S E 492
Team sdmay23-05
April 30, 2023
On-Ground vs On-Cloud AI Training Report


**Introduction** 1
**Dataset** 1
**VM Specifications** 2
**AWS Specifications** 2
**Comparison Specs** 3
**AI Model** 4
**Model Hyperparameters** 4
**On-Ground Training on Full Dataset** 5
**Process to Train Model On-Ground** 5
**Process to Train Model in AWS Sagemaker** 5
**Comparison 1: VM GPU vs AWS CPU** 10
- Training Results on VM 10
- Training Results on AWS CPU 10

**Comparison 2: VM GPU vs AWS GPU** 11
- Training Results on AWS GPU 11
- Training Results on VM GPU 11

**Cost** 12
**Comparison/Analysis** 12
**Conclusion** 12
**Appendix** 13


## Introduction

For this part of the project, we developed a binary image classification AI model to classify skin lesions from the publicly available ISIC dataset as benign or malignant. We then trained the model in a local environment and a cloud environment to compare the performance and results. The goals are to document the process of training an AI model in the cloud and to analyze the costs and benefits of training in the cloud compared to training locally.

## 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.[1] The number 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.

## 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
CUDA Parallel-Processing Cores: 4,608
NVIDIA Tensor Cores: 576
NVIDIA RT Cores: 72
GPU Memory: 24 GB GDDR6

## 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).

---

[1] https://www.isic-archive.com/#!/onlyHeaderTop/gallery?filter=%5B%22benign_malignant%7Cbenign%22%2C%22benign_malignant%7Cmalignant%22%5D

| 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 |

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

**Why use this for machine learning on AWS?**

NVIDIA Tesla V100 GPUs The First Tensor Core GPU

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

## Comparison Specs

Theoretical Performance

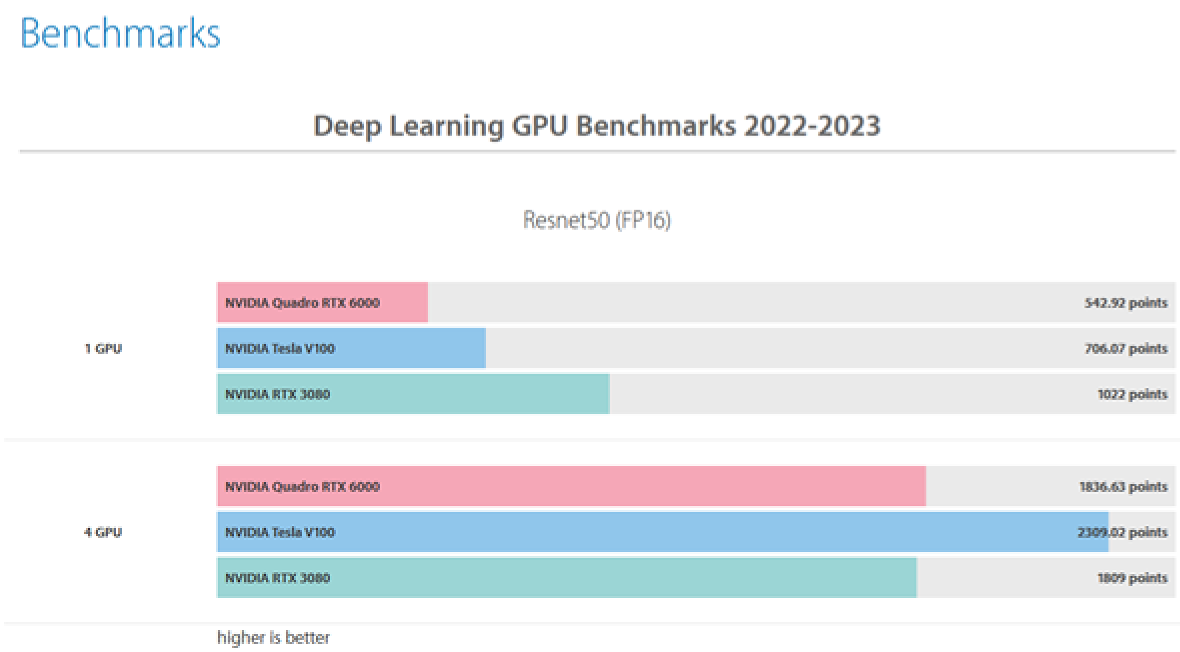
| | NVIDIA Quadro RTX 6000 | NVIDIA Tesla V100 | NVIDIA RTX 3080 |
|---|---|---|---|
| **FP16 (half) performance** | 32.62 TFLOPS | 28.26 TFLOPS | 59.54 TFLOPS |
| **FP32 (float) performance** | 16.31 TFLOPS | 14.13 TFLOPS | 29.77 TFLOPS |
| **FP64 (double) performance** | 509.8 GFLOPS | 7066 GFLOPS | 930.2 GFLOPS |
| **Pixel Rate** | 169.9 GPixel/s | 176.6 GPixel/s | 150.5 GPixel/s |
| **Texture Rate** | 509.8 GTexel/s | 441.6 GTexel/s | 465.1 GTexel/s |

Clock Speeds

| | NVIDIA Quadro RTX 6000 | NVIDIA Tesla V100 | NVIDIA RTX 3080 |
|---|---|---|---|
| **Boost Clock** | 1770 MHz | 1380 MHz | 1710 MHz |
| **GPU Clock** | 1440 MHz | 1230 MHz | 1440 MHz |
| **Memory Clock** | 14000 MHz | 1752 MHz | 19000 MHz |

Graphics Card

| | NVIDIA Quadro RTX 6000 | NVIDIA Tesla V100 | NVIDIA RTX 3080 |
|---|---|---|---|
| **Bus Interface** | PCIe 3.0 x16 | PCIe 3.0 x16 | PCIe 4.0 x16 |
| **Generation** | Quadro RTX | Tesla (Vxx) | GeForce 30 |




## AI Model

The model was adapted from an existing image classifier provided as a Keras tutorial[2]. 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.

## 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.

[2] https://keras.io/examples/vision/image_classification_from_scratch

- 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 59,676 images, we can already achieve an accuracy above 92% after only 3-5 epochs.
- Batch size: The batch size dictates the number of samples encountered in training before the model is updated. We experimented with batch sizes of 128 and 32 before finding that a batch size of 16 gave us the best results.

## On-Ground Training on Full Dataset

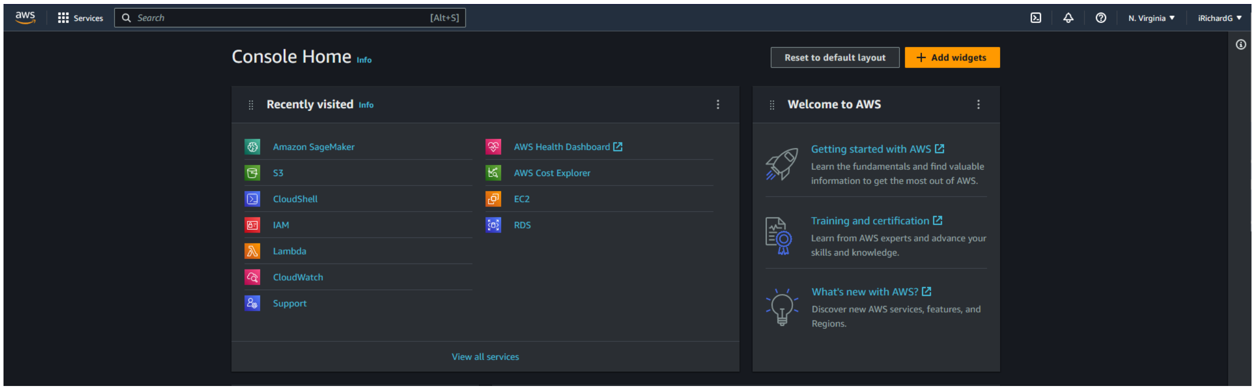
```
1669/1669 [==============================] - 772s 458ms/step - loss: 0.2563 - accuracy: 0.9001 - val_loss: 0.2555 - val_accuracy: 0.9022
Epoch 2/10
1669/1669 [==============================] - 783s 468ms/step - loss: 0.2284 - accuracy: 0.9066 - val_loss: 0.2214 - val_accuracy: 0.9120
Epoch 3/10
1669/1669 [==============================] - 813s 485ms/step - loss: 0.2210 - accuracy: 0.9092 - val_loss: 0.2138 - val_accuracy: 0.9163
Epoch 4/10
1669/1669 [==============================] - 805s 480ms/step - loss: 0.2176 - accuracy: 0.9106 - val_loss: 0.2415 - val_accuracy: 0.9069
Epoch 5/10
1669/1669 [==============================] - 809s 483ms/step - loss: 0.2109 - accuracy: 0.9137 - val_loss: 0.2212 - val_accuracy: 0.9157
Epoch 6/10
1669/1669 [==============================] - 812s 485ms/step - loss: 0.2086 - accuracy: 0.9146 - val_loss: 0.2118 - val_accuracy: 0.9170
Epoch 7/10
1669/1669 [==============================] - 820s 489ms/step - loss: 0.2058 - accuracy: 0.9158 - val_loss: 0.2190 - val_accuracy: 0.9141
Epoch 8/10
1669/1669 [==============================] - 801s 478ms/step - loss: 0.2013 - accuracy: 0.9173 - val_loss: 0.2069 - val_accuracy: 0.9207
Epoch 9/10
1669/1669 [==============================] - 822s 490ms/step - loss: 0.1995 - accuracy: 0.9173 - val_loss: 0.1976 - val_accuracy: 0.9197
Epoch 10/10
1669/1669 [==============================] - 800s 478ms/step - loss: 0.1994 - accuracy: 0.9184 - val_loss: 0.1997 - val_accuracy: 0.9177
```

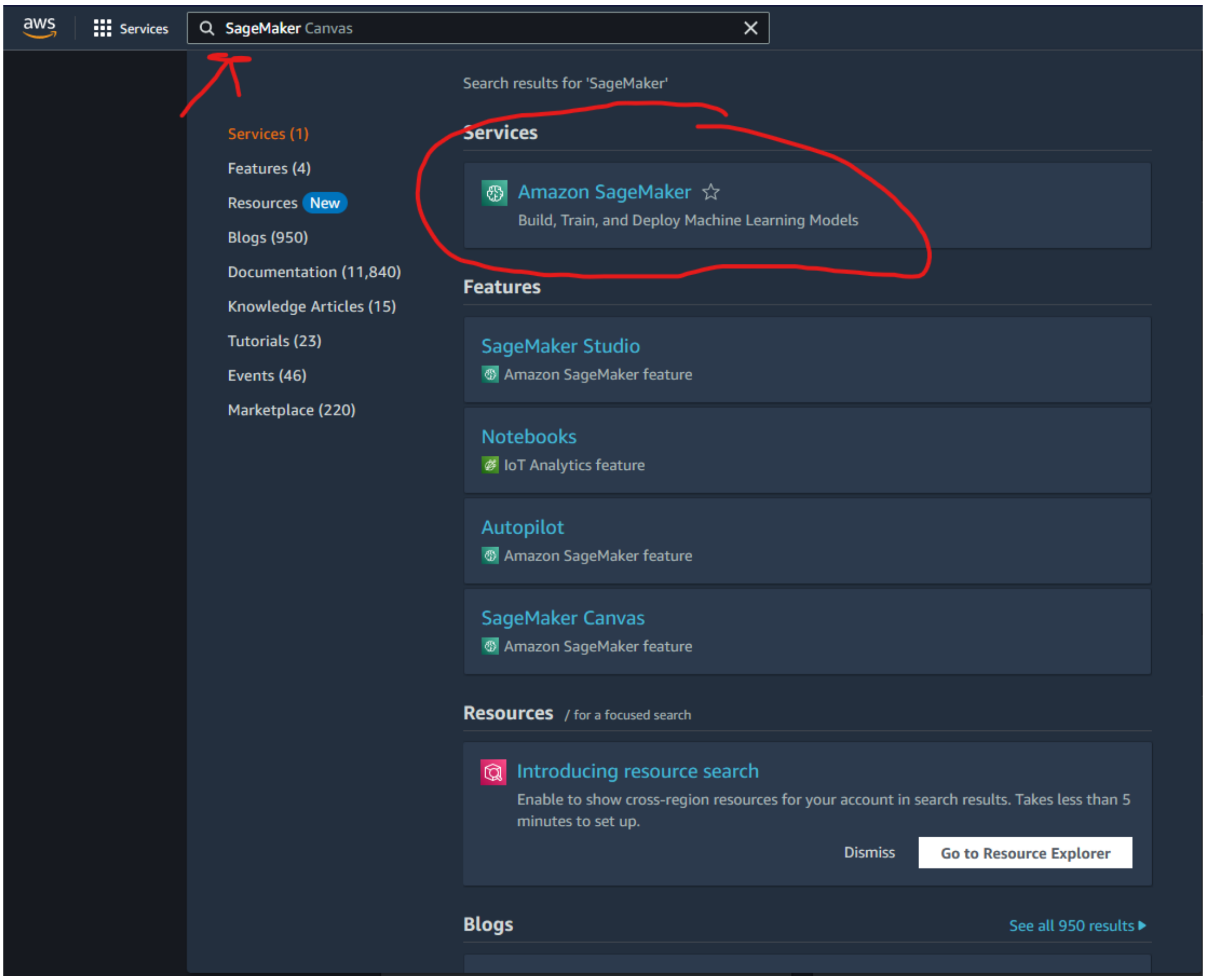
## Process to Train Model On-Ground

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**

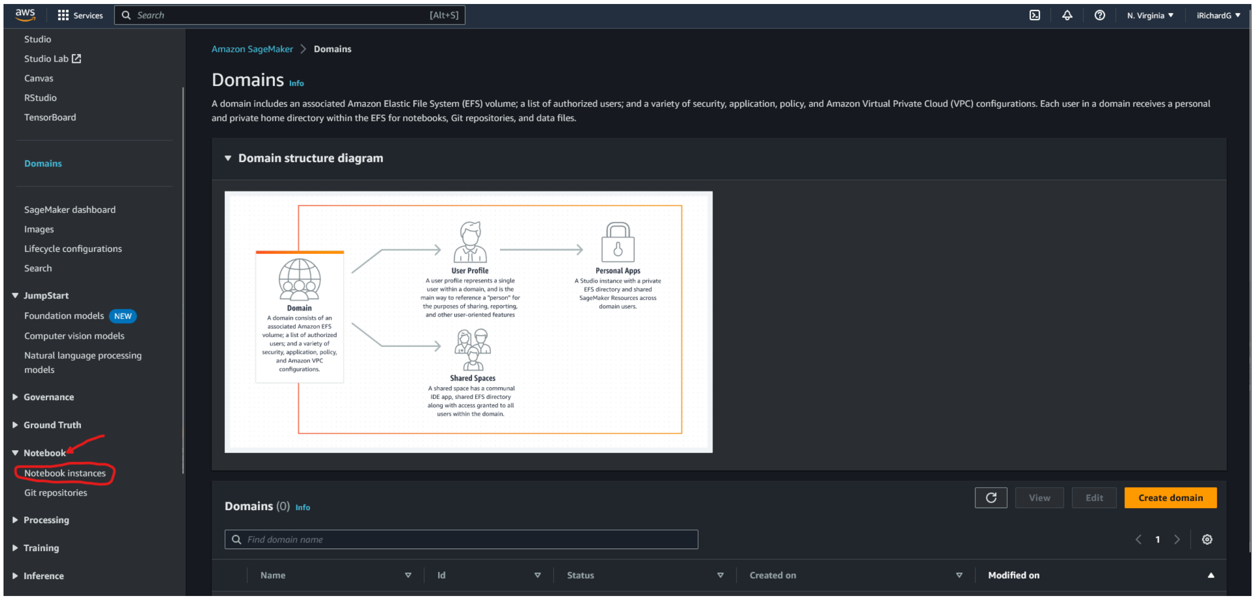
## Process to Train Model in AWS Sagemaker

1) Sign into the Amazon Sagemaker Console

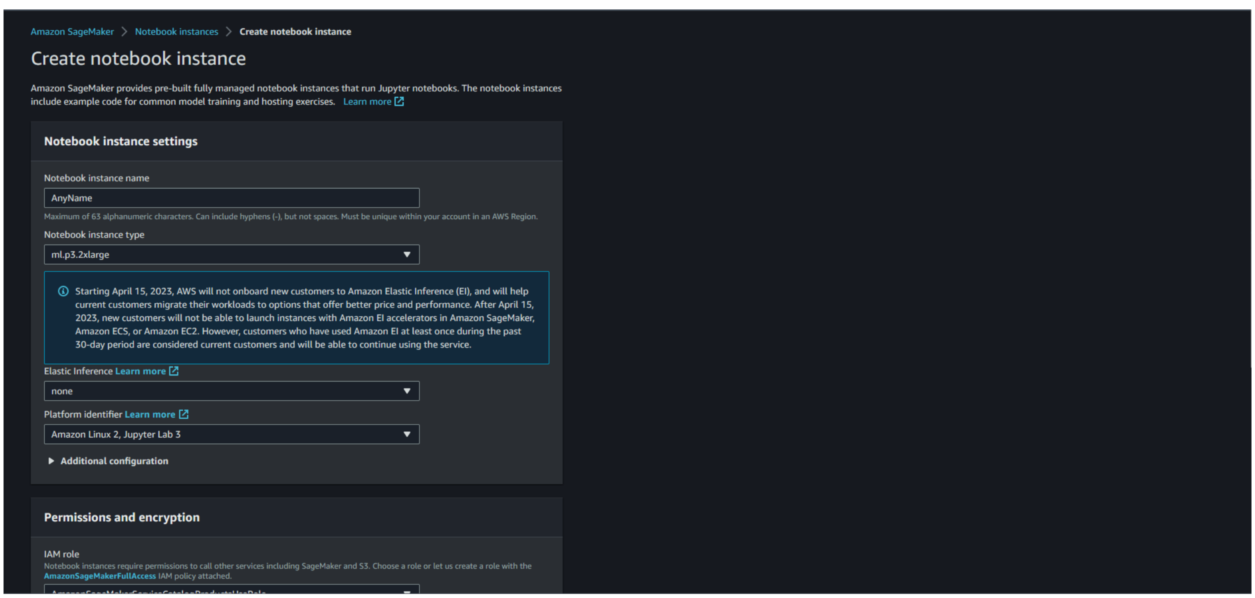


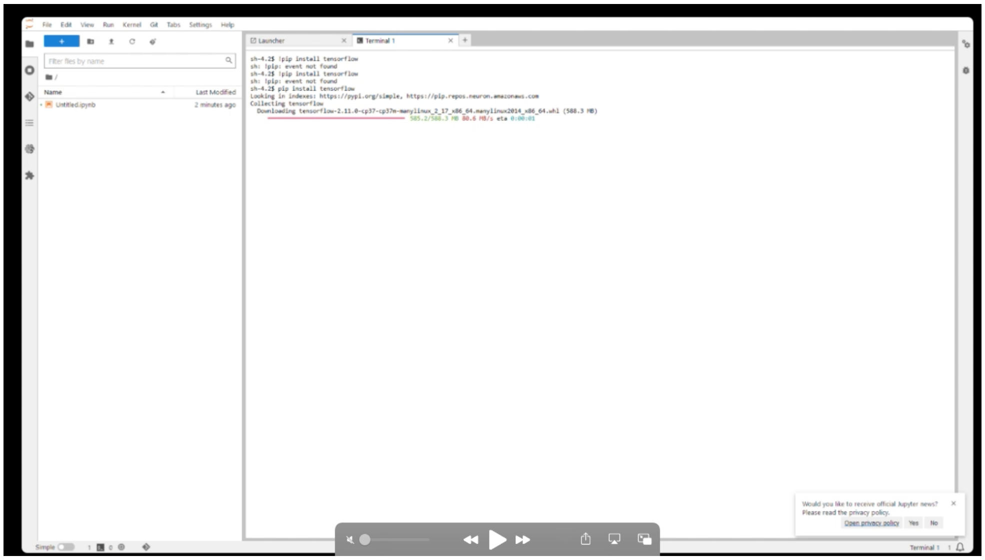

2) Create a new Notebook Instance



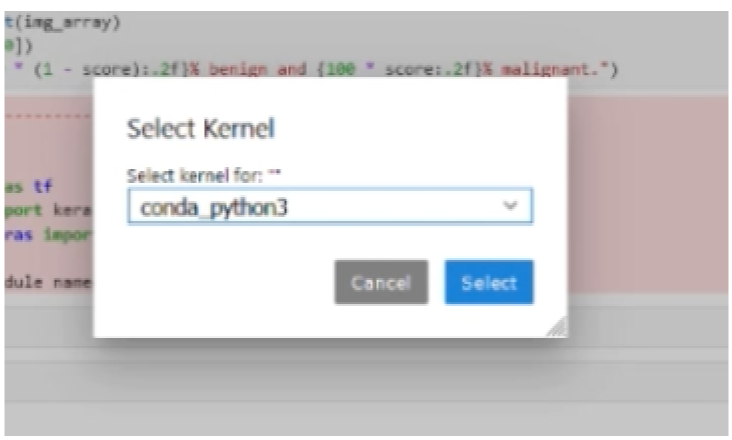

a) Specify an instance name
b) Specify an instance type(We used a ml.p3.2xlarge) a Single NVIDIA V100 GPU as referenced above.



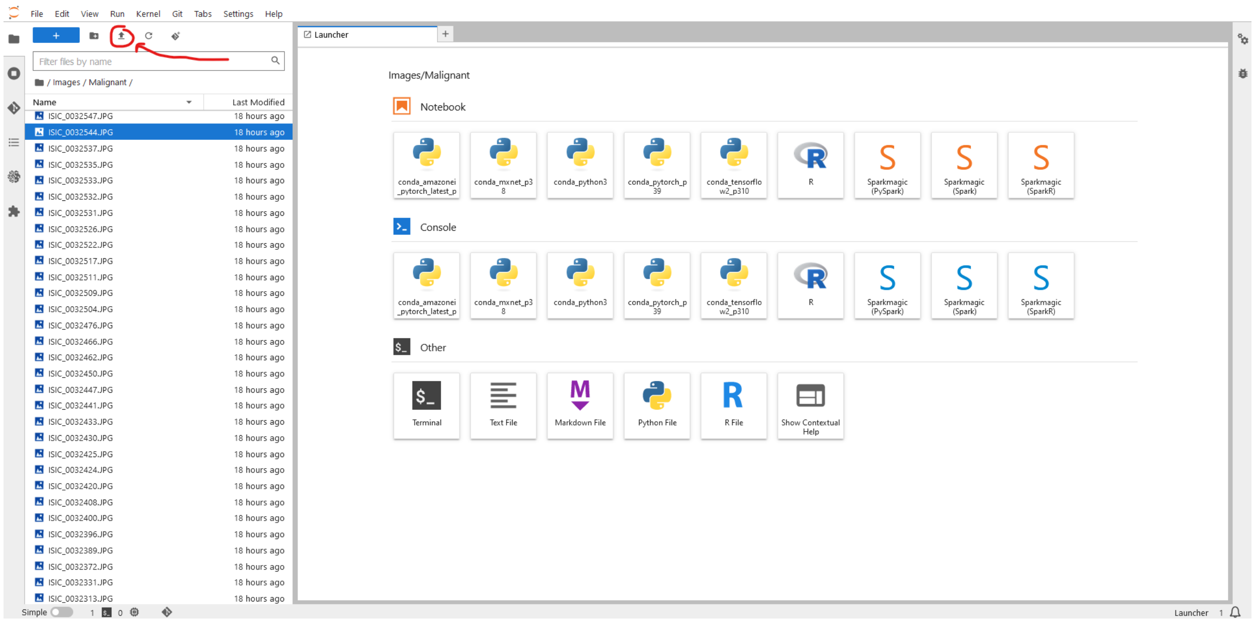
3) Open terminal and Ensure TensorFlow is installed



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**

**Comparison 1: VM GPU vs AWS CPU**

Dataset: 200 images

Epochs: 10

## Training Results on VM

```
Epoch 1/10
2023-03-02 22:47:08.230740: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8100
2023-03-02 22:47:09.023929: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-03-02 22:47:09.024520: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-03-02 22:47:09.024547: W tensorflow/stream_executor/gpu/asm_compiler.cc:80] Couldn't get ptxas version string: INTERNAL: Couldn't invoke ptxas --version
2023-03-02 22:47:09.025157: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-03-02 22:47:09.025222: W tensorflow/stream_executor/gpu/redzone_allocator.cc:314] INTERNAL: Failed to launch ptxas
Relying on driver to perform ptx compilation.
Modify $PATH to customize ptxas location.
This message will be only logged once.
10/10 [==============================] - 7s 168ms/step - loss: 0.7456 - accuracy: 0.5813 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 2/10
10/10 [==============================] - 2s 118ms/step - loss: 0.6064 - accuracy: 0.6687 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 3/10
10/10 [==============================] - 2s 118ms/step - loss: 0.4836 - accuracy: 0.7750 - val_loss: 0.6927 - val_accuracy: 0.5000
Epoch 4/10
10/10 [==============================] - 2s 114ms/step - loss: 0.5836 - accuracy: 0.7188 - val_loss: 0.6929 - val_accuracy: 0.5000
Epoch 5/10
10/10 [==============================] - 2s 117ms/step - loss: 0.5608 - accuracy: 0.7188 - val_loss: 0.6932 - val_accuracy: 0.5000
Epoch 6/10
10/10 [==============================] - 2s 117ms/step - loss: 0.4588 - accuracy: 0.8125 - val_loss: 0.6955 - val_accuracy: 0.5000
Epoch 7/10
10/10 [==============================] - 2s 117ms/step - loss: 0.5154 - accuracy: 0.7563 - val_loss: 0.6939 - val_accuracy: 0.5000
Epoch 8/10
10/10 [==============================] - 2s 117ms/step - loss: 0.5530 - accuracy: 0.7563 - val_loss: 0.6952 - val_accuracy: 0.5000
Epoch 9/10
10/10 [==============================] - 2s 115ms/step - loss: 0.5393 - accuracy: 0.6875 - val_loss: 0.6928 - val_accuracy: 0.5000
Epoch 10/10
10/10 [==============================] - 2s 116ms/step - loss: 0.4776 - accuracy: 0.7500 - val_loss: 0.6984 - val_accuracy: 0.5000
```

## Training Results on AWS CPU

```
Epoch 1/10
10/10 [==============================] - 132s 13s/step - loss: 0.8234 - accuracy: 0.6125 - val_loss: 0.6926 - val_accuracy: 0.5000
Epoch 2/10
10/10 [==============================] - 264s 28s/step - loss: 0.6178 - accuracy: 0.6938 - val_loss: 0.6926 - val_accuracy: 0.5000
Epoch 3/10
10/10 [==============================] - 235s 25s/step - loss: 0.5250 - accuracy: 0.7125 - val_loss: 0.6920 - val_accuracy: 0.5000
Epoch 4/10
10/10 [==============================] - 196s 21s/step - loss: 0.4914 - accuracy: 0.7500 - val_loss: 0.6914 - val_accuracy: 0.5000
Epoch 5/10
10/10 [==============================] - 198s 17s/step - loss: 0.4970 - accuracy: 0.8062 - val_loss: 0.6918 - val_accuracy: 0.5000
Epoch 6/10
10/10 [==============================] - 206s 18s/step - loss: 0.4866 - accuracy: 0.7812 - val_loss: 0.6923 - val_accuracy: 0.5000
Epoch 7/10
10/10 [==============================] - 281s 29s/step - loss: 0.5685 - accuracy: 0.7500 - val_loss: 0.6912 - val_accuracy: 0.5000
Epoch 8/10
10/10 [==============================] - 245s 26s/step - loss: 0.4168 - accuracy: 0.8313 - val_loss: 0.6907 - val_accuracy: 0.5000
Epoch 9/10
10/10 [==============================] - 184s 20s/step - loss: 0.5282 - accuracy: 0.7688 - val_loss: 0.6900 - val_accuracy: 0.5000
Epoch 10/10
10/10 [==============================] - 196s 16s/step - loss: 0.4877 - accuracy: 0.7750 - val_loss: 0.6891 - val_accuracy: 0.5000
1/1 [==============================] - 2s 2s/step
```

Results: The AWS training was much slower, largely due to the superior computing power of the VM GPU compared to the AWS CPU. In order to achieve better performance on AWS, we will need to upgrade the computing resources.

# Comparison 2: VM GPU vs AWS GPU

Dataset: 10,000 images

Epochs: 10

## Training Results on AWS GPU

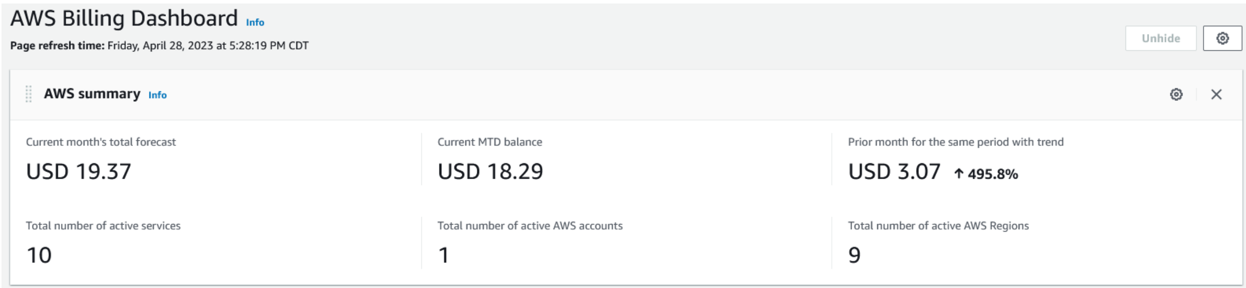
```
Terminal 1    model.py

WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
WARNING:tensorflow:From /home/ec2-user/anaconda3/envs/tensorflow2_p310/lib/python3.10/site-packages/tensorflow/python/autograph/pyct/static_analysis/liveness.py:83: Analyzer.lamba_check (from tensorf
low.python.autograph.pyct.static_analysis.liveness) is deprecated and will be removed after 2023-09-23.
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they are used, or at least in the same block. https://github.com/tensorflow/tensorflow/issues/56089
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting RngReadAndSkip cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting Bitcast cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting StatelessRandomUniformV2 cause there is no registered converter for this op.
WARNING:tensorflow:Using a while_loop for converting ImageProjectiveTransformV3 cause there is no registered converter for this op.
Epoch 1/10
151/151 [==============================] - 46s 222ms/step - loss: 0.6646 - accuracy: 0.6244 - val_loss: 0.6932 - val_accuracy: 0.4043
Epoch 2/10
151/151 [==============================] - 36s 229ms/step - loss: 0.5993 - accuracy: 0.6859 - val_loss: 0.6944 - val_accuracy: 0.3727
Epoch 3/10
151/151 [==============================] - 35s 224ms/step - loss: 0.5547 - accuracy: 0.7371 - val_loss: 0.6925 - val_accuracy: 0.4226
Epoch 4/10
151/151 [==============================] - 35s 224ms/step - loss: 0.5206 - accuracy: 0.7616 - val_loss: 0.6306 - val_accuracy: 0.7704
Epoch 5/10
151/151 [==============================] - 35s 224ms/step - loss: 0.5110 - accuracy: 0.7671 - val_loss: 0.4956 - val_accuracy: 0.8253
Epoch 6/10
151/151 [==============================] - 36s 226ms/step - loss: 0.4691 - accuracy: 0.7962 - val_loss: 0.4152 - val_accuracy: 0.8236
Epoch 7/10
151/151 [==============================] - 35s 224ms/step - loss: 0.4559 - accuracy: 0.8007 - val_loss: 0.3890 - val_accuracy: 0.8403
Epoch 8/10
151/151 [==============================] - 35s 225ms/step - loss: 0.4364 - accuracy: 0.8141 - val_loss: 0.4315 - val_accuracy: 0.8037
Epoch 9/10
151/151 [==============================] - 35s 223ms/step - loss: 0.4163 - accuracy: 0.8215 - val_loss: 0.3638 - val_accuracy: 0.8469
Epoch 10/10
151/151 [==============================] - 35s 223ms/step - loss: 0.4005 - accuracy: 0.8415 - val_loss: 0.3318 - val_accuracy: 0.8602
Saved model to disk
1/1 [==============================] - 0s 410ms/step
This image is 0.82% benign and 99.18% malignant.
(tensorflow2_p310) sh-4.2$

Terminal 1   1
```

## Training Results on VM GPU

```
Epoch 1/10
2023-04-29 18:12:39.256354: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8100
2023-04-29 18:12:40.341635: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-04-29 18:12:40.342240: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-04-29 18:12:40.342266: W tensorflow/stream_executor/gpu/asm_compiler.cc:80] Couldn't get ptxas version string: INTERNAL: Couldn't invoke
2023-04-29 18:12:40.342849: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process: No such file or directory
2023-04-29 18:12:40.342917: W tensorflow/stream_executor/gpu/redzone_allocator.cc:314] INTERNAL: Failed to launch ptxas
Relying on driver to perform ptx compilation.
Modify $PATH to customize ptxas location.
This message will be only logged once.
384/384 [==============================] - 59s 137ms/step - loss: 0.2776 - accuracy: 0.8900 - val_loss: 0.9841 - val_accuracy: 0.7467
Epoch 2/10
384/384 [==============================] - 56s 145ms/step - loss: 0.2009 - accuracy: 0.9168 - val_loss: 0.3331 - val_accuracy: 0.8570
Epoch 3/10
384/384 [==============================] - 56s 144ms/step - loss: 0.1688 - accuracy: 0.9312 - val_loss: 0.1390 - val_accuracy: 0.9413
Epoch 4/10
384/384 [==============================] - 54s 138ms/step - loss: 0.1652 - accuracy: 0.9318 - val_loss: 0.2398 - val_accuracy: 0.9008
Epoch 5/10
384/384 [==============================] - 52s 134ms/step - loss: 0.1560 - accuracy: 0.9357 - val_loss: 0.1687 - val_accuracy: 0.9426
Epoch 6/10
384/384 [==============================] - 53s 135ms/step - loss: 0.1486 - accuracy: 0.9411 - val_loss: 0.2111 - val_accuracy: 0.9099
Epoch 7/10
384/384 [==============================] - 52s 133ms/step - loss: 0.1499 - accuracy: 0.9390 - val_loss: 0.1765 - val_accuracy: 0.9373
Epoch 8/10
384/384 [==============================] - 53s 135ms/step - loss: 0.1377 - accuracy: 0.9392 - val_loss: 0.1845 - val_accuracy: 0.9132
Epoch 9/10
384/384 [==============================] - 52s 134ms/step - loss: 0.1322 - accuracy: 0.9463 - val_loss: 0.1598 - val_accuracy: 0.9295
Epoch 10/10
384/384 [==============================] - 53s 135ms/step - loss: 0.1273 - accuracy: 0.9455 - val_loss: 0.1241 - val_accuracy: 0.9517
```

Results: In this case, the AWS instance trained faster than the VM environment with similar computing power.

## Cost




This cost was directly related to training 10,000 Images with SageMaker on a P3 Instance as described in the specifications section.

## Comparison/Analysis

Our experience with training a machine learning model on AWS using SageMaker has been incredibly rewarding. We successfully trained our model on the ISIC dataset, which is widely used by Mayo Clinic for skin cancer research. By leveraging AWS and cloud services, we were able to scale this research effectively and efficiently. Remarkably, we achieved similar results to those produced using the $5,000+ equipment at Iowa State ETG, but at a fraction of the cost, spending less than $100. This breakthrough demonstrates that the barriers to entry, such as cost and scalability, can be significantly reduced when harnessing the power of cloud-based services like AWS, making advanced machine learning more accessible and affordable for researchers and organizations worldwide.

## Conclusion

We discovered that both on-cloud and on-premises training approaches yielded similar results in our experiments. However, the on-cloud training proved to be superior due to its reduced reliance on local resources. This advantage enables researchers and organizations to access state-of-the-art computing power without the need for expensive hardware, making the cloud-based training approach more cost-effective, flexible, and scalable for machine learning applications.

# Appendix

Local model.py

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

image_size = (180, 180)
batch_size = 32

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

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(int(labels[i]))
        plt.axis("off")
plt.savefig('data.png')

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

plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
plt.savefig('augment.png')

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)
keras.utils.plot_model(model, show_shapes=True)

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,
)

img = keras.preprocessing.image.load_img(
    "Images/Malignant/ISIC_9998682.JPG", target_size=image_size
)
img_array = keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0)  # Create batch axis

predictions = model.predict(img_array)
score = float(predictions[0])
print(f"This image is {100 * (1 - score):.2f}% benign and {100 *
score:.2f}% malignant.")
```

AWS_Model.py

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

#tf.keras.backend.set_image_data_format("channels_last")

image_size = (180, 180)
batch_size = 16

train_ds, val_ds = tf.keras.utils.image_dataset_from_directory(
      "Images",
```

```
        validation_split=0.2,
        subset="both",
        seed=1337,
        image_size=image_size,
        batch_size=batch_size,
)

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
        for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(int(labels[i]))
        plt.axis("off")
plt.savefig('data.png')

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

plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
        for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
#plt.savefig('augment.png')

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)
#keras.utils.plot_model(model, show_shapes=True)

epochs = 10

callbacks = [
    keras.callbacks.ModelCheckpoint("save_at_{epoch}.keras"),
]
```

```
model.compile(
      optimizer=keras.optimizers.Adam(1e-5),
      loss="binary_crossentropy",

#loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=["accuracy"],
)
model.fit(
      train_ds,
      epochs=epochs,
      callbacks=callbacks,
      validation_data=val_ds,
)

# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
      json_file.write(model_json)
# serialize weights to HDF5
print("Saved model to disk")

model.save(
      "model.h5"
)

img = keras.preprocessing.image.load_img(
      "Images/Malignant/ISIC_0032547.JPG", target_size=image_size
)
img_array = keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0)  # Create batch axis

predictions = model.predict(img_array)
score = float(predictions[0])
# print(f"This image is {100 * (1 - score):.2f}% benign and {100 *
score:.2f}% malignant.")
```