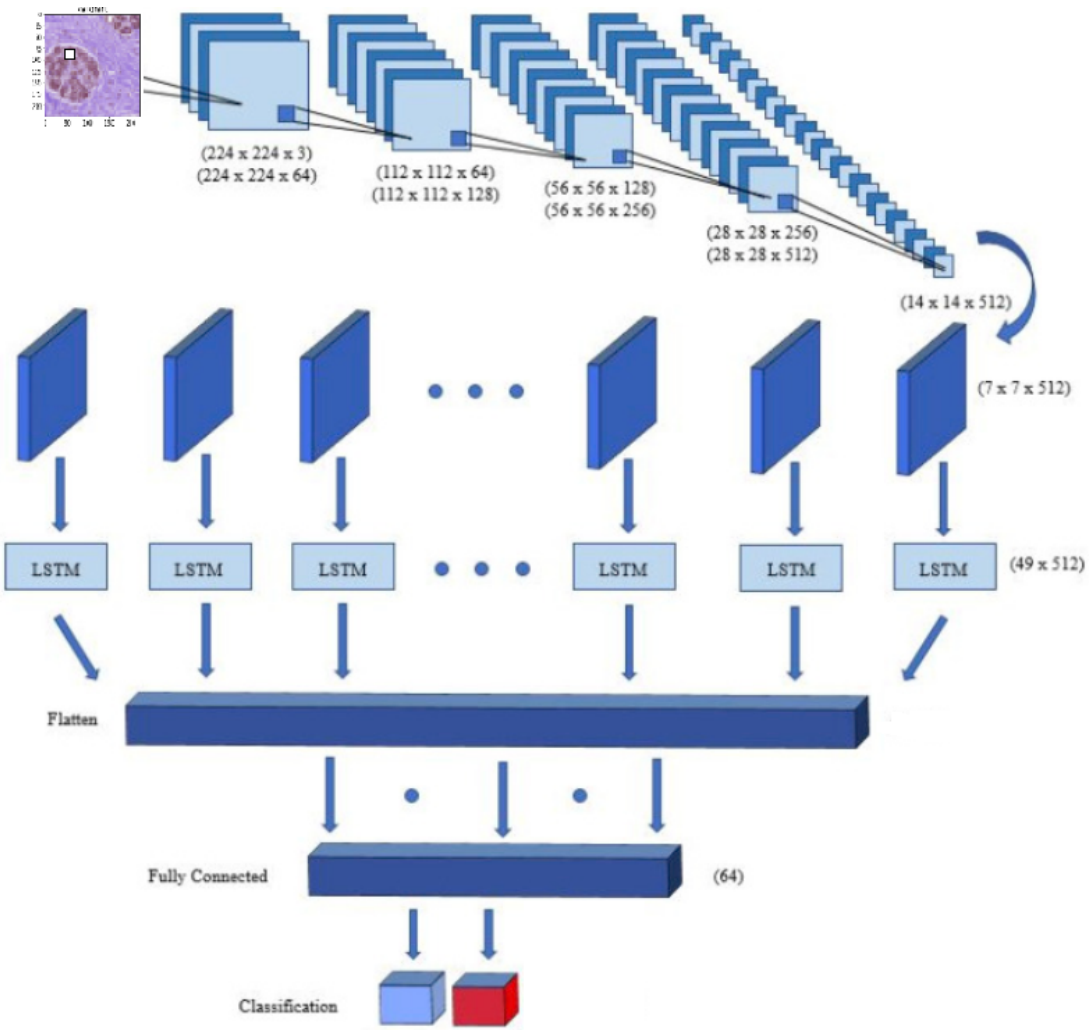


Fig 2.0. Convolutional Neural Network (CNN) with Long Short Term Memory (LSTM) layers

The experiment section of this report presents details of experiments performed on these models to get the best model for breast cancer image classification on the Breast histopathological images dataset.

Related Work

In recent years, several studies have attempted to apply different hybrid deep networks, particularly transfer learning approaches, to image classification and cancer detection. However, only a few review articles concerning transfer learning on medical image analysis have been published.

The paper [1] describes how a rudimentary Convolutional Neural Network (CNN) with Long Short-Term Memory (LSTM) is integrated, leading to a new paradigm in the study of image classification. According to the authors, when LSTMs are used in a layered order, they should supplement CNN's functions of extracting features. Moreover, LSTMs also have the capability of selectively remembering patterns over a long period, and CNN can extract important features from them. Particularly, this study shows that LSTM-CNN layered structure produces a more robust model suitable for a wider range of classification tasks than conventional CNNs.

The efficiency of transfer learning in breast cancer detection techniques has been studied in paper [2]. The paper states that training with Convolutional Neural Network requires a large volume of labeled images, which is not readily available for mammographic tumor images. Three approaches are used here for the research, one is training a CNN from scratch and the other two is the combination of Transfer learning approaches using the VGG-16 model. These experiments show that the pre-trained VGG-16 model is more accurate for breast cancer detection models when using these features to train Neural Networks (NN)-classifiers. The authors in paper [3] represent a comparative study using three pre-trained networks: VGG16, VGG19, and ResNet50 and their performance in breast cancer classification. Using a fine-tuned pre-trained VGG16 and logistic regression classifier, the best performance with 92.60% accuracy was achieved.

Throughout the literature studies, we have learned that applying transfer learning approaches to detecting breast cancer and training an NN-classifier by feature extraction is a faster method and efficient approach. Hence, we have developed and implemented a breast cancer identification approach using transfer learning and LSTM-CNN layered structure implemented in paper [1].

Data

The original dataset for this study was prepared for the research titled "**Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases**", undertaken by Janowczyk A and Madabhushi A, both professors of Biomedical Engineering at Case Western Reserve University in Cleveland, OH, USA, in 2016.

The original dataset can be found here.

1. 162 whole mount slide images of Breast Cancer specimens were scanned at 40x resolution.
2. 277,524 patches of size 50 x 50 (198,738 IDC negative and 78,786 IDC positive) were extracted.
3. Each patch was given a file name in the format: uxXyYclassC.png — > example 10253idx5x1351y1101class0.png . Where u is the patient ID (10253idx5), X is the x-coordinate of where this patch was cropped from, Y is the y-coordinate of where this patch was cropped from, and C indicates the class where 0 is non-IDC and 1 is IDC.

For our study, we took a sub-sample of these images. **Table 1.0. below provides a partitioned description of our sub-sample.**

Data	Benign + Malignant (Total)
Training	3000
Validation	1000
Testing	1000

Table 1.0.

Some images from the dataset are captured in Fig 3.0, Fig 4.0 and Fig 5.0 below.

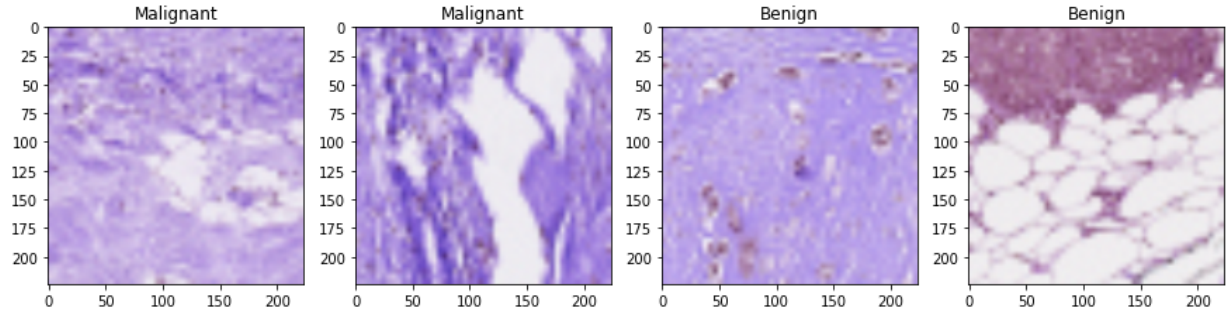


Fig 3.0. Images from the training set

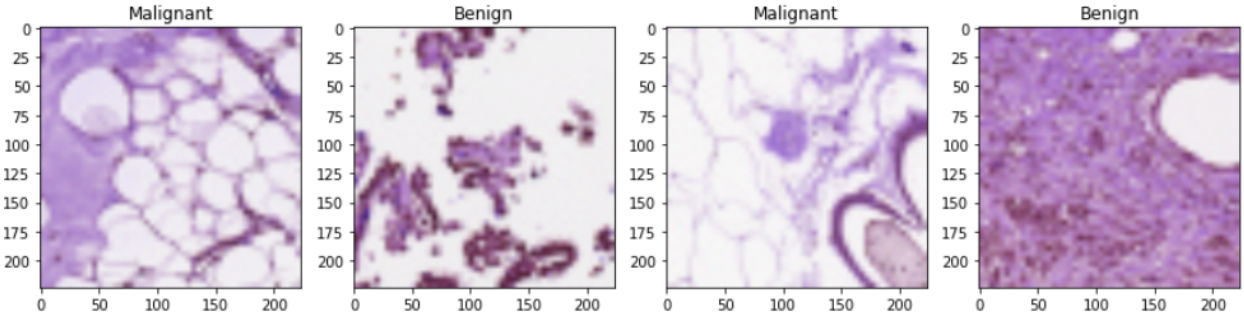


Fig 4.0. Images from the validation set

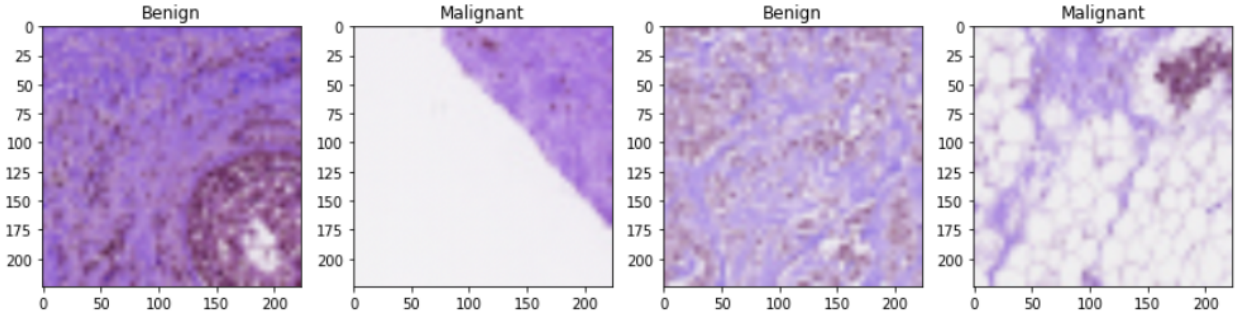


Fig 5.0. Images from the test set

Method

A typical **"Machine Learning"** process involving seven stages namely **Data Collection -> Data Pre-processing/ Sampling -> Model Development -> Training -> Evaluation Hyperparameter Tuning -> Prediction** was adhered to in this study. The best model was deployed on the Google Cloud AI Platform. Our flask application on EC2 would

make REST API requests to the model on Ai Platform to get a prediction. **The high level sub-tasks at key steps of this process are captured in Fig 6.0 below.**

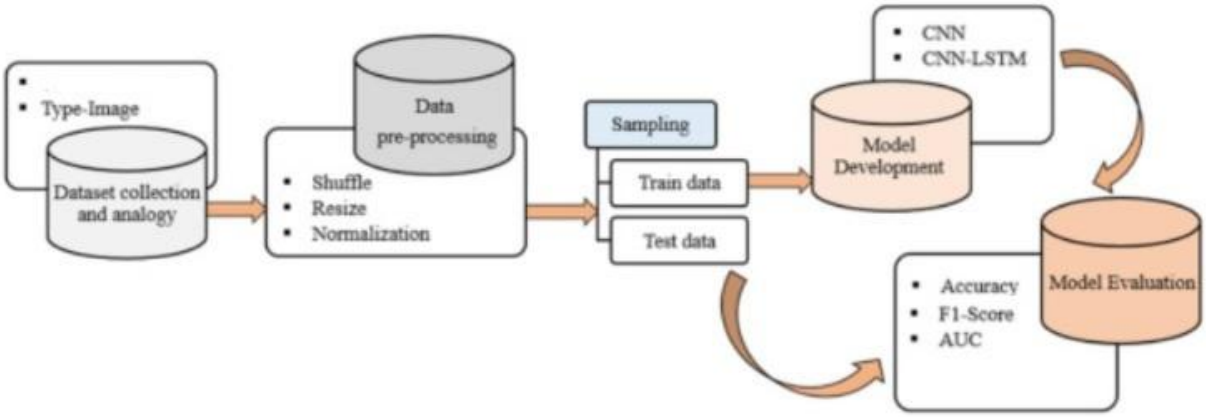


Fig 6.0. Machine learning tasks done to classify an image benign or malignant

TensorFlow Extended (**TFX**), an end-to-end platform for deploying production ML pipelines in conjunction with **GitHub and KubeFlow**, a free and open-source platform designed to orchestrate complicated workflows running on Kubernetes was used to achieve **Continuous Development, Continuous Integration, Continuous Training**.

In addition, **Google's Artificial Intelligence Platform** was integrated with **KubeFlow** to **version both the training pipeline and the trained models**. **The system architecture of the system is shown in Fig 7.0 below.**

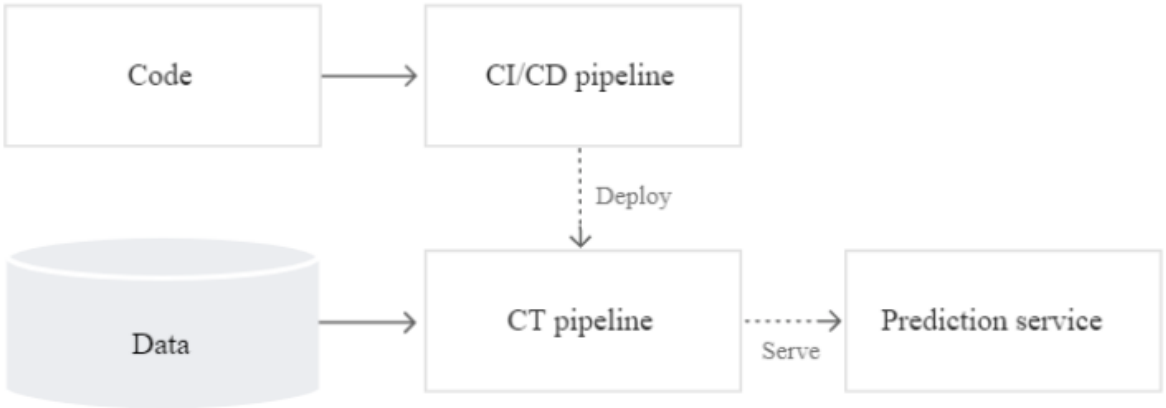


Fig 7.0. System Architecture

The different machine learning steps are elaborated below:

Data Collection

1. The original **Case Western University dataset** found here has since been amalgamated. This **amalgamated dataset** can be found here.

2. It has a total of 397476 IDC negative images representing Benign Breast Cancer and a total of 157572 IDC positive images representing Malignant Breast Cancer.

Data Pre-processing/ Sampling

1. The class distribution of the image samples is shown in Fig 8.0. below

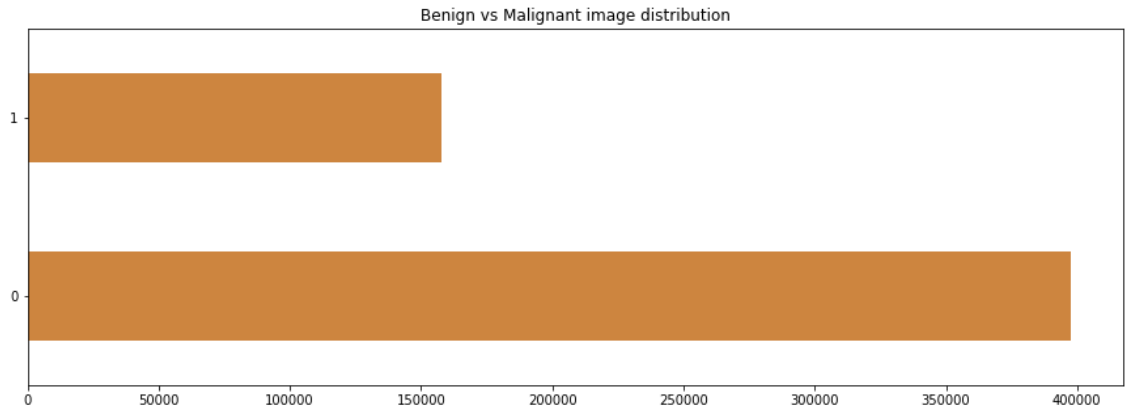


Fig 8.0. Class distribution of input images

2. Given the high imbalance in the class distribution and the available compute resources, we randomly sampled benign and malignant and down-sampled to 2500 images each to balance the data. The class distribution of the images post class balancing is shown in Fig 9.0 below.

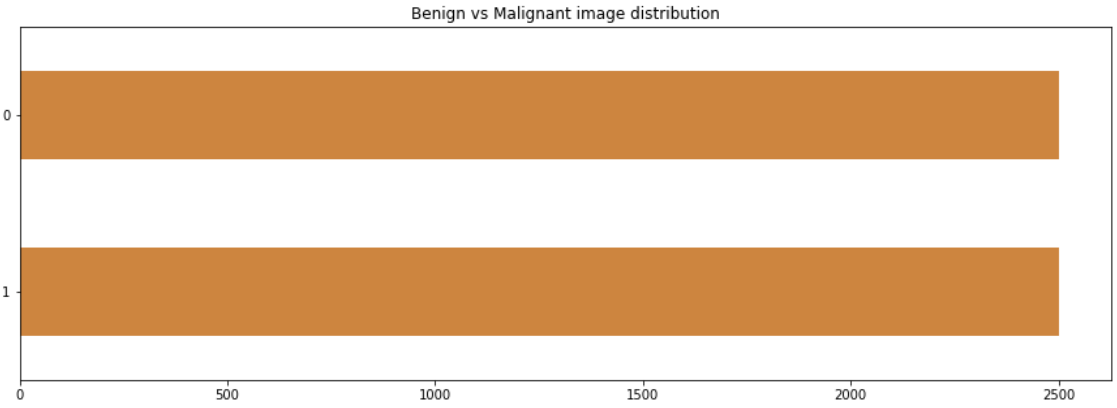


Fig 9.0. Class distribution post balancing

3. All images were normalized and reshaped to 224 x 224 x 3 to leverage the pre-trained CNN image models.
4. These 5000 normalized, reshaped images were subsequently used to train/validate/test the ensembled models with CNN + DNN layers.

Model Development (CNN + DNN layers)

1. A custom model ensembling pre-trained CNN + DNN layers was constructed.
2. The following **pre-trained CNN models** were compared and contrasted
    - DenseNet201
    - InceptionResNetV2
    - NASNetMobile
    - ResNet50V2
    - ResNet152V2 and
    - VGG16
3. The **DNN layers included** a Dense (512) layer, a Dense (128) layer, two on-demand Dropout layers, an on-demand Batch Normalization layer and a Dense (2) layer with softmax activation to make classifications.

Model Development (CNN + LSTM)

1. A custom model ensembling pre-trained CNN + LSTM was constructed.
2. The **pre-trained CNN model** chosen was VGG16.
3. **Other DNN layers included** a GlobalAveragePooling2D layer, an Embedding layer, a LSTM layer, a Dense (512) and a Dense (128) layer, two on-demand Dropout layers, an on-demand Batch Normalization layer and a Dense (2) layer with softmax activation to make classifications.

Training

1. **First, a 60/20/20 train/validate/test split** of images was done.
2. **Next, Andreas Muller, (Phd, Principal Software Engineer, Microsoft) setup a loop** to construct different ensembled CNN + DNN layer models (one each for the six pre-trained CNNs), trained and results evaluated over 10 epochs and a batch_size of 32 using rmsProp optimizer.
3. **Then, training was done** with the following baseline hyper-parameters
    - Pre-trained CNN kept frozen and used only to extract features.
    - No image augmentation.
    - No batch normalization layer
    - Binary Cross Entropy Loss

- Adaptive learning rate with a patience of 20, factor 0.2, minimum learning rate of 1e-04 and early stop for regularization.
- Dropout with a rate of 0.2.

4. **Finally, results compared and contrasted** over various evaluation metrics namely
    - Training/ Validation Accuracy
    - F1 score
    - Recall score
    - AUC/ROC
    - Model size vs Accuracy

5. **Subsequently, the best ensemble model was hyper-parameter tuned**, over 20 epochs to further improve its performance over the evaluation metrics.
6. **Additionally, a CNN + LSTM ensemble model was also constructed** using the hyper-parameters for the best CNN + DNN layer model as reference.This model was also trained over 20 epochs and results compared over the same evaluation metrics.

Evaluation

1. The results of the Muller loop described under Training 2.0 with different pre-trained CNN models over various metrics are captured in Experiments, and Results sub-section 1. of this report.
2. **VGG16, a pre-trained CNN model that is 16 layers deep and trained on more than a million images from the ImageNet database** with multiple DNN layers was evaluated as the best model. The results are captured in Experiments, and Results sub-section 2. of this report.
3. **VGG16, with LSTM** with Image Augmentation, increased lstm units to 512, no Dropout layer out, no Batch Normalization, and no fine tuning of pre-trained CNN model layers produced the best results. However it was no match before the CNN + DNN layer model. The results are captured in Experiments, and Results sub-section 3. of this report.

Hyperparameter Tuning

1. **Six different combinations of hyper-parameters were tuned** to improve model accuracy, measured via the evaluation metrics for the best ensemble model identified by the Muller loop.
2. **The first experiment** was to increase the number of epochs of the best model to 20.
3. **The second experiment** was done to add a BatchNormalization layer based on findings of the paper titled "ImageNet pre-trained models with batch normalization" from Harvard University, that found BatchNormalization improves model accuracy of pre-trained CNN models.

4. **The third experiment** was augment images. Here both training and validation images were augmented. A range of hyper-parameters were tried. The best improvements were observed with the underneath settings.
    - rotation_range=20
    - zoom_range = [1.0,1.2]
    - horizontal_flip=**True**
    - vertical_flip=**True**

5. **The fourth experiment** was to change the Dropout to 0.5. This was done based on prior research titled "Recurrent Dropout without Memory Loss" by Stanislau Semeniuta et. all and Google Research that suggested a dropout of 0.5 is better suited for later layers of the DNN.
6. **The fifth experiment** was to try the "**adam**" optimizer inplace of the "**rmsProp"** optimizer.
7. **The sixth and final experiment** was to unfreeze the last 4 layers of the VGG16 model, allowing them to extract features from the breast cancer image set as well.
8. The best model obtained was the ensemble of VGG16 + DNN, after unfreezing the last 4 layers of VGG166 pre-trained model.
9. **All results were also captured on the tensorboard.**

NOTE: Each experiment was done incrementally on top of the previous one.

Prediction

1. The best model was deployed and served using a tensorflow serving REST API on the local host and prediction made on a test image.
2. Additionally, the best model was also saved for deployment testing on EC2, flask.

With the machine learning experimental process complete, the team was ready to move from research to production. We used TensorFlow Extended (TFX) framework to create and manage pipelines. Together with GitHub, KubeFlows, Google AI platform Continuous Development (CD), Continuous Integration (CI) and Continuous Training (CT) was achieved. These are detailed in section End to End Pipeline and beyond.

Experiments & Results

1. **Andreas Muller (Phd, Principal Software Engineer, Microsoft) training results** over 10 epochs with ensemble models constructed with a pre-trained CNN model, DNN layers and baseline hyper-parameter settings as described in the **Training section 3.** are evaluated in **Fig 10 - Fig 17** below.

2. The best model evaluated over a range of metrics will then be further hyper-parameter tuned using the settings as described in section **Hyper Parameter Tuning 1.0 - 7.0.**

2.1. Evaluation of models by accuracy of prediction

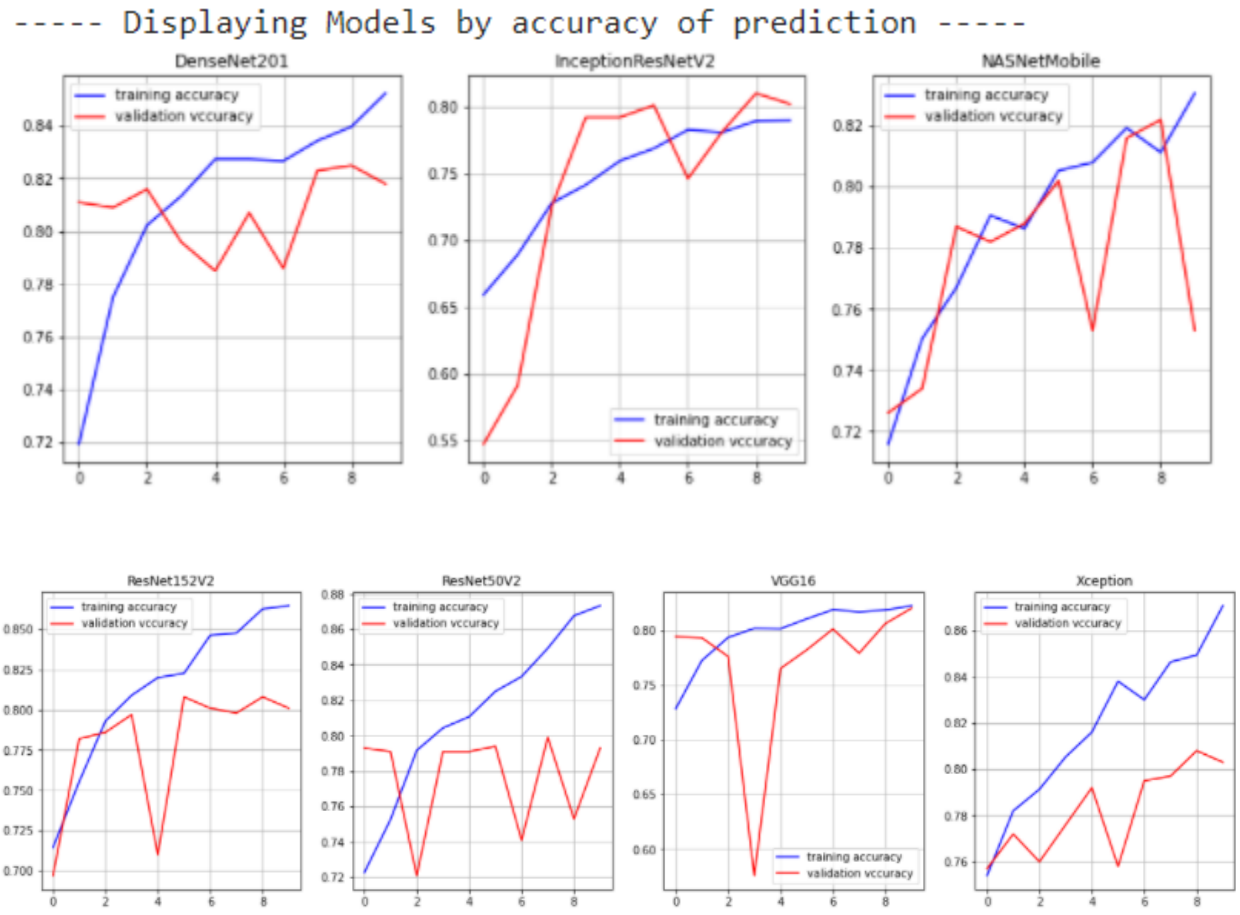


Fig 10. Different pre-trained models compared by accuracy of prediction

2.2. Evaluation of models by size

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
2	NASNetMobile	4269716	0.830667	0.753
5	VGG16	14714688	0.822333	0.820
0	DenseNet201	18321984	0.852333	0.818
6	Xception	20861480	0.870667	0.803
4	ResNet50V2	23564800	0.873333	0.793
1	InceptionResNetV2	54336736	0.789667	0.802
3	ResNet152V2	58331648	0.864667	0.801

Fig 11. Different pre-trained models compared by size

2.3. Evaluation of models by accuracy vs size

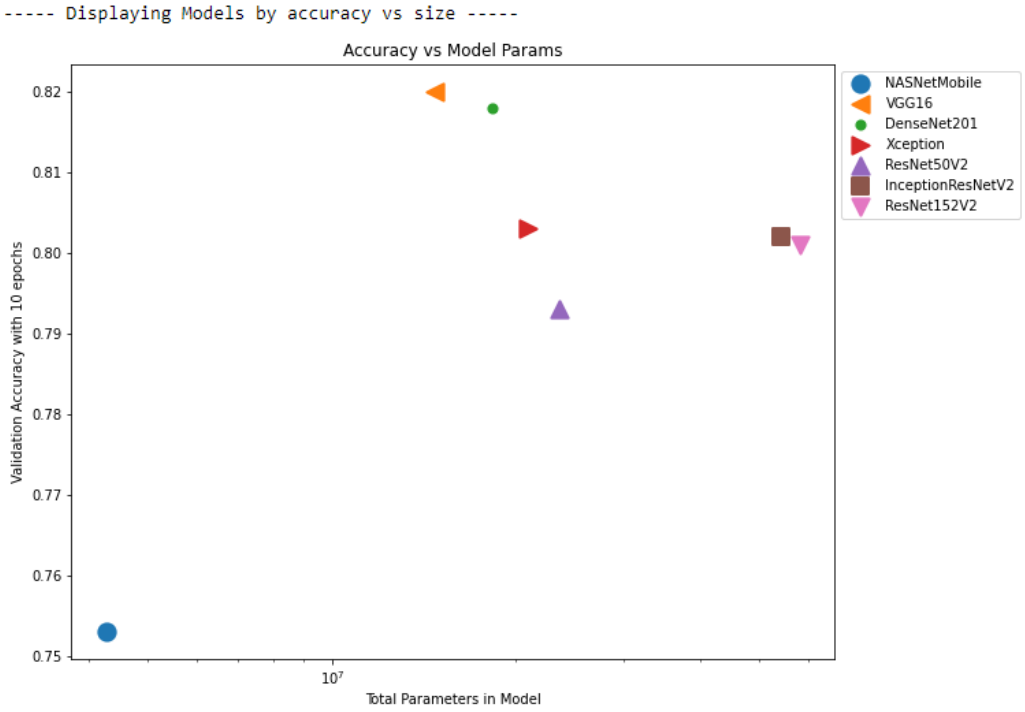


Fig 12. Different pre-trained models compared by accuracy vs size

2.4. Evaluation of models by precision

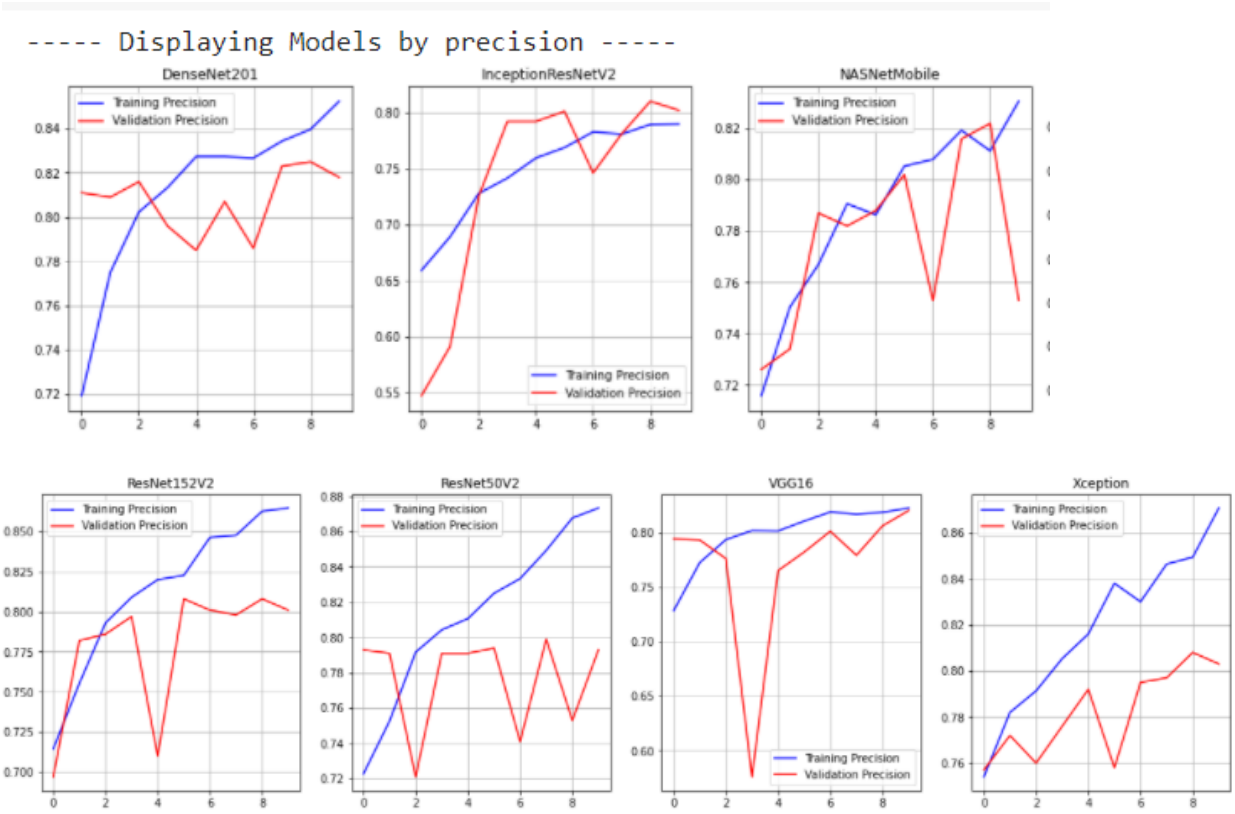
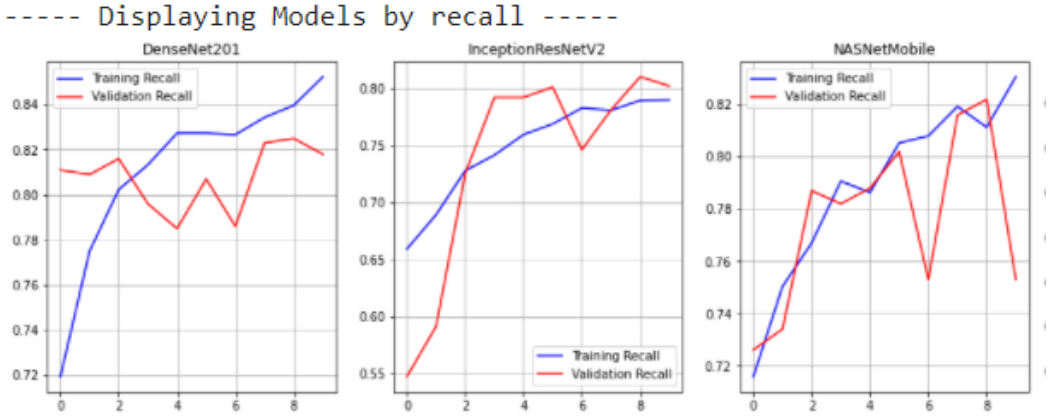


Fig 13. Different pre-trained models compared by precision

2.5. Evaluation of models by recall



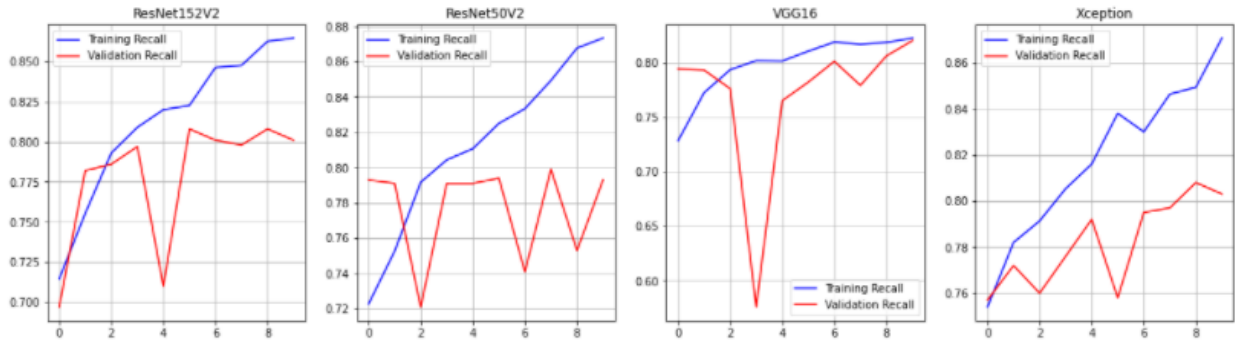


Fig 14. Different pre-trained models compared by recall

2.6. Evaluation of models by loss

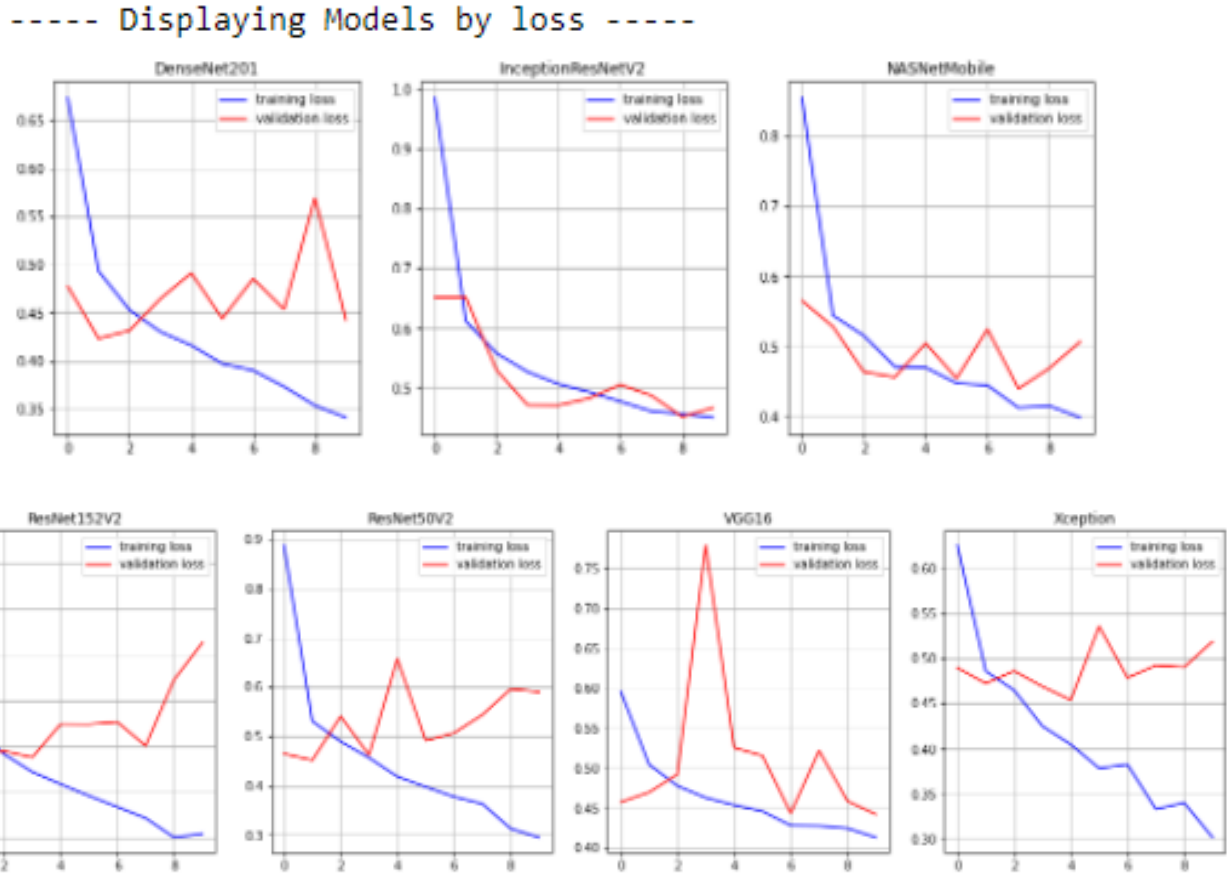


Fig 15. Different pre-trained models compared by loss

2.7. Evaluation of models by AUC

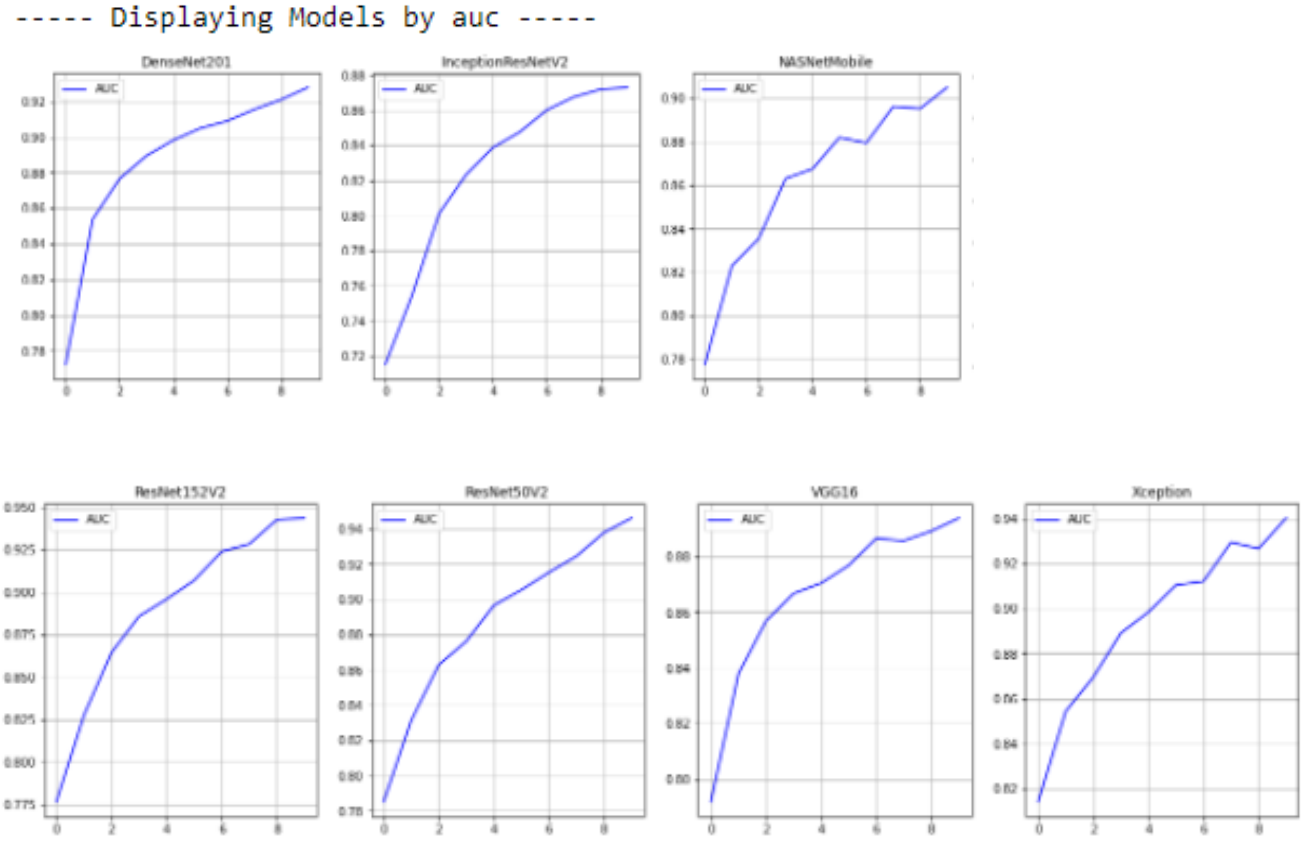


Fig 16. Different pre-trained models compared by AUC

3. After evaluating the different ensemble models, **we picked the ensemble with VGG16 pre-trained CNN model as the best model** for it had the highest validation accuracy of 82%. It had a high 82.3% training accuracy, increasing training and validation accuracy after 10 epochs showing promise of improvement. With increased epochs, a narrow gap between training and test accuracy showed the model did not overfit the training data, high precision, recall, AUC numbers and was not too big in size. Big models require compute resources with higher RAM, while we were pressed with compute resources with a maximum RAM of 25GB.

4. **The NASNetMobile had a higher training accuracy of 83% and a much smaller size** (~10 million fewer training parameters) than VGG16 but a higher

7% gap between the training and test accuracy, compared to a statistically identical ~82% training and test accuracy for VGG16.

5. **The RESTNet50V2 had the highest training accuracy of 87.3%** amongst all pre-trained CNN models, but a lower validation accuracy of 79.3% compared to VGG16.

6. **Next, hyper-parameter tuning was performed on the ensembled model with VGG16 pre-trained CNN model** as described in section on Hyper-Parameter Tuning 1-through 7. The results of these experiments are captured below.

6.1 **The experimental results for the first experiment** was to increase the number of epochs from 10 to 20 are captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.845667	0.8

Fig 17. Ensembled model with VGG16 with num_epochs = 20

6.2. **The experimental result for the second experiment** to add a BatchNormalization layer on top of 6.1 is captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.839333	0.733

Fig 18. Ensembled model with VGG16 and BatchNormalization layer

6.3. **The experimental result for the third experiment to add augmented training and test images** on top of 6.2 is captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.814	0.793

Fig 19. Ensembled model with VGG16 and image augmentation

6.4 **The experimental result for the fourth experiment** to change the Dropout to 0.5 on top of 6.3 is captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.811667	0.798

Fig 20. Ensembled model with VGG16 with dropout 0.5

6.5. **The experimental result for the fifth experiment** to try the "**adam**" optimizer inplace of the "**rmsProp"**on top of 6.4 is captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.804	0.789

Fig 21. Ensembled model with VGG16 with adam optimizer

6.6. **The experimental result for the sixth and final experiment** to unfreeze the last 4 layers of the VGG16 model, allowing them to extract features from the breast cancer image on top of 6.5 is captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.848667	0.826

Fig 22. Ensembled model with VGG16 with last 4 layers unfrozen

7. **The CNN with LSTM ensembled model was also evaluated** with VGG16 as its pre-trained CNN model, first with baseline hyper-parameter settings described in section on **Model Development (CNN with LSTM)**. It had a low training accuracy of 63.6% and validation accuracy of 65%.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.636	0.65

Fig 23. CNN with LSTM model baseline hyper-parameter settings

8. **The CNN with LSTM ensembled model was also hyper-parameter tuned** by increasing lstm units to 512, turning off dropout, turning off batch normalization and turning off fine tuning of the pre-trained CNN model. This slightly improved the training and validation accuracy vs baseline results.These results are captured below.

```
----- Displaying Models by size=(model params) -----
```

	model_name	model_params	training_accuracy	val_accuracy
0	VGG16	14714688	0.657333	0.653

Fig 24. CNN with LSTM model post hyper-parameter tuning

9. **With the best ensemble model identified in step 6.6**, the model described below was evaluated on various other metrics such as Confusion matrix, precision, recall and support. These results are captured below.

9.1. **Best model summary**

```
----- Displaying Best Model summary -----
Best model for breast cancer image classification is VGG16 with validation accuracy of 82.60 percent.
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 512)               14714688
_________________________________________________________________
dense (Dense)                (None, 512)               262656
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 128)               512
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 258
=================================================================
Total params: 15,043,778
Trainable params: 7,408,258
Non-trainable params: 7,635,520
_________________________________________________________________
```

Fig 25. VGG16 with DNN layer model summary

9.2. **Best model confusion matrix**

```
Confusion matrix, without normalization
[[414  81]
 [ 91 414]]
```

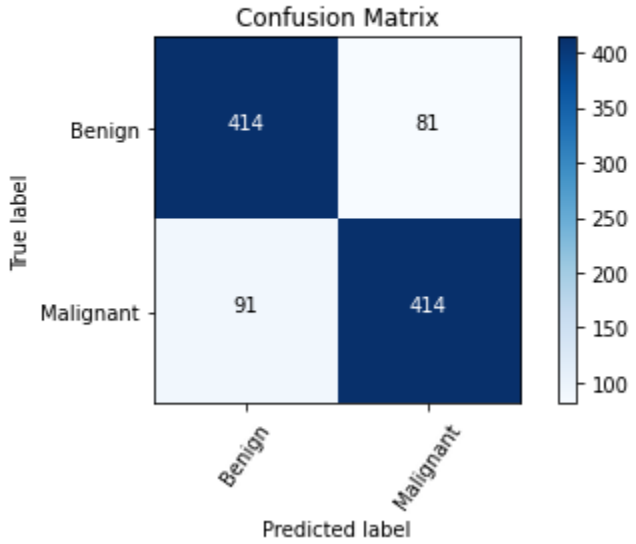


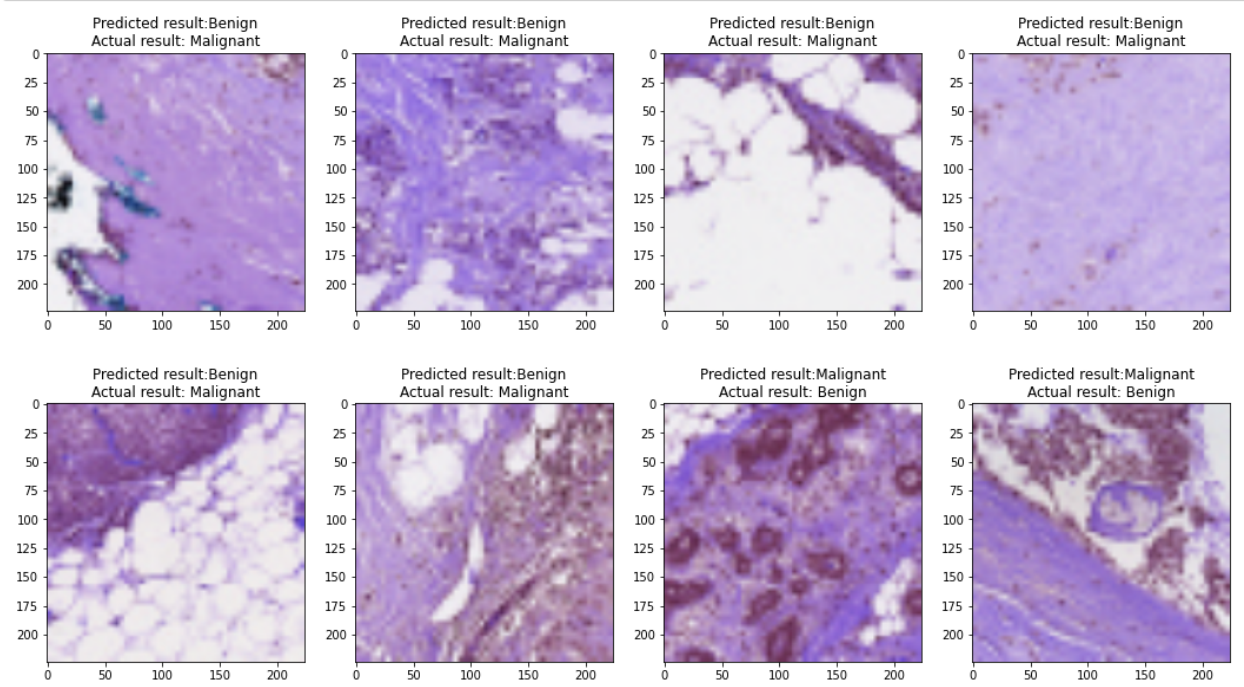
Fig 25. VGG16 with DNN layer model confusion matrix

9.3. **Best model precision, recall and support**

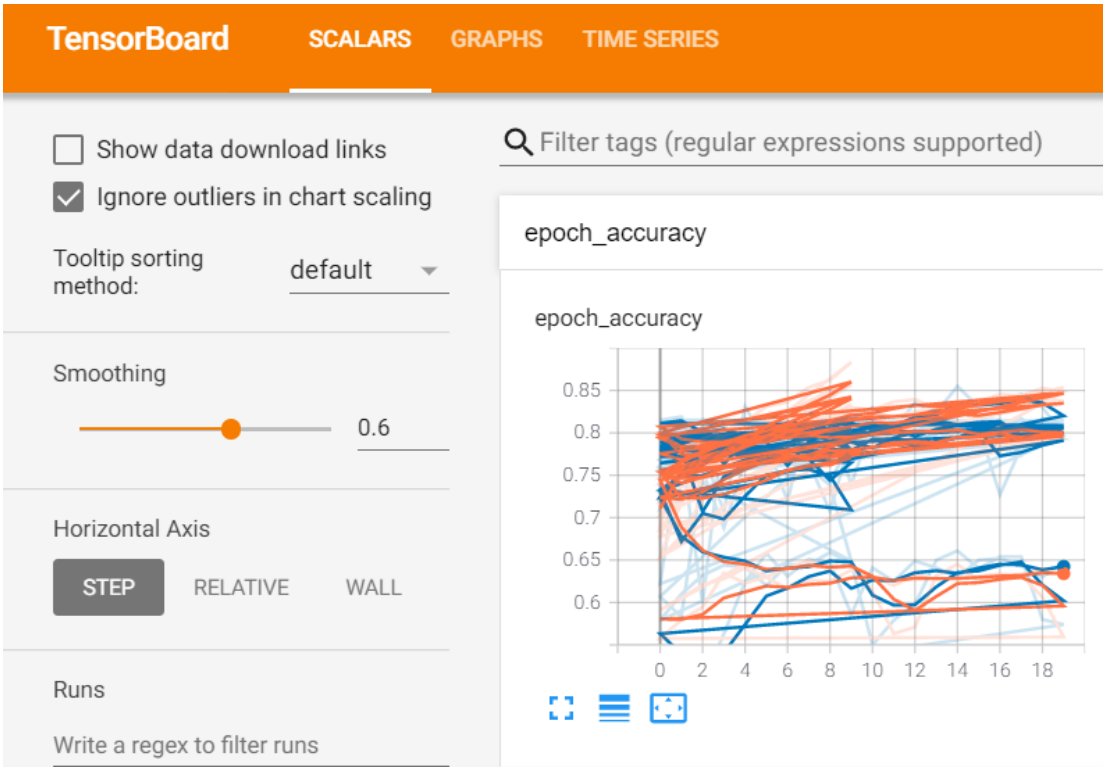
	precision	recall	f1-score	support
negative	0.82	0.84	0.83	495
positive	0.84	0.82	0.83	505
accuracy			0.83	1000
macro avg	0.83	0.83	0.83	1000
weighted avg	0.83	0.83	0.83	1000

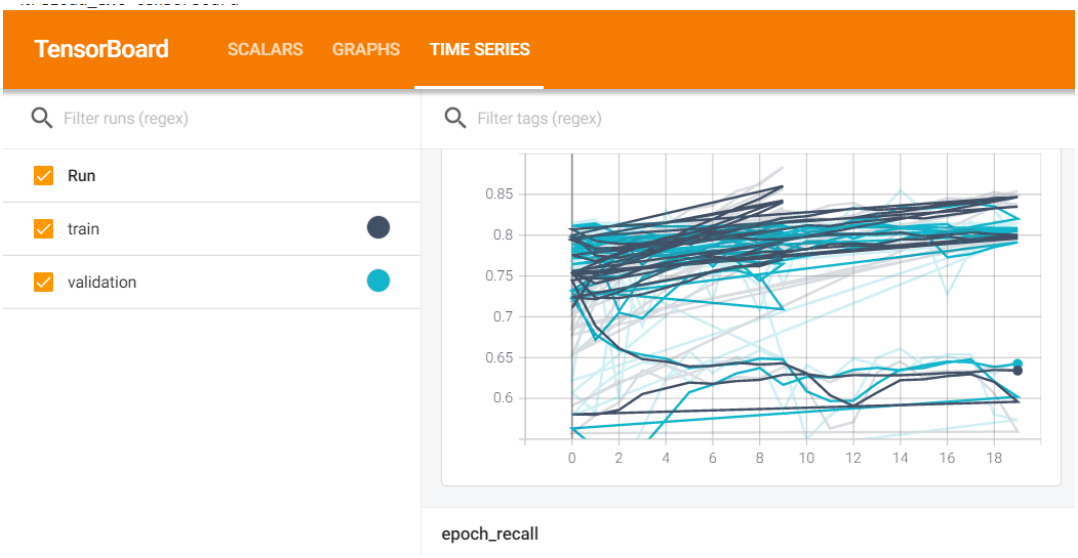
Fig 26. VGG16 with DNN layer model classification report

9.4. **A few common errors of prediction were also captured**



10. **The Tensorboard was also integrated** and used to evaluate results. A few snapshots from the tensorboard are captured below.





11. With the best model tuned, **tensorflow serving was used** to deploy the model on localhost and a REST API query was made to it to test predictions before putting the model in production with an end-to-end machine learning pipeline leveraging TFX, EC2 and flask .

```
!pip install -q requests
headers = {"content-type": "application/json"}
json_response = requests.post('http://localhost:8501/v1/models/breast_cancer_model:predict', data=data, headers=headers)
predictions = json.loads(json_response.text)['predictions']

for i in range(len(test_images[1:3])):
  print(np.argmax(predictions[i]),y_test[i])
```

```
0 [1. 0.]
1 [0. 1.]
```

12. Finally, the best model was saved to disk.

End to End ML Pipeline

In this project, **we developed an end-to-end machine learning pipeline**, starting with data loading through preprocessing, training, deployment and serving. To achieve a full-stack framework we integrated TensorFlow Extended (TFX) and Kubeflow Pipelines together and set up a production-level end-to-end ML pipeline.

It features configuration framework and shared libraries to automate the process of defining, launching, and monitoring model building systems.

A description of the different components of the End to End ML Pipeline is detailed below.

TensorFlow Extended (TFX framework)

1. The TFX platform allows users to build customized ML pipelines, which can use different orchestrators and different execution engines. Through the implementation of TFX, best practices for ML lifecycle management, such as the underneath can be ensured.

1. Python-based classes for defining components
2. Strongly-typed artifacts in the form of components input/output
3. Artifact and execution tracking metadata storage
4. Configuration of pipelines through text editors or notebooks
5. Workflow execution supported by open source orchestrators: Kubeflow
6. Adaptability and portability

2. TFX has a layered architecture that maintains coordination of execution across its components. Data ingestion is facilitated through TFX components that form the pipeline.Each component reads its dependencies from the metadata store and writes back its output/ artifact.

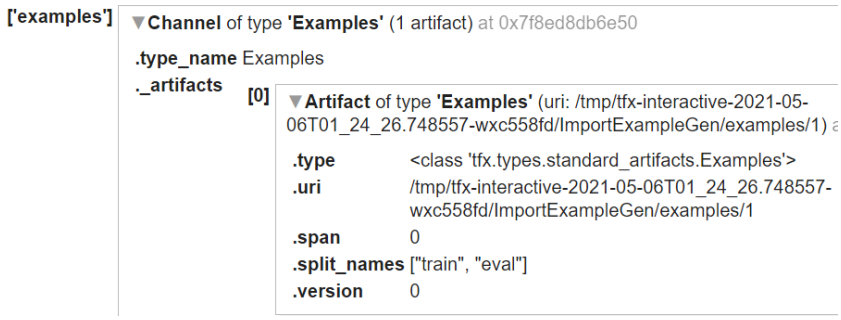
3. The adapted TFX components from our project are outlined below.

Preprocessing

In this step, we loaded the breast cancer image data directly from Kaggle source. Then we converted the csv image data into TFRecords. TFRecords is Tensorflow's standard file format. TFRecords improves our performance time by speeding up the processing of our large image dataset. It stores data in a series of binary strings, making it simpler to distribute and more efficient to read.

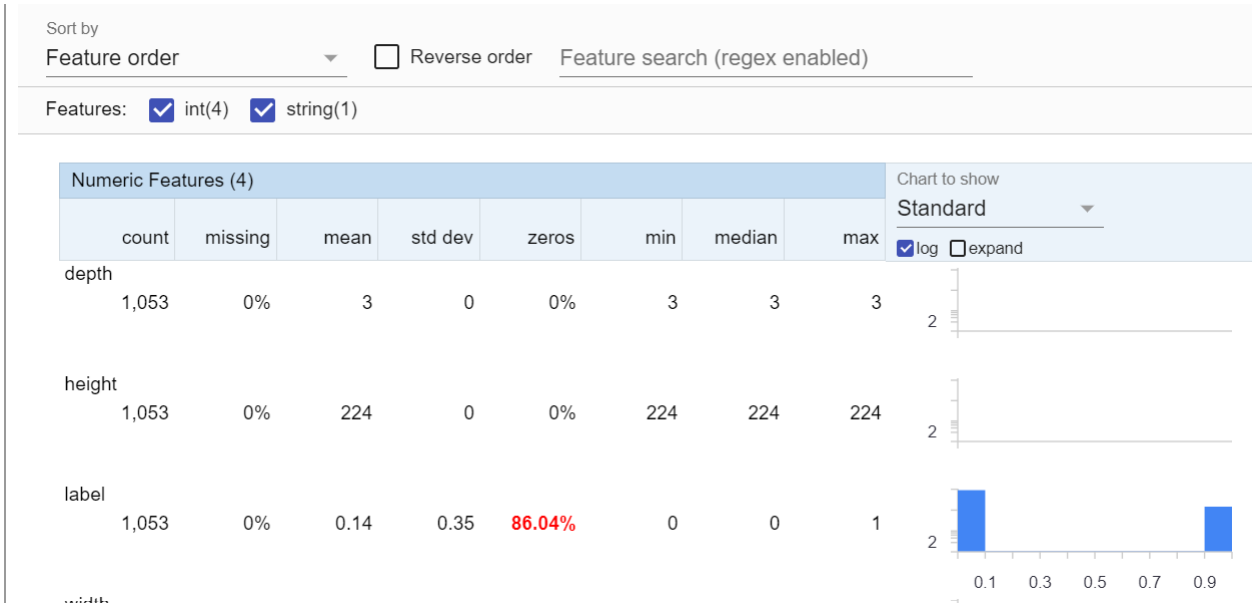
ExampleGen

This TFX component takes the TFRecord from the preprocessing step and generates TensorFlow examples. In addition, it splits the examples into Train and Eval. The result is then passed on to the StatisticsGen component.



StatisticsGen

StatisticsGen provides useful statistics to analyze the data and learn more about its characteristics with visualization. **Here is an image showing the output statistics from a portion of the breast cancer training dataset.**



SchemaGen

SchemaGen assesses the statistics and generates a schema for the data. The component checks the input data types to see whether they're int, categorical, etc., and checks the validity of the data types. **Finally, the component also visualizes inferred schema such as the example shown below:**

```
1 context.show(schema_gen.outputs['schema'])
2
```

Artifact at /tmp/tfx-interactive-2021-05-06T01_24_26.748557-w

Feature name	Type	Presence	Valency	Domain
'image'	BYTES	required		-
'depth'	INT	required		-
'height'	INT	required		-
'label'	INT	required		-
'width'	INT	required		-

ExamplesValidator

ExamplesValidator checks for errors in the data (missing values, 0 values that should not be 0) and reports any anomalies. Both the training and eval data show no anomalies.

'train' split:

```
/usr/local/lib/python3.7/dist-packag
  pd.set_option('max_colwidth', -1)
```

No anomalies found.

'eval' split:

No anomalies found.

Transform:

Transform combines data from ExampleGen with schema generated by SchemaGen to determine arbitrary complex logic, which is based on the data and model's needs, E.g. to perform features engineering. To eliminate the training-serving skew, this step performs the same feature engineering with the same code both during training and production.

```
transform.outputs
```

```
{'transform_graph': Channel(
    type_name: TransformGraph
    artifacts: [Artifact(artifact: id: 5
type_id: 13
uri: "/tmp/tfx-interactive-2021-05-06T01_24_
custom_properties {
  key: "name"
  value {
    string_value: "transform_graph"
  }
}
custom_properties {
  key: "producer_component"
  value {
    string_value: "Transform"
  }
}
```

Trainer

The trainer component performs the model training. Logs can be visualized to study the different performance metrics and understand the training process in greater detail and can compare the execution runs. We have chosen one of our base models and architecture for training a model.

```
_labels_path = os.path.join(_data_root, 'labels.txt')
trainer = Trainer(
      module_file=module_file,
      custom_executor_spec=executor_spec.ExecutorClassSpec(GenericExecutor),
      examples=transform.outputs['transformed_examples'],
      schema=schema_gen.outputs['schema'],
      transform_graph=transform.outputs['transform_graph'],
      train_args=trainer_pb2.TrainArgs(num_steps=150),
      eval_args=trainer_pb2.EvalArgs(num_steps=1),
      custom_config={'labels_path': _labels_path})
```

```
context.run(trainer)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_ker
58892288/58889256 [==============================] - 0s 0us/step
Epoch 1/30
4/4 [==============================] - 34s 3s/step - loss: 0.3885 - sparse_categorical_accuracy: 0.8906 - val_loss: 0.4259 -
Epoch 2/30
4/4 [==============================] - 7s 2s/step - loss: 0.3509 - sparse_categorical_accuracy: 0.8948 - val_loss: 0.3189 - v
Epoch 3/30
4/4 [==============================] - 7s 2s/step - loss: 0.3335 - sparse_categorical_accuracy: 0.8906 - val_loss: 0.2946 - v
```

Evaluator

The evaluator analyzes the training results and validates the exported models to ensure that they are "good enough" for production use. Different validation modes are used to compare the different versions of the model, e.g. a production model against a

newly generated model. In other words, post a successful evaluation, the generated model passes all deployment tests and is ready to deploy into production.

Pusher

This component manages the process of pushing the trained (validated) model to different deployment options. Our trained model is pushed to AWS storage for serving in the image below.

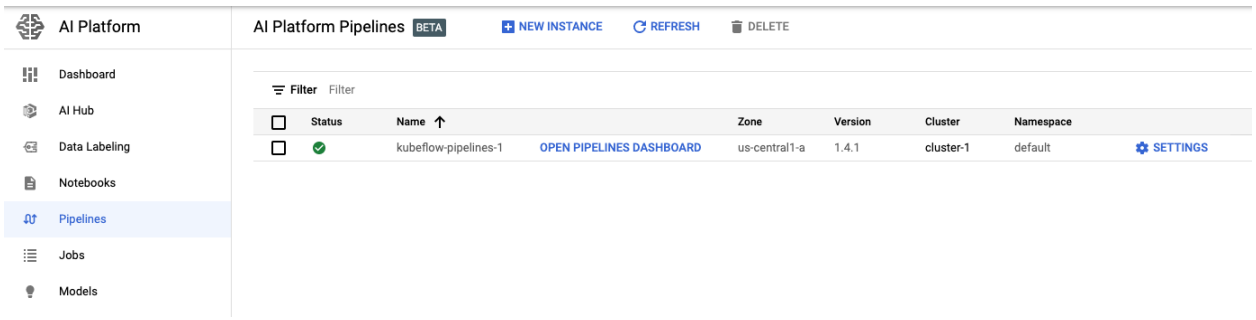
Kubeflow

Kubeflow is a cloud platform for machine learning operations. It consists of pipelines, training, and deployment automatically on the cloud.[4] In addition, Kubeflow is set up to be very simple, portable and scalable on the Kubernetes platform.[5] Essentially this is the modern version of Jenkins for Machine Learning Engineers.

Our KubeFlow implementation highly depended on the completion of TFX to follow common best practices. There are several well supported components and libraries for TFX in Kubeflow which allows for this to be implemented robustly and with great ease.

KubeFlow Setup

Everything KubeFlow related is deployed on GCP on the AI Platform.[6] We were able to use the KubeFlow package from the GCP marketplace to install KubeFlow with no difficulty.[7] The results can be visualized on a single cluster KubeFlow mounted and running machine on GCP as is seen in the following figure:



Developer Experience

We added our KubeFlow TFX code onto a Jupyter Notebook. This allows us to deploy easily and run our pipeline on KubeFlow. Our notebook is a setup for Developer mode which allows users to:

1. Retrieve the code from GitHub
2. Build and deploy any new changes made to Kubeflow
3. Run a Pipeline on KubeFlow
4. Deploy any changes right back to GitHub

All these requirements are set up at the push of a button for the developer. Here is a snippet from our notebook describing the build, deploy and run step:

Kubeflow Build, Deploy & Run

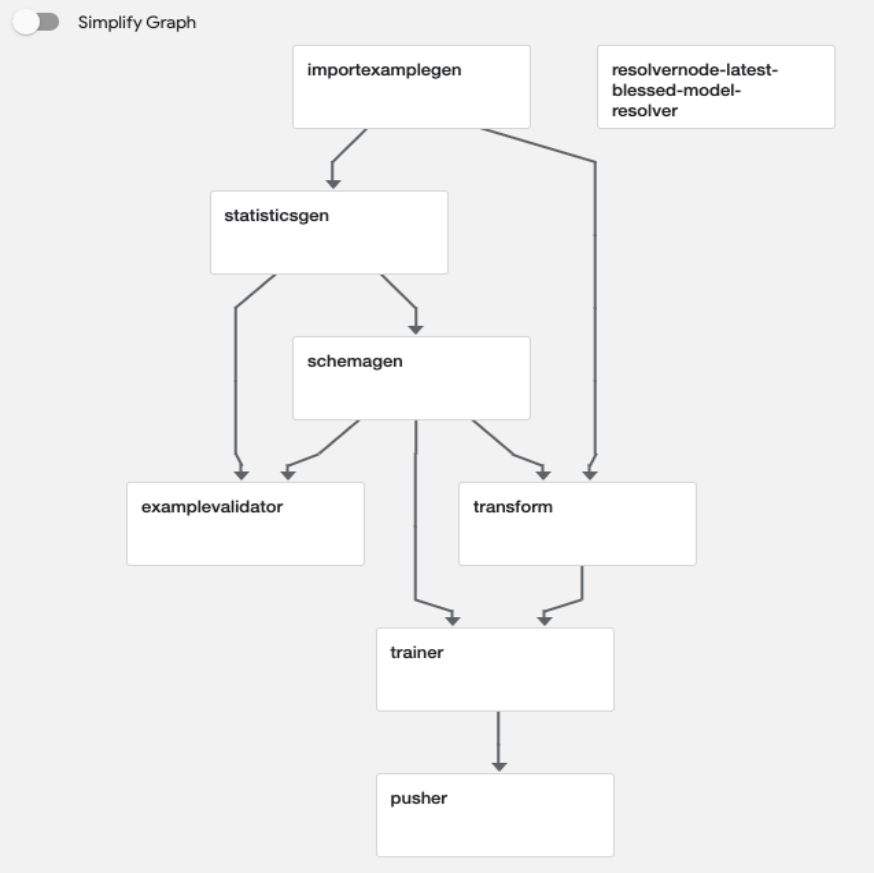
```
# Update the pipeline
!tfx pipeline update \
--pipeline-path=kubeflow_runner.py \
--endpoint={ENDPOINT}
```

```
# You can run the pipeline the same way.
!tfx run create --pipeline-name {PIPELINE_NAME} --endpoint={ENDPOINT}
```

Running On KubeFlow

After you execute the build and deploy cell, a new version of the pipeline is deployed to KubeFlow. This can be viewed in the "Pipeline" section in KubeFlow. Here we can see this in action on KubeFlow:

Notice that previous versions are available for the developer to see changes in the pipeline from one version to another. Should you click into a specified version, you will see a graph of the different steps in the pipeline:



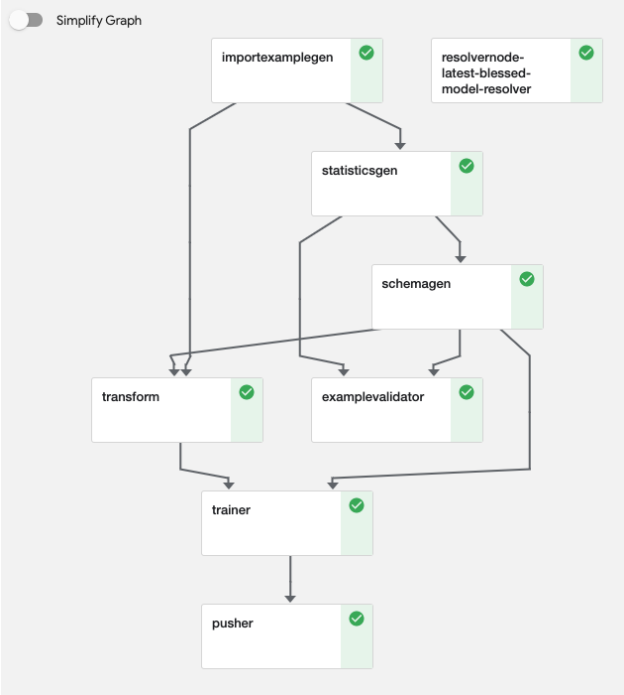
In the "Run" section in KubeFlow, we can view all the different executions of the different versions deployed to KubeFlow. Deploying doesn't trigger a run, but the user has the capability to execute a run from the notebook once the build and deploy of the pipeline is successful. Here is a history of some of our runs:

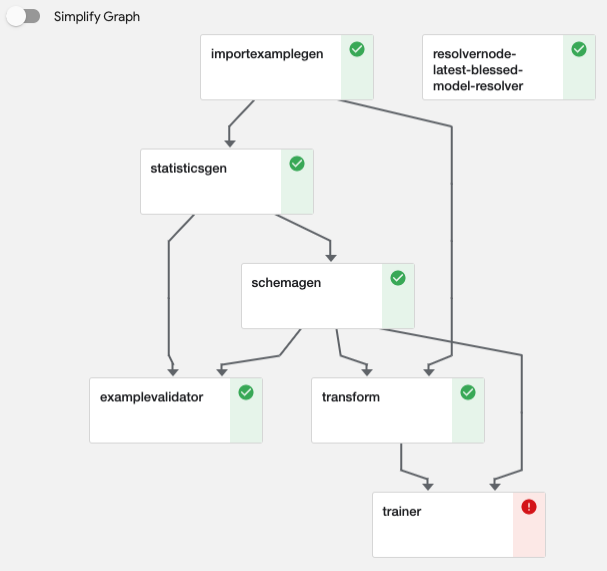
Run name	Status	Duration	Experiment	Pipeline Version	Recurring Run	Start time
cancer_cls_pipeline	✓	0:19:20	cancer_cls_pipeline	cancer_cls_pipeline_20210508020...	-	5/7/2021, 7:00:45 PM
cancer_cls_pipeline	✓	0:20:22	cancer_cls_pipeline	cancer_cls_pipeline_20210508015...	-	5/7/2021, 6:55:57 PM
cancer_cls_pipeline	✓	0:20:09	cancer_cls_pipeline	cancer_cls_pipeline_20210508012...	-	5/7/2021, 6:29:52 PM
cancer_cls_pipeline	!	0:00:22	cancer_cls_pipeline	cancer_cls_pipeline_20210508010...	-	5/7/2021, 6:06:36 PM
cancer_cls_pipeline	✓	0:19:17	cancer_cls_pipeline	cancer_cls_pipeline_20210508010...	-	5/7/2021, 6:05:55 PM
cancer_cls_pipeline	✓	0:00:16	cancer_cls_pipeline	cancer_cls_pipeline_20210507070...	-	5/7/2021, 5:19:40 PM
cancer_cls_pipeline	✓	0:22:29	cancer_cls_pipeline	cancer_cls_pipeline_20210507070...	-	5/7/2021, 12:02:08 AM
cancer_cls_pipeline	!	0:13:08	cancer_cls_pipeline	cancer_cls_pipeline_20210507064...	-	5/6/2021, 11:44:00 PM
cancer_cls_pipeline	!	0:04:56	cancer_cls_pipeline	cancer_cls_pipeline	-	5/6/2021, 11:36:36 PM

Rows per page: 10

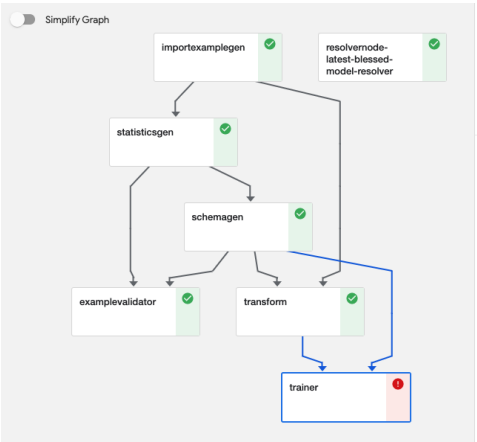
We have both successful and failed runs on KubeFlow. The most common errors we've faced are low resource allocation or mistakes made in the implementation of the

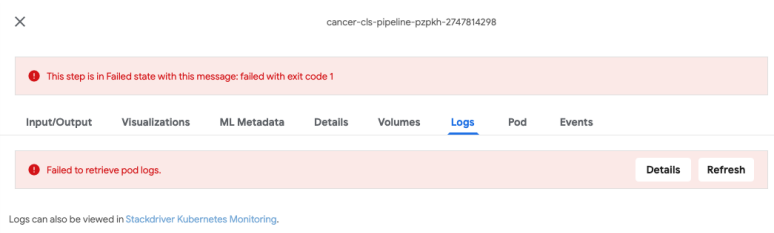
training step. If we click into the individual runs, we can see the status of each step in the pipeline and narrow down where a failure occurred.





A successful run shown on the left, a failed attempt on the right. We can see in the successful run that all steps are checked green. In the failed attempt, we can see that in the trainer step, there was a failure. By clicking on the step we can get more information and troubleshoot our problem:



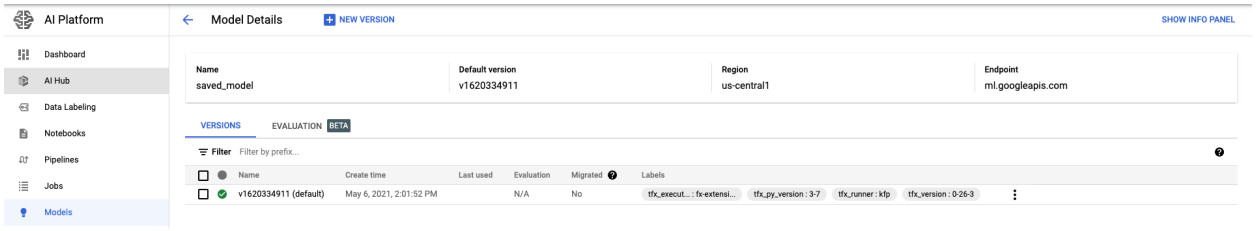


Note that there isn't much information here. Actually there are no logs here. This is because during our runtime, we tell this step to run in a separate cluster. In order to see these logs, we need to go to the cluster logs to view the error. The reason for this is

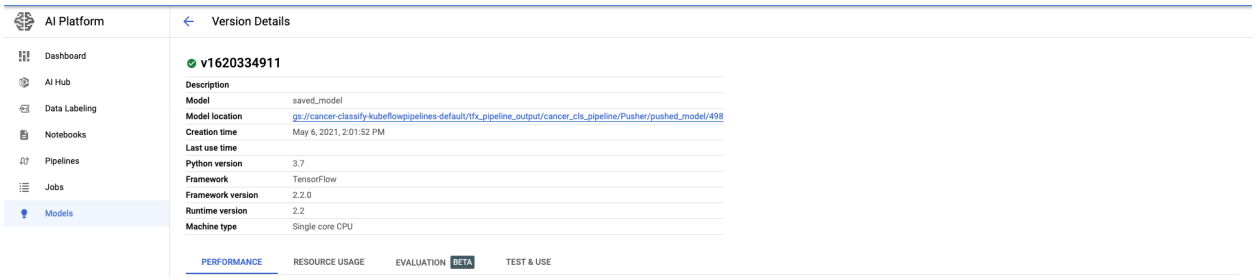
training takes up a lot of resources. Having a separate isolated instance in which this runs allows it to have more resource allocation to run successfully.

Our Pipeline Details

After building a sound understanding of KubeFlow, let's take a look at our pipeline. Our pipeline mimics the TFX steps from earlier. **This consists of loading data, inspecting data, training, evaluating and deploying**. In the pusher step, instead of pushing to a local filesystem, we deploy the model. This then allows our application to make requests to the latest model whenever a deployment happens, automatically.



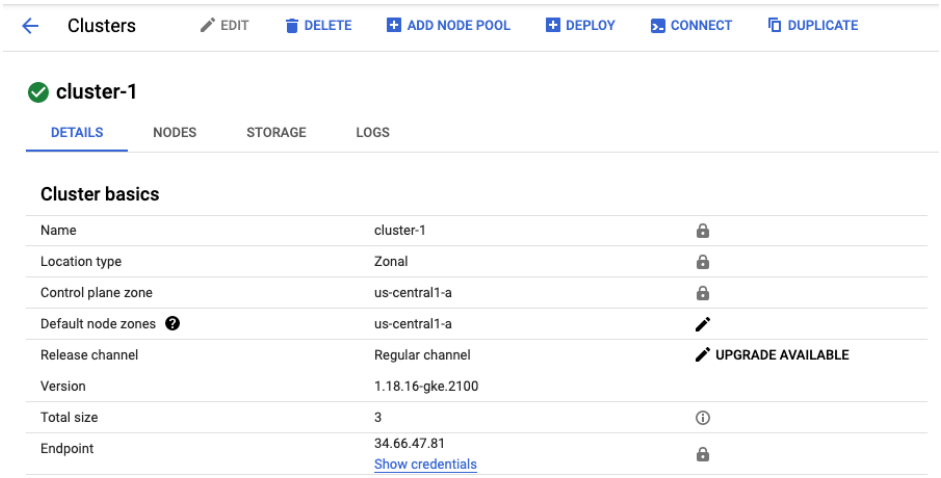
This is automatically generated and versioned from the Pusher step at the end of the pipeline. To see even further details, see the following:



We can clearly see there is a model which is served and versioned.

Resource Allocation For KubeFlow

Since KubeFlow is set up with Kubernetes, we can easily modify it and add nodes to the pool with different configurations. This is necessary to enable our instances to get higher memory and compute resources such as GPUs. By going to the Kubernetes Engine section of GCP, you can view all the details and configure your cluster based on need.



Web Application

We created a very simple web app using flask, a simple web framework designed to create quick and very easy web applications. Our app uses bootstrap to automatically make the content of the page look prettier.

An interesting implementation we added was the direct connection from the application to the served model on the google AI Platform on GCP via google API calls. This allows us to change model version numbers in the flask app whenever a new model is deployed. The new model gets automatically picked.

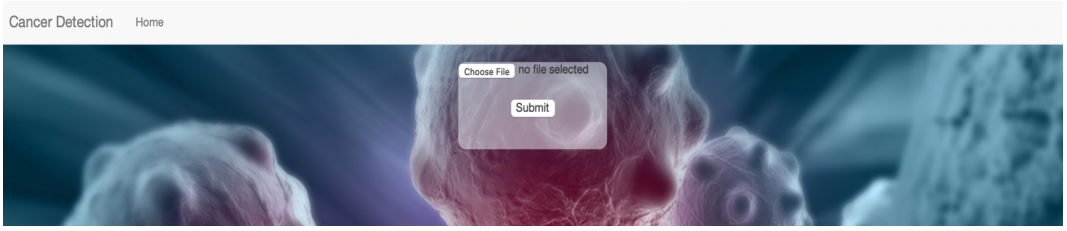
The snippet below captures the google API invocation from within the flask application.

```
def predict_json(project, model, instances, version=None):
    """Send json data to a deployed model for prediction.

    Args:
        project (str): project where the Cloud ML Engine Model is deployed.
        model (str): model name.
        instances ([Mapping[str: Any]]): Keys should be the names of Tensors
            your deployed model expects as inputs. Values should be datatypes
            convertible to Tensors, or (potentially nested) lists of datatypes
            convertible to tensors.
        version: str, version of the model to target.
    Returns:
        Mapping[str: any]: dictionary of prediction results defined by the
            model.
    """
    # Create the ML Engine service object.
    # To authenticate set the environment variable
    # GOOGLE_APPLICATION_CREDENTIALS=<path_to_service_account_file>
    service = googleapiclient.discovery.build('ml', 'v1')
    name = 'projects/{}/models/{}'.format(project, model)

    if version is not None:
        name += '/versions/{}'.format(version)

    response = service.projects().predict(
        name=name,
        body={'instances': instances}
    ).execute()

    if 'error' in response:
        raise RuntimeError(response['error'])

    return response['predictions']
```

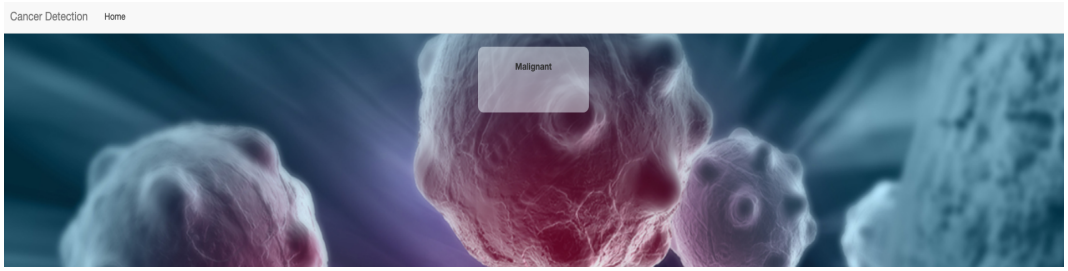
We also have the capability of setting the model version to an older model, should there arise any issues with the latest model. We can also add a feature or change the model version directly in the code to reflect it until the latest model deployed on google AI Platform is not fixed.

Demo

A quick demo of our app is presented underneath. Should you visit the following page, http://cd.wasaequreshi.com, you see the following web page.



Should you now upload an image and hit submit, t**he flask app will send the request to the model on GCP and return the image classification prediction on the UI as shown underneath. Here the image uploaded represents Malignant Cancer.**



Conclusion

In this study , we have successfully implemented a breast cancer detection approach using Transfer Learning and employed 8 different pre-trained models. Furthermore, we have adapted the LSTM-CNN structure using transfer learning and achieved an accuracy of <<%>>. We have also conducted an extensive experiment to identify the best model. Using TFX and Kubeflow we have implemented a CI/CD ML Ops level-2 workflow to implement a full-fledged ML pipeline. This completely automated ML pipeline enables us to handle all of these phases in one go, from loading data to train our models and from validation of our models to deployment and serving our models. In recent years, many researchers are studying the Transfer learning work for cancer detection from images. In the future, this study would enable us to solve more complicated problems with cancer detection or with image classification problems using limited labeled data.

References

1. Breast Cancer Detection Using Transfer Learning in Convolutional Neural Networks. 2017.
2. Breast cancer histology images classification: Training from scratch or transfer learning, 2018.
3. Image Classification using a Hybrid LSTM-CNN Deep Neural Network, 2019.
4. KubeFlow
    - https://github.com/kubeflow/kubeflow
    - https://www.kubeflow.org/
    - Quick Start With KubeFlow Pipelines on Google Cloud Platform
5. Installation Options for KubeFlow Pipelines