Skin Lesion Classification

S E 492
Team sdmay23-05
April 29, 2023
Final Report

Team Members
Adam Sweiger - On-Ground AI Developer
Asad Abdalla - User Interface Developer
Rashed Alyammahi - User Interface Developer
Mohammed Elbermawy - User Interface Developer
Yannick Fumukani - User Interface Developer
Richard Gonzalez - On-Cloud AI Developer
Meet Patel - User Interface Developer

Client/Advisor
Dr. Ashraf Gaffar, Teaching Professor[E CPE]

Table of Contents

Revised Project Design 3
Implementation Details 8
Deployment 9
1. Overview 9
2. AWS SageMaker 10
Overview 10
Use of Sagemaker in this project 10
3. Model and endpoint 11
Overview 11
Use of endpoint workflow in this project 11
4. Lambda function 11
Overview 11
Use of lambda function workflow 11
5. API Gateway 12
Overview 12
Use of API Gateway workflow 12
6. NodeJs Server 12
7. UI 13
Overview 13
Use of React.JS in this project 13
Testing Process and Testing Results 14
Work Context 16
Appendix I - Operation Manual 17
Appendix II - Alternative Designs 22
Appendix III - Other Considerations 23
Appendix IV - Code 24

Revised Project Design

This project is composed of four main components: an AI image classification model that classifies skin lesions as benign or malignant, a virtual machine for on-ground model training, an AWS Sagemaker Notebook instance for on-cloud model training, and a User Interface for running predictions using the trained model.

Implementation Architecture
Our project design aims to train the model in a local VM environment and Cloud AWS. Figure 1 represents our approach to the project.

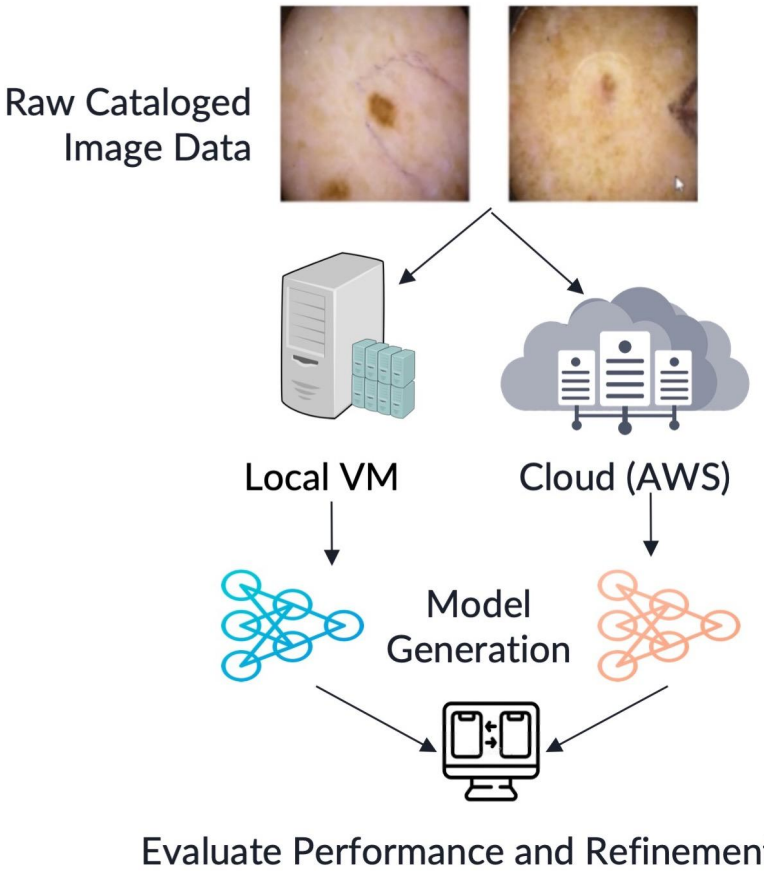


Figure 1: Local VM vs Cloud approach utilized in ascertaining performance differences

We used VM where storage and RAM have been allocated by the University. They allocated us the new linux kernel which used GPU for the research project work. This was apparent to us as the best option to us to take advantage of the high storage and allocated memory for the project. We used the Conda Tensorflow environment to train our model.

We collected a dataset of skin images from the International Skin Imaging Collaboration (ISIC). We used publicly available libraries for skin lesions.

During training, the model will output information about the current training iteration including time spent training, accuracy, etc. Once the model has been trained and we are using it to predict if the particular image has skin cancer or not the output will be the result of the prediction, either benign or malignant, as well as the confidence in this prediction as a percentage

Skin Cancer Classification in Medical Terms:

Benign: an indication of not having skin cancer. In our case, the model will provide output based on its confidence if the particular image has skin cancer or not.

Malignant: an indication of having skin cancer. In our case, the model will provide output based on its confidence if the particular image has skin cancer or not.

We trained the model in Conda TensorFlow with about 67,000 images and achieved an accuracy of 92%.

Afterward, we used Amazon Sagemaker Notebook to train our model on the cloud. We efficiently scaled research using AWS (200, 2K, 10K images). We used JupyterLab IDE and Conda Tensorflow to train our model. We evaluated and compared the performance in VM and on Cloud. We achieved comparable results to the expensive VM GPU used at Iowa State ETG.

Dataset

The data stems from the International Skin Imaging Collaboration (ISIC) publicly-available library of skin lesions. The organization represents a collaborative effort between academics and industry agents toward developing methods of recognizing and detecting melanoma. They provide a vast open-source library of images of skin lesions of known classifications, with a broader goal of classifying other skin disorders. Throughout this project, we trained our model using images found through the ISIC archive's benign/malignant filter. The total number of benign and malignant images in the archive is 59,676 and 7,061 images respectively. We started training on a small subset of these images and incrementally scaled the dataset before eventually training on the full 67,835 images.

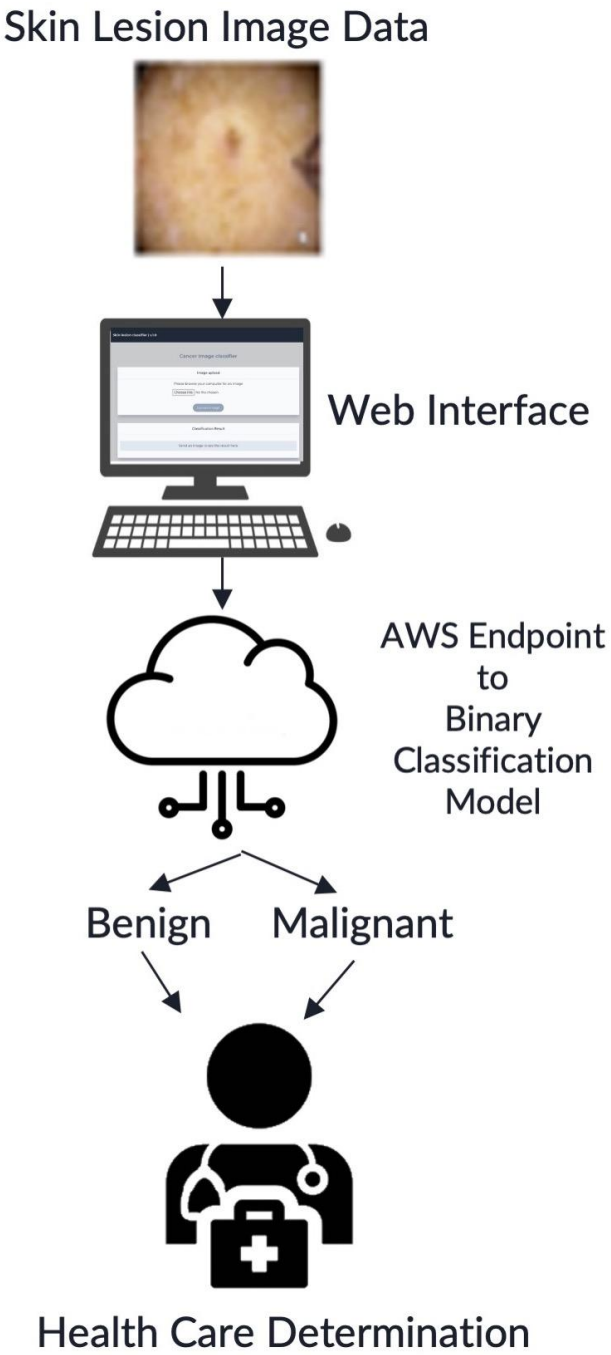


Figure 2: Schematic for Web Interface interaction with user given a sample image to pass into model

We used AWS EndPoint for harnessing the web gateway into the generated model. We used API Lambda Function to link Endpoint to API Gateway and post method. Afterward, we used Node.js as a helper backend for image conversion with Heroku. We used Web Interface for uploading and requesting predictions from the model. This schema is presented in Figure 2.

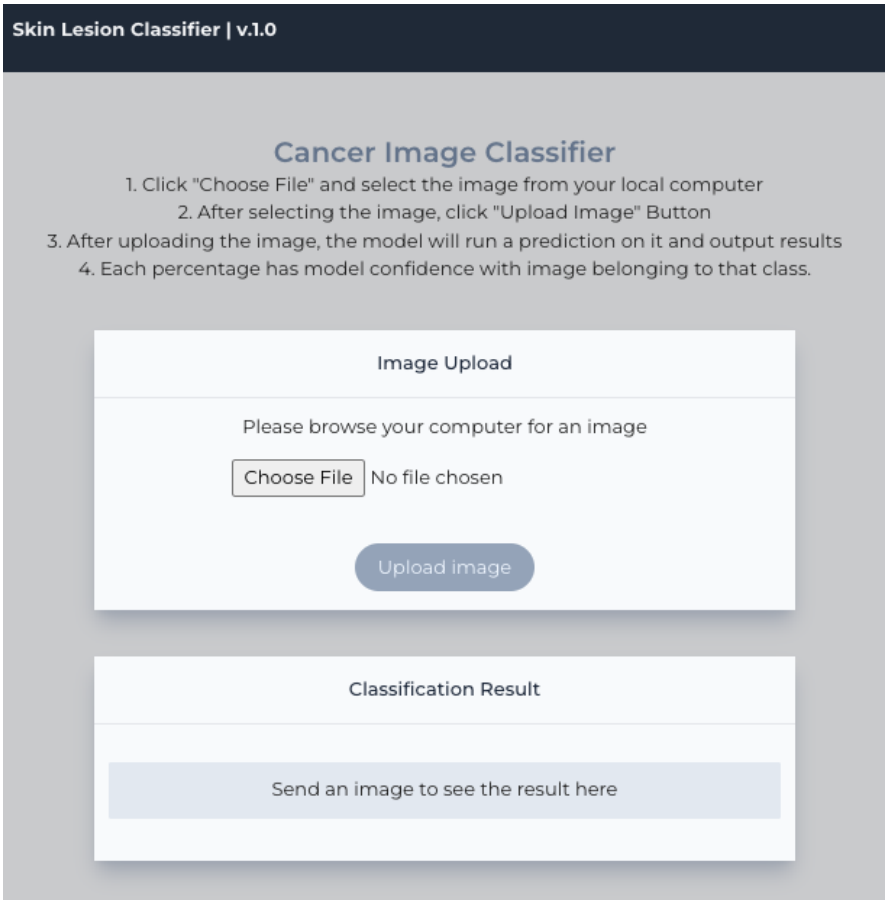


Figure 3: Skin Lesion Website Landing Page

AWS Endpoint is used to store web gateway into the generated model. Frontend UI encodes the image to 64-bit before sending it to the server for model prediction, shown in Figure 3.

How The Design Evolved

In 491, the team considered developing an image classification model to classify livestock animals as healthy or sick. The proposed model would take audio files of sounds made by animals, transform them into spectrogram (image) representations, and then analyze the image representation to determine what patterns exist for healthy and sick animals. As part of the research process, the team looked into existing binary image classification examples. At the beginning of 492, the team received a dataset of livestock animal sounds, but it was a small dataset of only a few hundred samples with very low audio quality. The team decided that this dataset was not sufficient enough to train a capable model, so instead, the team began working towards developing and training a skin lesion classification model. The team had access to a large, publicly available dataset of high-quality images of skin lesions. In addition, classifying skin lesions can assist in skin cancer detection and diagnosis. Although the applications are different, the core problem of binary image classification is the same, so the same model architecture can be applied to both problems.

Requirements

1. Functional Requirements:
    - AI model must be trained on images of benign and malignant skin lesions.
    - AI model should be scalable and should be able to train on datasets of varying sizes up to 67,000 images.
    - After training, the model should be able to analyze an image of a skin lesion and classify the lesion as benign or malignant.
    - The accuracy of the AI model should improve with each iteration of training.
2. Non-Functional Requirements:
    - The accuracy of the model should be at least 80%.
    - When trained with the same hyperparameters, the model's accuracy and training speed should be comparable between the on-ground environment and the on-cloud environment.
    - When a user uploads an image to the user interface, the model's prediction should be outputted to the user within 2 seconds.

Relevant Standards

1. IEEE 1003.1-2008 - This standard is simultaneously ISO/IEC 9945, IEEE Std 1003.1, and forms the core of the Single Unix.

2. IEEE 1680.1-2009 - This standard ensures consistent environmental performance for the resources used.

Engineering Constraints

- Limited storage space for on-ground training environment: When the virtual machine was first set up by ETG for the team, it only had about 60 GB of disk space. At first, this was sufficient to store the dataset, but as the dataset was scaled, this became too little storage space. In order to allow the dataset to scale, the team requested additional storage from ETG, and the disk space on the VM was increased to 150 GB, which was plenty of space to store the full dataset.
- Limited computing power: The team's virtual machine has access to a powerful GPU that is incredibly effective for machine learning, but as the dataset scales, training becomes slower. The same is true for the computing resources on AWS. The computing resources are a constraint that slows down the training process.

Security Concerns and Countermeasures

1. Physical Security: Because this project is entirely software-based, there are no concerns for physical security.

2. Cybersecurity: The only cybersecurity concerns on this project are ensuring that only the team has access to the virtual machine and AWS accounts. External parties gaining access to the virtual machine could lead to the team losing important data, files, and code. Comprising the security of an AWS account can lead to unwanted charges and allow others to gain access to team members' payment information. The team is not concerned with the security of our AI model's dataset because it contains publicly available data.

Implementation Details

AI Model

The model was adapted from an existing image classifier from a Keras tutorial. We generate a dataset of images with two labels, Benign and Malignant. This dataset is split into a training set containing 80% of the images and a validation set containing 20% of the images. The dataset is bolstered artificially through data augmentation, in which random transformations are applied to the training images. This allows the model to analyze different aspects of the training data and slows down overfitting. The images' size and color values are standardized to make the neural network process them more efficiently. The model starts with a data augmentation preprocessor, followed by a Rescaling layer and a Dropout layer before the final classification layer. The Dropout layer is used to prevent overfitting. The full code for the model is provided in Appendix IV.

Model Hyperparameters

The following are hyperparameters we configure before training the model:

- Training/validation data split: We use 80% of the dataset to train the model, and 20% to validate the model's ability to classify images. This helps us understand how well our model is performing its assigned task with the given hyperparameters.
- Optimization algorithm: We use the Adam algorithm with a learning rate of 0.001 as our optimizer. Keras offers 10 optimizers to choose from, but we selected Adam because the image classification tutorial the model is adapted from uses Adam, and because Adam has lower training cost compared to other algorithms.
- Layer activation functions: For the model's hidden layers, we use the ReLU activation function which, for input x, outputs max(0.0, x). For the model's classification (output) layer, we use the sigmoid (logistic) activation function because it is the best output activation function for binary classification.
- Loss function: Because we are working in binary classification, we use the Binary Cross-entropy loss function to calculate the difference between expected and predicted labels.
- Drop-out Rate: The drop-out rate of 0.5 causes half of the input units for the Dropout layer to be set to 0, and the other half to be scaled up so that the sum over all inputs remains the same. This helps prevent overfitting.
- Epochs: The number of epochs defines the number of times that the neural network will analyze the entire training set. We vary the number of epochs (iterations) for training depending on the size of the training set and what our target metrics are. For instance, with a dataset of 2,000 images, we train for 25 epochs before the accuracy stops growing at a significant rate. With the full dataset of 67,835 images, we can already achieve an accuracy of 92% after only 10 epochs.
- Batch size: The batch size dictates the number of samples encountered in training before the model is updated. The team experimented with batch sizes of 128 and 32 before finding that a batch size of 16 gave the best results.

Deployment

1. Overview

   The team used a multi-layered architecture to deploy a model consisting of AWS services such as SageMaker, Lambda function, API Gateway, and a NodeJS server. This architecture allowed for the deployment of a scalable, production-grade infrastructure to serve user predictions regarding the malignancy of a particular skin lesion image. The overall data flow of information is shown in Figure 4.

   Using SageMaker allowed for the creation of an endpoint (i.e. the URL of the entry point for an AWS web service) for the generated model, a managed infrastructure that can serve malignancy predictions to users. To interact with this endpoint, a Lambda function that invokes the endpoint and returns the prediction as a JSON file was utilized. However, since direct requests cannot be sent to Lambda, an API Gateway was added as an additional layer that triggers the Lambda function.

   To further enhance the user experience, a NodeJS server was incorporated to more easily allow the user to upload and request a prediction from the model, without directly communicating with it in a secure shell command line interface. This server sends requests to the API Gateway, which triggers the Lambda function and ultimately triggers the model endpoint. This multi-layered architecture allowed for the creation of a robust, scalable infrastructure for serving predictions to users while providing a seamless user experience.

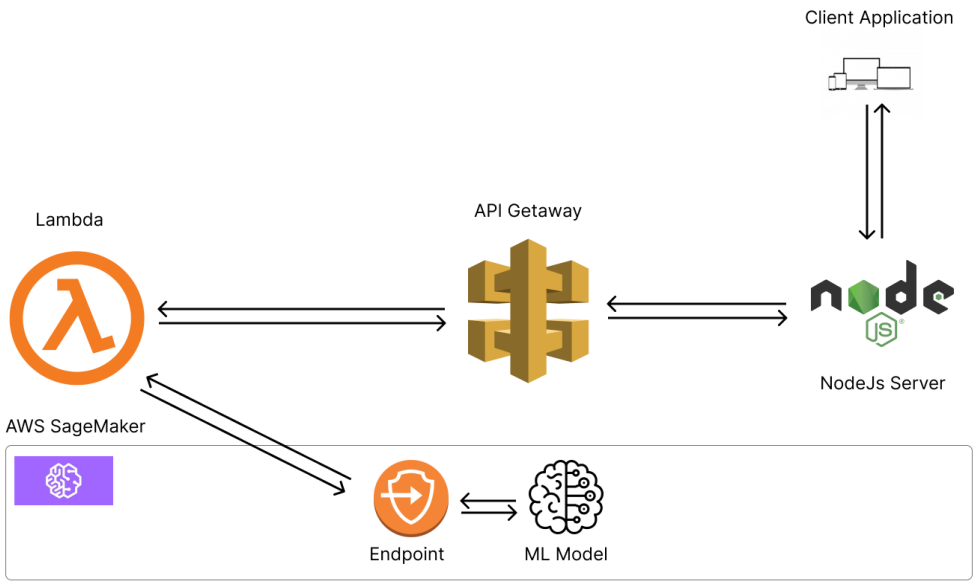


Figure 4: Deployment of model toward the generation of graphical user interface

2. AWS SageMaker

Overview

SageMaker is a fully-managed service provided by Amazon Web Services (AWS) that enables developers and data scientists to build, train, and deploy machine learning models at scale. It offers tools and services that streamline the entire machine-learning workflow, from data preparation and labeling to deployment and monitoring.

One of the key advantages of SageMaker is that it abstracts away many of the complexities involved in building and deploying machine learning models. It provides a range of pre-built algorithms and frameworks and powerful training and deployment capabilities, making it easier and faster to build and deploy models in production environments. With SageMaker, developers, and data scientists can focus on building better models and delivering value to their organizations rather than worrying about infrastructure and management tasks.

Use of Sagemaker in this project

To deploy the generated machine learning model, a three-step process was undertaken. First, the model file (h5 format) was uploaded and a shared notebook instance was created, providing a collaborative environment for developing and testing our models.

Next, the model was converted to a less proprietary archive format (i.e. tar.gz), which is more common for deploying machine learning models. This allowed for the preparation of the model for deployment in a way compatible with traditional deployment options.

Finally, an endpoint was created for the model and tested locally. This endpoint is a managed infrastructure that allows for the serving of predictions to users. Various tools and services in the AWS ecosystem (i.e. S3, Lambda, etc.) were available to us to create the endpoint and manage its lifecycle. Once the endpoint was created, it was tested locally.

Deploying the machine learning model involved converting it to a deployable file format, creating an endpoint to serve predictions, and testing it locally. As previously noted, this process allowed for the deployment of the model in a manner that is compatible with traditional deployment options and met the stated goals of the project.

3. Model and endpoint

Overview

One of the key features of SageMaker is its ability to simplify the process of deploying machine learning models in production. It provides various options for deploying models, including real-time endpoints for serving predictions, batch transforms for processing large datasets, and hosting models on edge devices. SageMaker also provides automatic model scaling, fault tolerance, and monitoring, which helps ensure that the models are always available and performing optimally.

Use of endpoint workflow in this project

The specific endpoint implementation found in this project involves sending images in the Content-type of Application/x-image. A binary classification model generated by the team was deployed, predicting whether the input image is benign or malignant with a probability of confidence. The response from the endpoint is a binary array of [decimal point, decimal point], which corresponds to the probabilities of the input image being benign or malignant. The first decimal point represents the probability of the image being benign, while the second decimal point represents the probability of the image being malignant.

To summarize, the endpoint workflow in SageMaker involves deploying the model, sending data in the expected format, running the inference pipeline, and receiving the prediction. In this implementation, images are sent in the Content-type of Application/x-image, and the response is a binary array of [decimal point, decimal point] corresponding to the probabilities of the input image being benign or malignant. This implementation allows us to use SageMaker to serve predictions to the user, with the flexibility of customizing the input and output formats based on the specific use case.

4. Lambda function

Overview

The lambda function is a serverless computing service. It allows running code without provisioning or managing servers, enabling to build scalable, event-driven applications. With Lambda, you can upload your code as a function, and AWS runs and scales the function in response to requests or events.

Various events, including HTTP requests, can trigger Lambda functions. When triggered, a function runs in a containerized environment, automatically scaled up or down based on the incoming request volume.

Use of lambda function workflow

We used Lambda to invoke our endpoint. Instead of sending requests directly to our endpoint, we used a Lambda function to handle the request and send it to the endpoint. This approach allows us to add processing or validation before invoking the endpoint and helps us manage the request flow more efficiently.

The Lambda function workflow involves receiving a base64-encoded image as input and using the AWS SDK to send the request to the endpoint. Once the prediction is generated, it is returned as a binary array in the format of [benign, malignant]. The Lambda function returns the result in a JSON format file that includes the prediction, which can be used in our application to make decisions or take actions. By using Lambda, we could simplify the integration process and create a more efficient and scalable system.

5. API Gateway

Overview

API Gateway is a fully managed service that enables developers to create, publish, and manage secure APIs. It allows users to expose their AWS Lambda functions and HTTP endpoints as RESTful APIs, making connecting applications and services to the cloud easy. The service provides a range of features, such as authentication and authorization, making it a powerful tool for building scalable and secure APIs. It acts as a front door for the APIs, allowing developers to define the request and response format, control access, and monitor usage.

Use of API Gateway workflow

The API Gateway workflow starts with a POST request that contains an image in binary format and a specified content-type of application/x-image. This request is sent to our API Gateway endpoint, which is set up to trigger a Lambda function.

The Lambda function invokes our model endpoint with the provided image and receives the prediction result. The Lambda function then returns a JSON response containing the prediction result, a binary array of [0, 1], indicating whether the image is benign or malignant.

The API Gateway serves as an interface between the client and the Lambda function, allowing for seamless communication and triggering of our endpoint. By leveraging API Gateway and Lambda, we created a scalable and efficient infrastructure for serving predictions to our users.

6. NodeJs Server

Adding a NodeJS server to our prediction application provided a solid foundation for building a full-fledged application that includes user registration and authentication. The server acts as a middleware between the client and our API gateway, enabling the client to securely send requests to our prediction application. The server handles user authentication, manages user sessions, and forwards requests to the API gateway.

The complete workflow of our application is as follows: The client sends a request to the NodeJS server, which validates the user's credentials and creates a session. The server then sends a request to the API gateway, passing along the user's session information and the image data. The API gateway triggers the Lambda function, which calls our model endpoint and generates a prediction. The prediction is then returned to the Lambda function, which formats it as a response in JSON format and returns it to the API gateway. Finally, the API gateway returns the response to the NodeJS server, which forwards it to the client.

7. UI

Overview
For our graphical user interface, our team created a Single Page Application. It is simple and easy to use. We built this application using React.js.

An open-source JavaScript package called React.js is mainly used for creating user interfaces (UIs). It was created by Facebook, and Facebook and a developer community are now maintaining it.

React gives programmers the ability to create reusable user interface (UI) components that may be combined to create intricate, dynamic online apps. The main strength of React is its ability to quickly update and render components in response to modifications in the application state without requiring a complete page refresh.

Developers describe the intended user interface, while React takes care of the updates and rendering thanks to its declarative programming approach.

Use of React.JS in this project
Because React.js offers a simple and highly effective way to create UI components, we chose it for our frontend development. We used React's features for every part of our front-end application. HTTP requests are sent through Axios, a third-party module that offers more robust handling of such requests than the native JavaScript fetch API, we made it easier for users to communicate with our backend. We used hooks, and in our instance the useState hook, to update our UI based on HTTP responses.

In conclusion, React is a great UI library that has changed the way many developers create UI today; it’s been a very good choice for our frontend application, and it has made our frontend development simple and easy.

8. Frontend and Backend Connection

Overview:
AWS Endpoint is built upon existed training model. We have stored our model in AWS Endpoint. We used axios module to connect the frontend with the backend. The AXIOS module is an easy-to-use API for making HTTP requests from Web Server to the Backend. AXIOS supports methods such as GET, POST, PUT, and DELETE. When a GET request is sent to the API, then it returns a list of posts in JSON Format.

Use of AXIOS Module
AXIOS module is installed as a dependency. AXIOS dependency is called in the Package.json file. The JSON file is used to send and receive data configuration. When receiving data, AXIOS automatically parses the response data and returns it as Javascript. In the main file, a function is

created to receive the prediction data. When a user uploads the file, this data is sent to aws endpoint and it returns the prediction result.

Testing Process and Testing Results

Machine learning incorporates testing into the training process. For this model, we split our dataset into a training set and a validation set. The training set, comprising 80% of the images, is used by the neural network to learn during each training iteration. The validation set, making up 20% of the images, is used after each epoch to evaluate the performance of the model on new data that it has not been trained on. As the model tests itself during training, it will adjust its behavior in order to improve its accuracy. This is a core part of how the model's performance continues to improve after successive training iterations. The following is a sample training run of 10 epochs for the model on the full ISIC dataset of about 67,000 images, as illustrated in Figure 5.

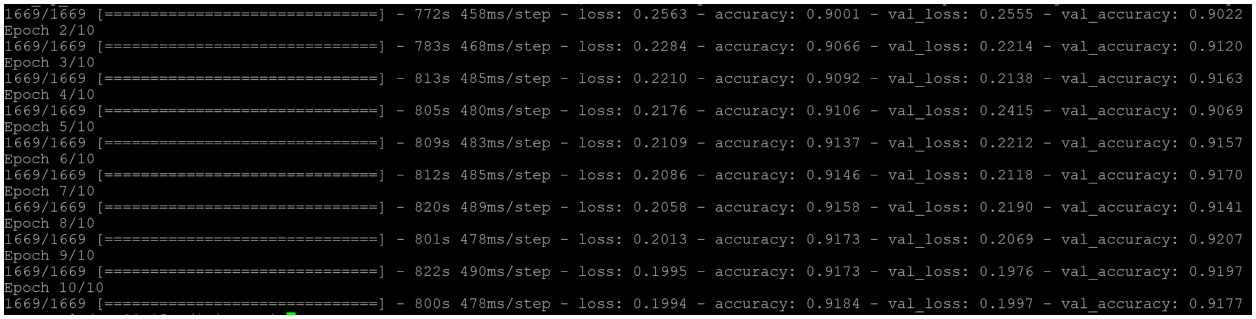


Figure 5: Demonstration of 10 epochs run on full ISIC dataset

The accuracy measures the percentage of correct predictions made by the model on the training data. The validation accuracy measures the same but for the validation data. The loss and validation loss measure the error of the model on the training set and validation set, respectively. These metrics are used to describe the ability of the model to classify images. Here, a high accuracy and a low loss indicate that the model predicts correctly very often, and when it is incorrect, the error is usually small. Because the loss is very close to the validation loss and the accuracy is very close to the validation accuracy, we can conclude that the model is fitting the data well and there is no overfitting or underfitting. When training, if we achieve a result that is lower than desired, we change hyperparameters and observe the effect to determine what hyperparameter tuning will maximize the model's performance.

Another part of the testing process for this project is comparing training results for our on-ground environment and our cloud environment. By comparing these results, we can evaluate the cost-effectiveness of training on the cloud. Figure 6 demonstrates an example of the difference between the training speed of the two environments when training the same model with the same dataset and hyperparameters. The AWS training was faster than the on-ground training while achieving similar accuracy at a fraction of the cost. Throughout this project, the total cost to train on AWS Sagemaker was less than $100, while purchasing a GPU equivalent in power to those on the VM and AWS would cost upwards of $5,000 making AWS the more cost-effective option. From this, we conclude that the cloud is a powerful and cost-effective tool for machine learning and that many large-scale machine learning projects can save money by training in the cloud.

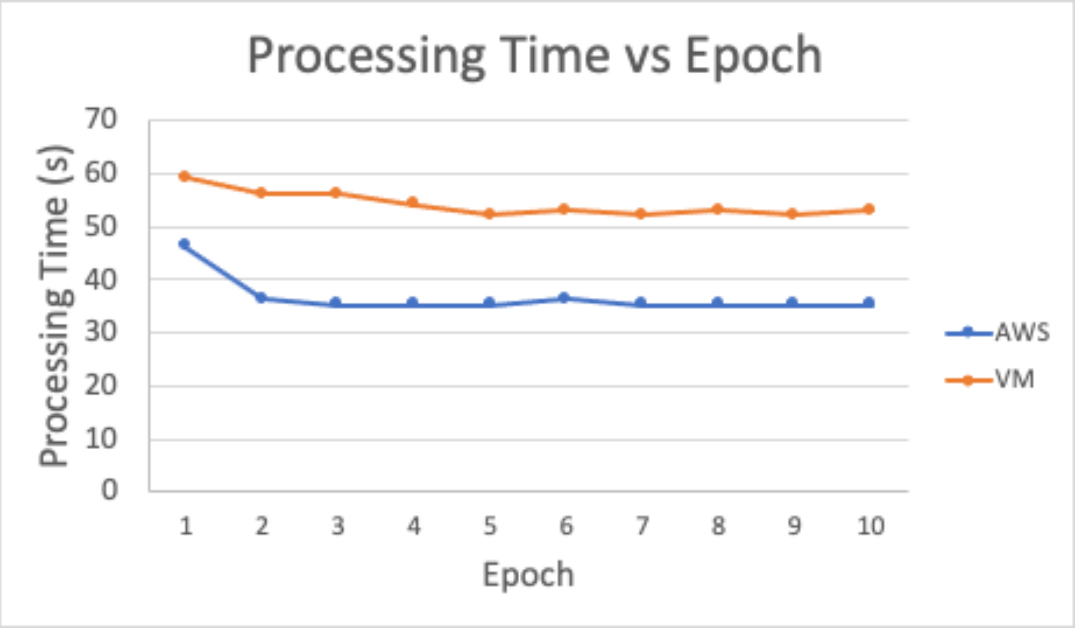


Figure 6: Variation in processing time as a function of epoch between AWS and VM environments

The user interface also presents opportunities for testing by utilizing data outside the existing training dataset. As noted previously, the model incorporates a portion of the existing training data for its official validation metric, but it does not use data that it has never seen before. The user interface, in this case, allows for the model to be rigorously tested against data that is otherwise completely foreign to the original dataset that was used to both train and validate the model. Therefore, the stated numerical accuracy of the model can have greater significance when tested against completely new images as each new user continuously tests against the existing model, and allows the developer to update the model in the future as new data comes in. Figure 7 illustrates the use of the built UI with an underlying model making an accurate prediction of the malignancy of a test image that it has never interacted with (either as a part of the training or validation set).

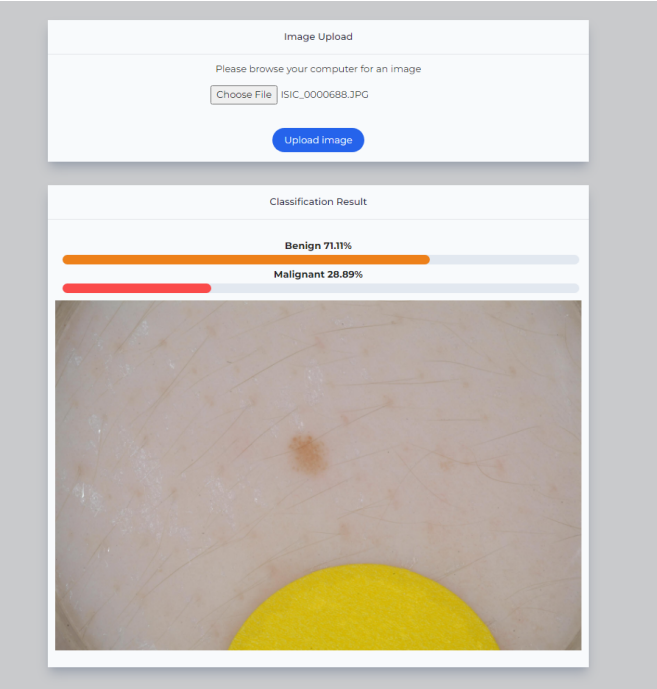


Figure 7: User Interface generating an accurate malignancy prediction of a sample skin lesion image

Work Context

Throughout our research for this project, we found examples of machine-learning models being used as diagnostic tools for detecting and diagnosing skin cancer. Many studies have found that the use of AI tools in diagnostic settings significantly increases the accuracy of diagnosing skin lesions. With this project, the team further demonstrates that an image classification algorithm can be used to diagnose skin cancer. In addition, the project demonstrates the benefits of using the cloud for training machine learning models. The team's advisor and client, Dr. Gaffar, has previously worked on a project with Mayo Clinic that trains AI models as diagnostic tools for skin cancer using large datasets. This project, however, has never used on-cloud training, only on-ground training. The team's documentation and results can be used by the Mayo Clinic project and other large-scale machine learning projects to assist in transitioning from on-ground training to on-cloud training, which has been demonstrated to be extremely cost-effective. By avoiding purchasing expensive GPUs for on-ground training, large-scale machine learning projects can increase their training capabilities and improve the performance of their models.

Appendix I - Operation Manual

User Interface

To obtain predictions for an image using the user interface, please follow the steps below:

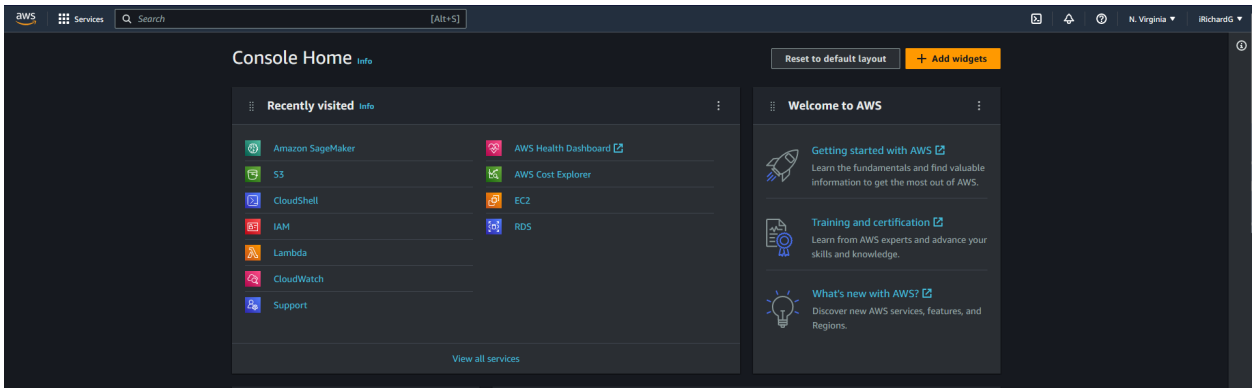
1. Click on the "Choose File" button to select a picture from your local computer.
2. Once you have selected a picture, click on the "Upload Image" button to submit it to the model for prediction.
3. Wait while the application processes your request for prediction. This may take a few moments.
4. Once the prediction is ready, two bars will appear displaying the probability of the image being Malignant and Benign, respectively.
5. If you wish to predict another image, repeat the above steps starting from step 1.

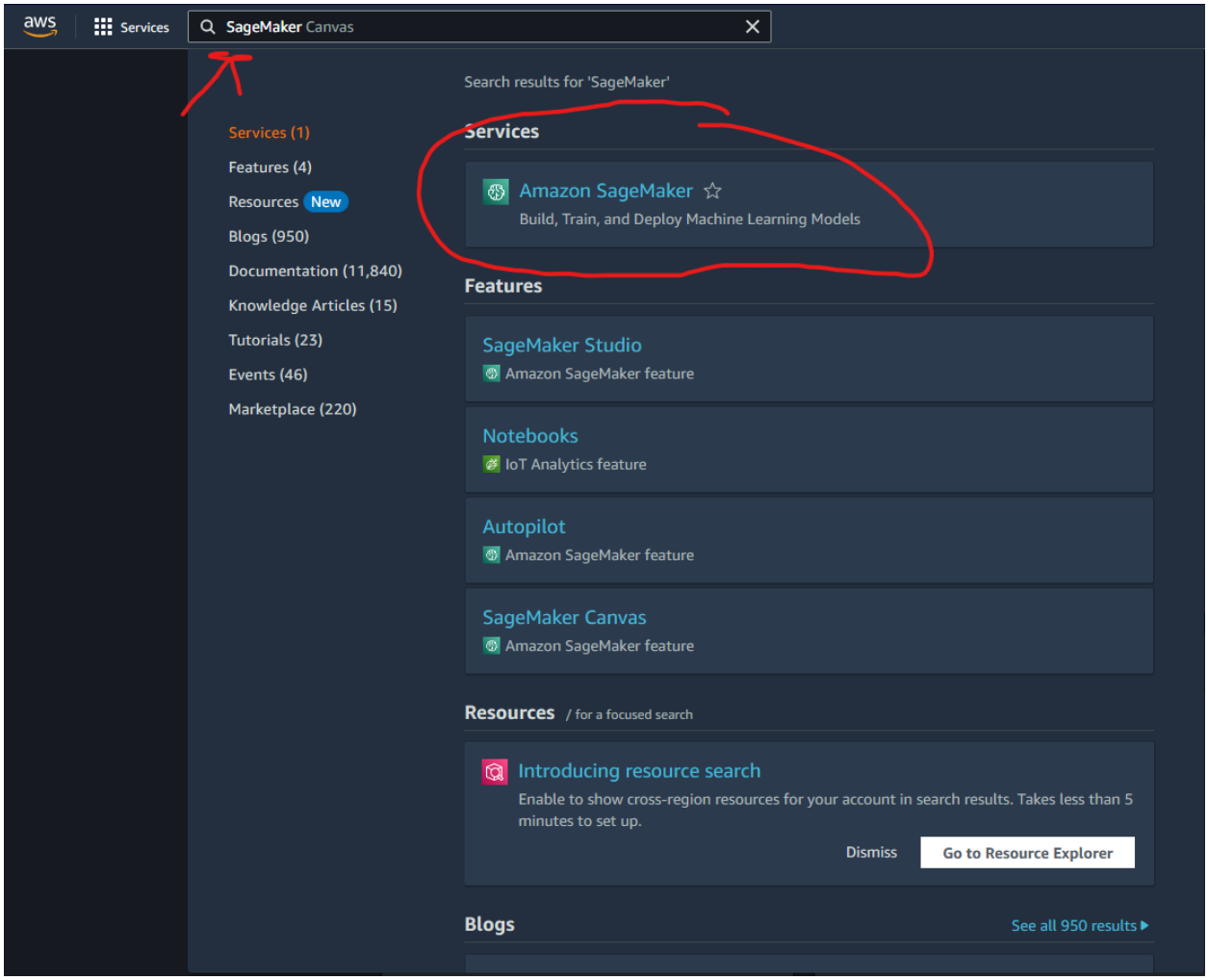
Virtual Machine Training Environment Setup

1) Install conda and TensorFlow
2) In the same directory, store image files and model.py file
3) To train the model, run **python3 model.py**

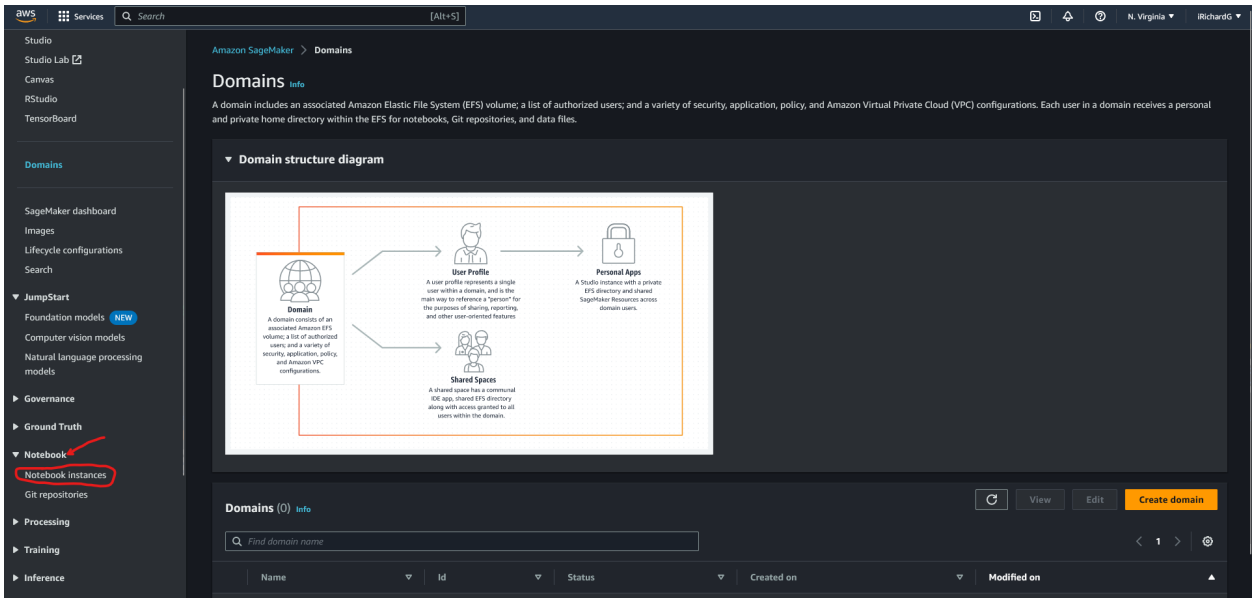
AWS Training Environment Setup

1) Sign into the Amazon Sagemaker Console

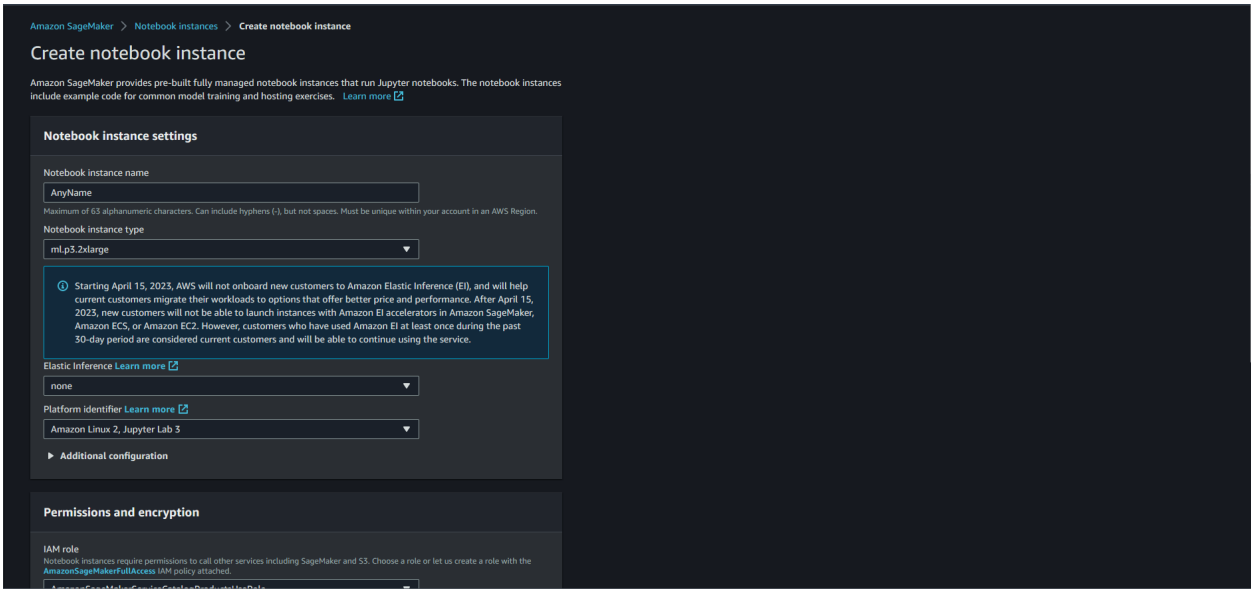



aws
Services
SageMaker Canvas
Search results for 'SageMaker'
Services (1)
Features (4)
Resources New
Blogs (950)
Documentation (11,840)
Knowledge Articles (15)
Tutorials (23)
Events (46)
Marketplace (220)
Services
Amazon SageMaker
Build, Train, and Deploy Machine Learning Models
Features
SageMaker Studio
Amazon SageMaker feature
Notebooks
IoT Analytics feature
Autopilot
Amazon SageMaker feature
SageMaker Canvas
Amazon SageMaker feature
Resources / for a focused search
Introducing resource search
Enable to show cross-region resources for your account in search results. Takes less than 5 minutes to set up.
Dismiss
Go to Resource Explorer
Blogs
See all 950 results

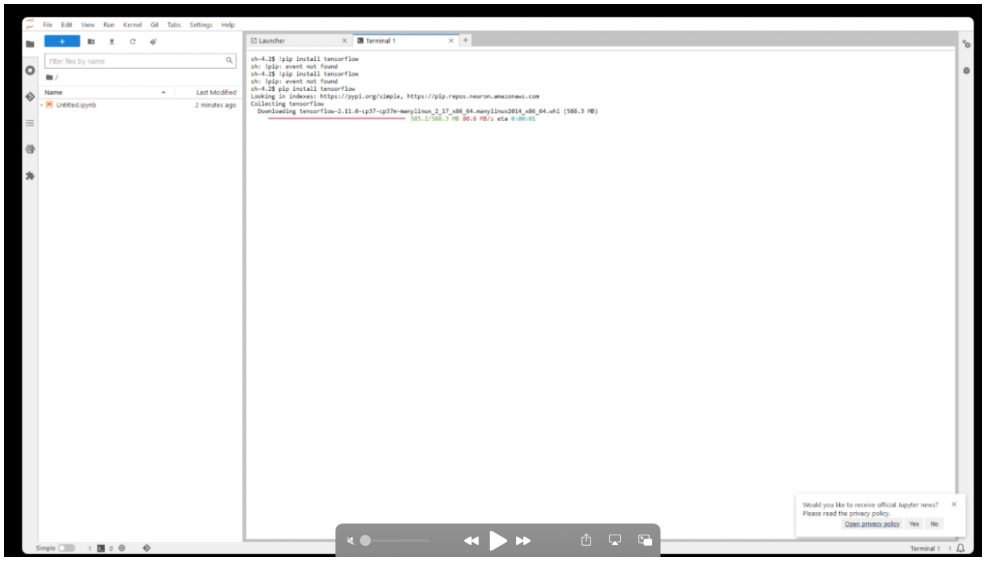
2) Create a new Notebook Instance



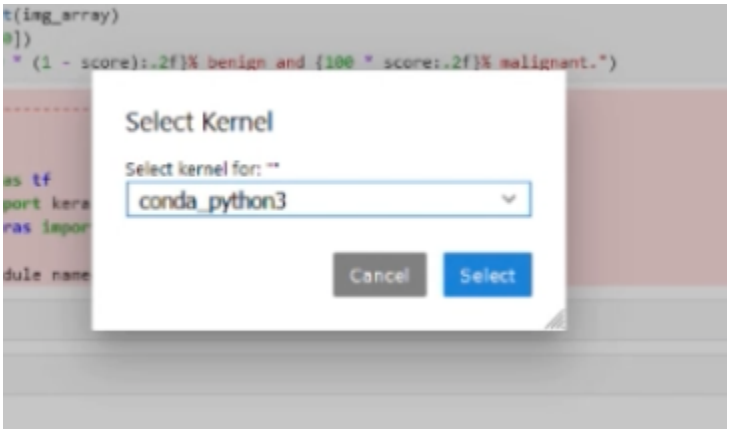
a) Specify an instance name
b) Specify an instance type(We used a ml.p3.2xlarge) a Single NVIDIA V100 GPU as referenced above.



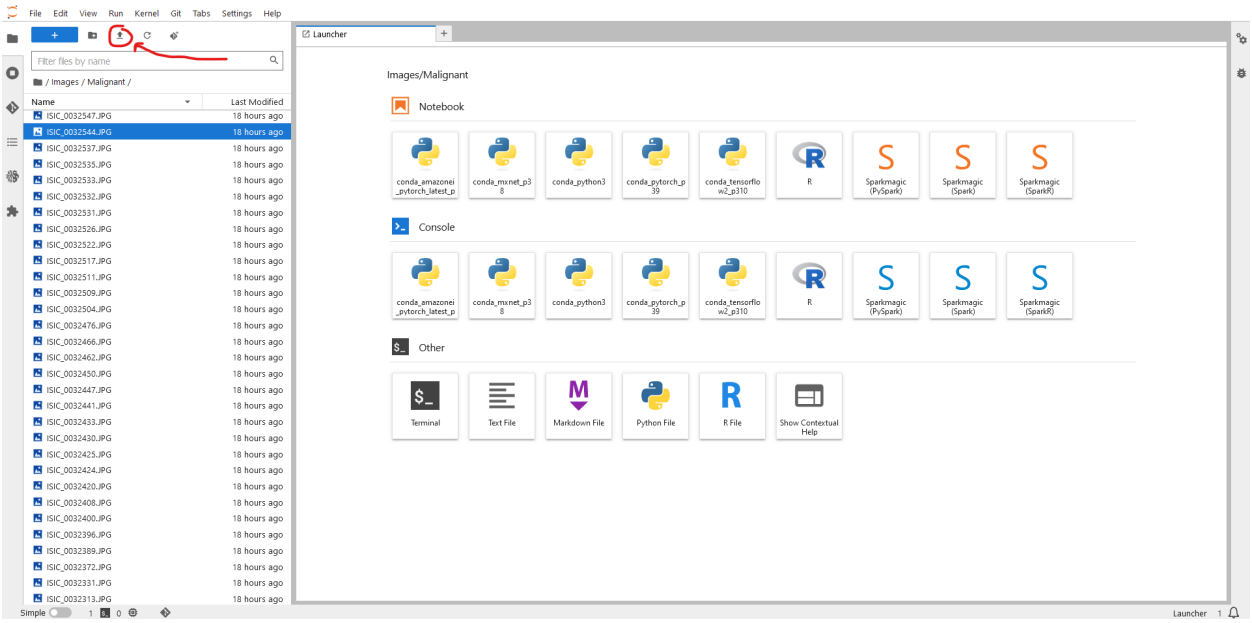
3) Open terminal and Ensure TensorFlow is installed



4) Open Jupyter and ensure that ‘conda_python3’ is the selected kernel



5) Download image data into the Sagemaker instance



6) Import the local/VM 'model.py' source (Note: ensure the image data format is 'channels_last' as opposed to 'channels_first')
7) Open Terminal
8) Run the following command to enter the conda environment for TensorFlow:
   source activate tensorflow2_p310
9) Run the following command to train the model:
   python3 model.py

Appendix II - Alternative Designs

In 491, the team considered developing an image classification model to classify livestock animals as healthy or sick. The proposed model would take audio files of sounds made by animals, transform them into spectrogram (image) representations, and then analyze the image representation to determine what patterns exist for healthy and sick animals. We started creating and training the model. We used binary classification to create the model so that if our project goes wrong, then we do not have to create a new model.

Our plan was that if the audio does not work, we will train the model with different datasets. We presented this model during lightning talks and our presentation for 491. As part of the research process, the team looked into existing binary image classification examples. At the beginning of 492, the team received a dataset of livestock animal sounds, but it was a small dataset of only a few hundred samples with very low audio quality. The audio files were not labeled which made them unusable as a dataset. So, the team decided that this dataset was not sufficient enough to train a capable model, so instead, the team began working towards developing and training a skin lesion classification model.

Appendix III - Other Considerations

We used following specifications:-

VM Specifications

Linux sdmay23-05.ece.iastate.edu 5.15.0-52-generic #58-Ubuntu SMP x86_64 x86_64 x86_64 GNU/Linux

CPU Info:

Product: Intel(R) Xeon(R) Gold 6140 CPU @ 2.30 GHz
Architecture: x86_64
Cores: 8
Max Memory Size: 768 GB
Memory Type: DDR4-2666
Maximum Memory Speed: 2666 MHz

GPU info:

Product: TU102GL [Quadro RTX 6000/8000]
Width: 64 bits
Clock: 66 MHz

AWS Specifications

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instance types comprise varying combinations of CPU, memory, storage, and networking capacity, giving us the flexibility to choose the appropriate mix of resources.
For this model, we used a high-frequency 3.3 GHz Intel Xeon Scalable processor.
P3 instances use customized Intel Xeon E5-2686v4 processors running at up to 2.7 GHz. They are available in three sizes (all VPC-only and EBS-only).

Model	NVIDIA Tesla V100 GPUs	GPU Memory	NVIDIA NVLink	vCPUs	Main Memory	Network Bandwidth	EBS Bandwidth
p3.2xlarge	1	16 GiB	n/a	8	61 GiB	Up to 10 Gbps	1.5 Gbps

NVIDIA Tesla V100 GPUs The First Tensor Core GPU

Packed with 5,120 CUDA cores and another 640 Tensor cores and can deliver up to 125 TFLOPS.

The P3 instances are designed to handle compute-intensive machine learning, deep learning, and computational heavy workloads.

Appendix IV - Code

Binary image classification model code:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt

image_size = (180, 180)
batch_size = 16

train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
    "AllImages",
    validation_split=0.2,
    subset="both",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)

data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]
)

augmented_train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y))

# Apply `data_augmentation` to the training images.
train_ds = train_ds.map(
    lambda img, label: (data_augmentation(img), label),
    num_parallel_calls=tf.data.AUTOTUNE,
)
# Prefetching samples in GPU memory helps maximize GPU utilization.
train_ds = train_ds.prefetch(tf.data.AUTOTUNE)
val_ds = val_ds.prefetch(tf.data.AUTOTUNE)

def make_model(input_shape, num_classes):
```

```
    inputs = keras.Input(shape=input_shape)

    # Entry block
    x = layers.Rescaling(1.0 / 255)(inputs)
    x = layers.Conv2D(128, 3, strides=2, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    for size in [256, 512, 728]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(size, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    x = layers.SeparableConv2D(1024, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.GlobalAveragePooling2D()(x)
    if num_classes == 2:
        activation = "sigmoid"
        units = 1
    else:
        activation = "softmax"
        units = num_classes
```

```
    x = layers.Dropout(0.5)(x)
    outputs = layers.Dense(units, activation=activation)(x)
    return keras.Model(inputs, outputs)


model = make_model(input_shape=image_size + (3,), num_classes=2)

epochs = 10

callbacks = [
    keras.callbacks.ModelCheckpoint("save_at_{epoch}.keras"),
]
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_ds,
    epochs=epochs,
    callbacks=callbacks,
    validation_data=val_ds,
)
```