Fachhochschul-Masterstudiengang
MOBILE COMPUTING
A-4232 Hagenberg, Austria

Project Paper

GEOGUESSR-AI

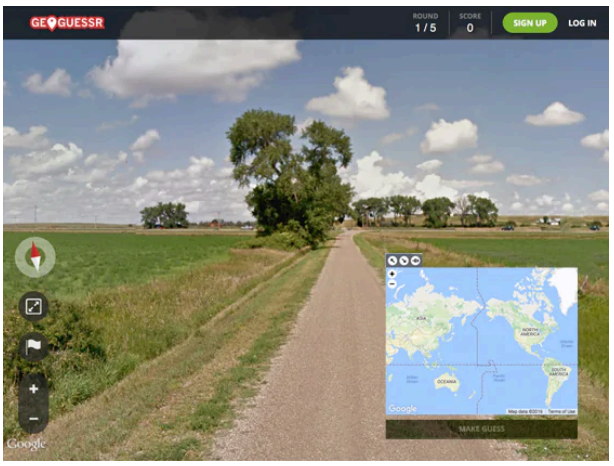
Created by Philipp Ollmann, Lukas Rockenschaub, Sebastian Weidlinger, Michael Zauner

1. Idea **2**
GeoGuessr 2
Initial Idea 2
Refined Idea 2
Goal 3
2. Dataset **3**
3. Data Preprocessing **4**
Mapping of the Countries 4
Image scaling and cropping 5
Potential approaches for data augmentation 7
Preprocessing in Keras 8
Data splitting 9
4. Model Architectures **9**
CNN 9
ResNet50 11
DenseNet 121 12
VGG-16 13
5. Training **14**
Approach 14
Transfer Learning 14
Training Tools 15
Training approach and continuous improvements 17
6. Evaluation **22**
Confusion Matrix 22
ROC Curve 22
VGG - center cropped Images 23
DenseNet - center cropped Images 24
ResNet50 - non cropped Images 25
ResNet - center cropped Images 26
VGG - non cropped Images 27
7. Lessons learned **28**
Transfer Learning is its own Science 28
Compute Resources and Image Data are vital 28
Overfitting Challenges with Complex Network Architectures 28
8. Ideas for improvement **29**
Places 365 Dataset 29
YOLO (You Only Look Once) 29
Capture Panorama 29
9. Conclusion **29**
10. Sources **31**

1. Idea

GeoGuessr

geoguessr.com
GeoGuessr is an online game that combines geography and exploration. In the game, players are placed in a random location around the world using Google Maps Street View. The objective is to guess the location's geographical coordinates based on the surroundings. Players can explore the environment, look for clues, and make an educated guess about where they are.



An exemplary illustration of a GeoGuessr game.

Initial Idea

The initial idea and one of the end goals for the project was to take a random GeoGuessr image and guess the country based on that image. The classification would be inspired by the way humans play GeoGuessr. Some examples of this approach include:

- Position of the sun
- Road markings
- Traffic signs
- License plates
- Position of the StreetView car (left- or right-hand traffic)
- Vegetation

After consulting with the professors, we concluded that this approach would be too complex and likely would not produce any satisfying results. It would need extensive preprocessing and even then a good outcome would be unlikely with our limited resources.

Refined Idea

We considered the provided feedback for the initial proposal and reduced the complexity of our goal. The refined idea and one of the end goals for the project is training a machine learning model using GeoGuessr images specifically from either Europe or Asia, aiming to obtain the corresponding country as the output.

Goal

As previously mentioned, the project aims to achieve accurate classification between Asia and Europe using GeoGuessr images. The performance of said classification should surpass the accuracy (0.5) of a baseline dummy classifier. Furthermore, we intend to assess various model architectures and configurations to identify and adopt the most effective model for our specific task.

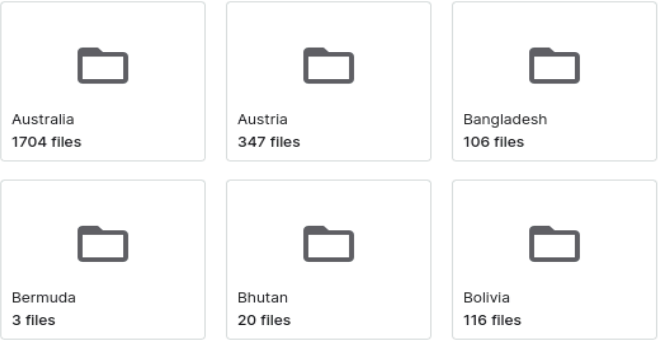
2. Dataset

[GeoLocation - Geoguessr Images (50K) (kaggle.com)]

We looked at various datasets and ended up with one consisting of 50000 images of GeoGuessr games as it would provide a reasonable amount of data to train the model on.

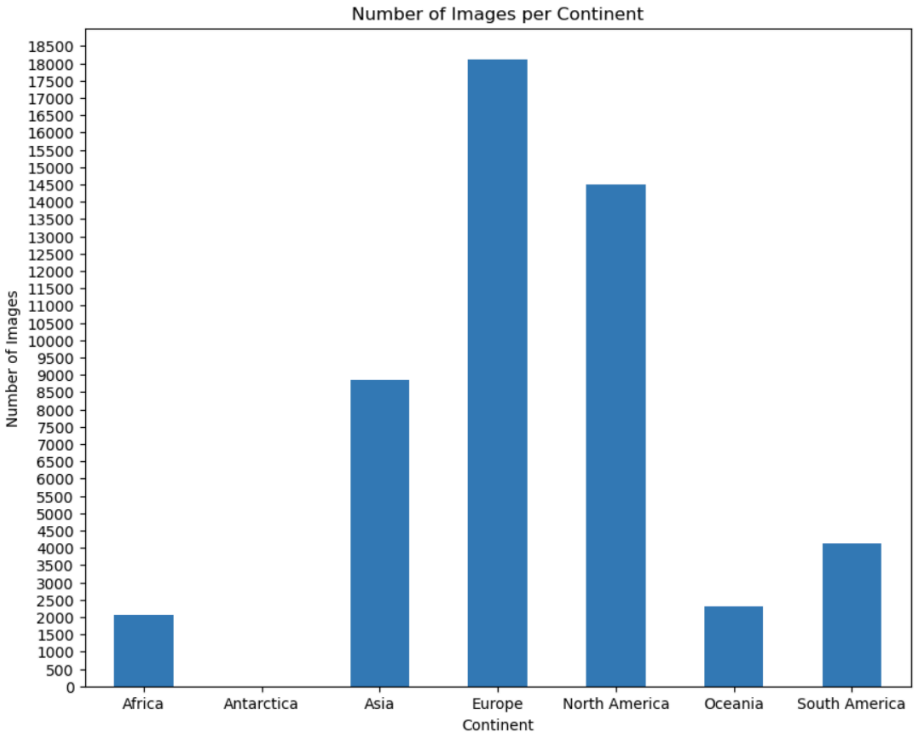
A sample image of this dataset.

This dataset is split up into folders representing the corresponding countries the images are from.



Since the GeoGuessr images are based on StreetView data, the amount of images per country is drastically different. For example, Antarctica only has one image featured in the dataset, while Austria has 350. Due to this significant class imbalance, we need to apply techniques to address this issue before proceeding with model training. For better visualization, we provided a plot with the amount of images featured in the dataset, grouped by the country. In this graph, we can see that while Europe roughly features 18000 images,

Asia only features 8000 images. Because of this imbalance, we opted to downsample the data of Europe to only use 8000 images, for model training, to have two balanced classes.



3. Data Preprocessing

Mapping of the Countries

Since we decided to limit the vast scope of the GeoGuessr task by only attempting to differentiate between two continents rather than all specific countries of the datasets, we needed a way to convert the country dataset we found on Kaggle. For context, most datasets that can be found online that are created through either scraping images from Google StreetView directly or from the actual game itself are usually arranged by country. Thus, upon downloading the dataset we were presented with a large folder with subdirectories for the countries.

Since we wanted to automate the association of the countries with the continents in case we want to add/combine multiple datasets in the future, we decided to use a Python package called pycountry-convert. It converts the country names to their country ISO codes, which in turn can be associated with their continent. The package worked quite well for our use case as we only had to filter out some islands and territories which would have only provided an insignificant number of images, at most 29 images for one of the sorted out countries, to the total of the dataset regardless.

Each continent is then represented in its own subdirectory of the target directory. In addition to the name of the continent to which each image was assigned, the name of the country, the path and the image dimensions are saved for further analysis in a CSV file, which is presented in the data section of this paper. The image below shows the fairly simple function

that was used for this part of the data processing which utilizes the package mentioned above:

```
# Helper function for converting the folder name of each country to the respective continent it is in
def country_to_continent(country_name):

    country_alpha2 = pc.country_name_to_country_alpha2(country_name)

    # Handle exceptions that are not part of the package
    if country_alpha2 == 'AQ':
        return 'Antarctica'

    country_continent_code = pc.country_alpha2_to_continent_code(country_alpha2)
    country_continent_name = pc.convert_continent_code_to_continent_name(country_continent_code)
    return country_continent_name
```

Image scaling and cropping

Early in the process of our research regarding the use of CNNs with TensorFlow, we realized that all model architectures we were looking into required the images to be square RGB images, of fixed size usually 224x224 pixels, which can be achieved in multiple different ways. The simplest way is to just let Keras take care of it when loading in the dataset. However, since we switched to Google Colab during the process and we wanted to keep the data load to a minimum when uploading as well as working with the dataset, we did the scaling ourselves. This is also beneficial because this way we can easily just look at the various images in their respective directory after performing manipulations on them.

To reduce the size, the images can be cropped or padding can be added to avoid having to squeeze the contents of the photo when it is down scaled to a square aspect ratio. Due to the wide aspect ratio of our images, we decided against adding padding as the actual content area of the images would be very small, leaving large parts of the 224x224 without information and just white. While we were not planning on cropping our images at first, we later decided to add a center crop option, because we were concerned about the model potentially using the map and other UI overlays that were part of the images for the recognition. An example of an image with and without center crop can be seen in the following graphic:

Sample non cropped

Sample center cropped

Technically, we could also use this center crop to not crop the image completely square, but to set a larger width. This would still result in a slight squeeze, but less than with the uncropped, scaled image.

Another thought when it comes to cropping was that we could slice the entire image into multiple smaller square tiles to retain more of the original image but split up into multiple tiles per photo, which would also drive up the total image count while avoiding having to squeeze the image. However, we decided against using this technique as we would still have to throw out the tiles that include the UI of GeoGuessr and the resulting individual images aren't representative anymore of a Google StreetView image.

To achieve the scaling and cropping that were described in this section we utilized the two following functions, the main one being our resize function which was then applied to each image in the dataset by using the map operator on the data frame that included all the path information for the input photos.

```
# resizes images to height and width dimensions with an optional center crop
def resize_sample(row: pd.Series) -> str:
    continent = row["continent"]
    image_path = row["path"]
    image_name = row["image_name"]

    image = cv2.imread(image_path)
    if CENTER_CROP:
        image = center_crop(image, (CROP_HEIGHT,CROP_WIDTH))
    scaled_image = cv2.resize(image, (SCALE_HEIGHT, SCALE_WIDTH))

    path = f'{IMAGE_FOLDER}/{continent}'
    if not os.path.exists(path):
        os.mkdir(path)

    img_path = f'{path}/{image_name}'
    cv2.imwrite(img_path, scaled_image)
    return img_path
```

This function was accompanied by the helper function for center cropping the images to the maximum height and width within the provided crop height and width, which in our case ended up being 600x600 for the cropping. Afterward, the 600x600 image is down scaled to 224x224. The center crop function can be seen in the following code sample:

```python
# center crops images to maximum possible height and width within the specified values
def center_crop(img, dim):
    width, height = img.shape[1], img.shape[0]

    # process crop width and height for max available dimension
    crop_width = dim[0] if dim[0]<img.shape[1] else img.shape[1]
    crop_height = dim[1] if dim[1]<img.shape[0] else img.shape[0]
    mid_x, mid_y = int(width/2), int(height/2)
    cw2, ch2 = int(crop_width/2), int(crop_height/2)
    crop_img = img[mid_y-ch2:mid_y+ch2, mid_x-cw2:mid_x+cw2]
    return crop_img
```

Potential approaches for data augmentation

A popular technique, especially for smaller datasets, is to utilize small image transformations, which are referred to as augmentation to artificially increase the size of the dataset by reusing samples in their altered form. Additionally, including this step in the process can be useful for limiting overfitting to some degree and ideally result in a more generalizable, robust model.

There are multiple operations, which can be performed on the images such as adding small random rotations as well as shifts and zooms of the image or random changes to the picture's brightness within a reasonable range. Furthermore, randomly flipping the samples either horizontally or vertically is another popular technique. Small perturbations such as blurring or distortions can also be easily applied with some tools.

This technique can be used for multiple deep learning approaches in ML, which is why both Keras and tools such as RoboFlow, which is primarily used for creating datasets and models for object detection with YOLO, provide functionality to apply these augmentations. RoboFlow offers both an API and a web interface to choose the desired operations, while also providing ways to directly see the effects of the augmentation on the size of the dataset within the web application. Keras used to provide the data augmentation as part of their ImageDataGenerator when loading datasets, but nowadays this can be done by adding augmentation and processing layers to the model itself. The benefit of this is that the operations are automatically applied to not only the training and validation data but also any data that is passed through the model to form a prediction. This means that new samples do not have to be augmented separately ahead of being used with the model. Another pro of this method is that the processing is done batch-wise and the altered images do not have to be permanently saved which saves disk space.

Not every augmentation makes sense for every problem. For instance, with our problem, we do not necessarily want to flip the images vertically, as StreetView images will not have the sky on the bottom. Nevertheless, in the early stages of the training setup, a random horizontal flip was applied to the images, however, this step was omitted after some testing due to a lack of computational resources and not wanting to work with even more images on our already limited hardware.

Nevertheless, we acknowledge that in a larger setup and adjusted approach, data augmentation would present a valuable technique to extend the dataset and ideally prevent overfitting the more complex models in such a fast manner as was the case with DenseNet. At the very least, these augmentations could aid in making the model more robust towards new input. Random rotations, shifts, and brightness changes along with horizontal flips would be our methods of choice in a revised approach as they potentially be a good fit for the problem. As mentioned earlier, we would omit flipping vertically or randomly cropping/zooming too far into the images as they would not resemble authentic StreetView images anymore.

Preprocessing in Keras

All of our considered and ultimately selected model architectures were available within the Keras API in TensorFlow, which not only enabled us to train the model but also provided us with the necessary tools to preprocess the image data according to the requirements of each architecture. This can of course also be done manually as it was mostly just scaling the values of the image data in the numpy array between either [-1,1] or [0,1], However, to avoid potential mistakes we decided to use the pre-processing function associated to each model which is available for each of our chosen architectures within Keras.

These functions can then be applied to the dataset by either using a map operation or by building a separate layer for both data processing and potential data augmentation to the rest of the model ahead of the first convolutional layer. The following images show the simple process that can be done thanks to Keras as the function can be imported with its respective model and applied after the dataset has been loaded in.

Import examples:

```
from tensorflow.keras.applications.densenet import DenseNet121, preprocess_input
```

```
from keras.applications.resnet import ResNet50, preprocess_input
```

Applying it on the dataset:

```
processed_train_data = loaded_train_data.map(lambda x, y: (preprocess_input(x), y))
processed_val_data = loaded_val_data.map(lambda x, y: (preprocess_input(x), y))
```

The Keras documentation for each model architecture provides insight into what kind of processing the function applies, which is mostly just scaling, but some model architectures such as ResNet expect a different kind of processing such as described in the following excerpt from the Keras documentation:

> Note: each Keras Application expects a specific kind of input preprocessing. For ResNet, call `keras.applications.resnet.preprocess_input` on your inputs before passing them to the model. `resnet.preprocess_input` will convert the input images from RGB to BGR, then will zero-center each color channel with respect to the ImageNet dataset, without scaling.

Data splitting

As with cropping and preprocessing, the splitting of the dataset into a train, validation, and held back test set is something that can either be done through the APIs of Keras and TensorFlow upon loading in the dataset for the training, or it can be done manually ahead of time. Either way, providing a seed for the random selection of samples makes sense to be able to achieve repeatable splits.

Once again we decided to split the data manually with a Python package to split the dataset beforehand instead of within the training script in the Google Colab environment as we ran into performance problems when splitting the data into multiple folders in Google Drive itself. Furthermore, the split which can be applied during loading of the dataset with Keras's functions does not account for a separate held-back test set, which we determined to be one of our requirements for evaluating our models.

The images from our continents of choice, in our case Asia and Europe, are split into three subdirectories. One for training, one for validation, and a third one to act as the held-back test set. This was easily achieved with the following function call thanks to a Python package called splitfolders.

```
splitfolders.ratio("scaled_images",seed=42, output="scaled_images_splitted", ratio=(0.7, 0.2, 0.1))
```

However, it is also notable that aside from this split we decided to artificially create balanced classes by using only 8000 images per continent. Therefore, the choice of the 8000 ahead of the train-test split was also randomized, once again with a seed of course. This was done with the sample method that can be used with dataframes:

```
if LIMIT_AMOUNT:
    for continent in CONTINENTS:
        df_sample = df_filtered_continent.groupby('continent').get_group(continent).sample(n=IMAGE_COUNT[continent], random_state=SEED)
        df_result= pd.concat([df_result,df_sample])
```

The downside to our approach is that for a different split, we had to re-upload the dataset to Google Drive for use within the Colab environment, but this was still more performant with a good internet connection than running the script in the cloud except for on the paid account.

In hindsight, we do acknowledge that upon studying for the exam and writing this paper we did realize that technically our test set includes a mistake as usually the original class balance should be kept for the held-back test set. This is something that would have to be adjusted for future work.

4. Model Architectures

CNN

After some initial research on the topic of image classification with the help of the in-class materials, YouTube, and various internet sources we decided that we will attempt to train a

neural network as part of this project. The research showed that convolutional neural networks (CNNs) are the common choice when it comes to image classification or similar tasks in the field of computer vision.

The big advantage that CNNs bring is the fact that the feature extraction is done by the network itself and it only requires the image to be converted into its pixel data, as well as preprocessed to some extent such as scaling the values between -1 and 1, as an input. Therefore, the features are learned by the network during the training process.

Strongly simplified, CNNs contain three kinds of layers that are assembled to form larger network architectures that will be described in upcoming sections. The three types of layers are convolutional layers, pooling layers, and fully connected, dense layers.

The convolutional layer is the main component of a CNN, and it is where the feature extraction takes place by “learning” the filters that can be used to identify features such as edges or textures from the image input. These layers have two important parameters which are the size of the extracted tiles from the input which is also referred to as kernel size and usually 3x3 pixels and the amount of the filters that are applied to the image to generate the output feature map. These 3x3 filters move over the grid of the input and extract the value for the output feature map through matrix multiplication, however, these mathematical details are outside the scope of this paper. Generally, a higher number of filters has two consequences, a larger number of features that can be extracted, but also an increase in training time. The goal is to use the smallest number of filters that are still able to extract the necessary features for the problem at hand. A convolution is usually followed by an activation function such as a Rectified Linear Unit (ReLU) operation, which we learned about in class and passed as an activation function to the layers that we instantiated with Keras for our training.

The second kind of layer is called the pooling layer which is where downsampling takes place to take the feature map that was the output of the convolutional layer and reduce its dimensions, essentially reducing the number of features and in turn the required computing power as well as weights. Like activation functions, there are various kinds of pooling operations out there, but max pooling is a popular choice. As the name suggests this process takes the maximum output from its neighborhood as a filter and it is moved through the feature map again, a similar principle to how it worked with the convolutional layer, but without weights. Another similarity is that pooling layers also take a filter size as an input parameter which is usually 2x2 pixels and a stride that defines the distance between each extracted tile. Aside from max pooling a second popular option is to use average pooling. This layer helps with a reduction in complexity and can help with limiting overfitting to a degree, but it is important to understand that information is lost in this process step.

Lastly, there are fully / densely connected layers which are usually found at the end of the network architecture. In these layers, all nodes are connected with all nodes in the next layer. This is where the actual classification takes place which often outputs a probability value from 0 to 1 which is the case when a softmax activation function is used for the last fully connected layer.

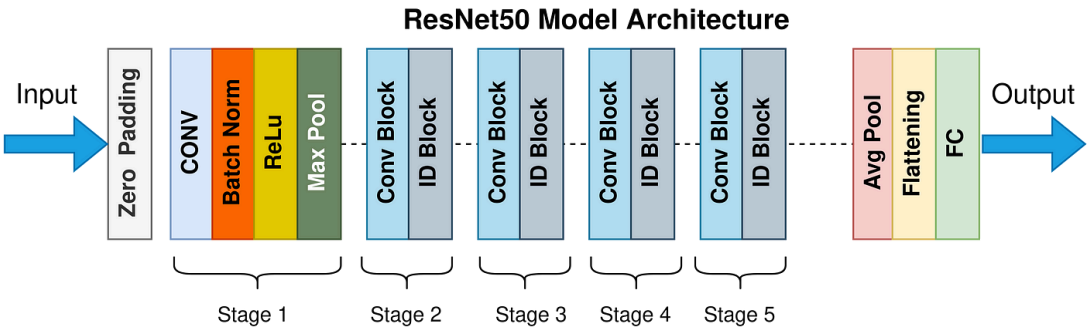
These layer types can be configured and assembled to build network architectures from scratch, but instead proven architectures can also be used which have been developed

through competitions in the field of image classification. Models of these architectures can then either be trained without initializing pre-trained weights or with the weights from previous training on a large, general dataset such as ImageNet.

Keras provides multiple ready-to-use architectures and pre-trained models that can be used with their API in TensorFlow. This is the path we chose for this project and the rest of this section will briefly describe the three architectures we chose for our project.

ResNet50

ResNet-50 is a deep learning architecture featuring 50 layers, renowned for its success in image classification tasks. It begins with an initial convolutional layer and max-pooling to extract basic features, followed by three sets of residual blocks with increasing filter sizes (64, 128, 256, 512). These residual blocks incorporate shortcut connections, enabling easier gradient flow and mitigating the vanishing gradient problem in deep networks. Global Average Pooling (GAP) reduces spatial dimensions, and a fully connected layer maps the output to the desired number of classes for classification. The depth of ResNet-50 enhances its ability to learn intricate features, making it effective in diverse computer vision applications.



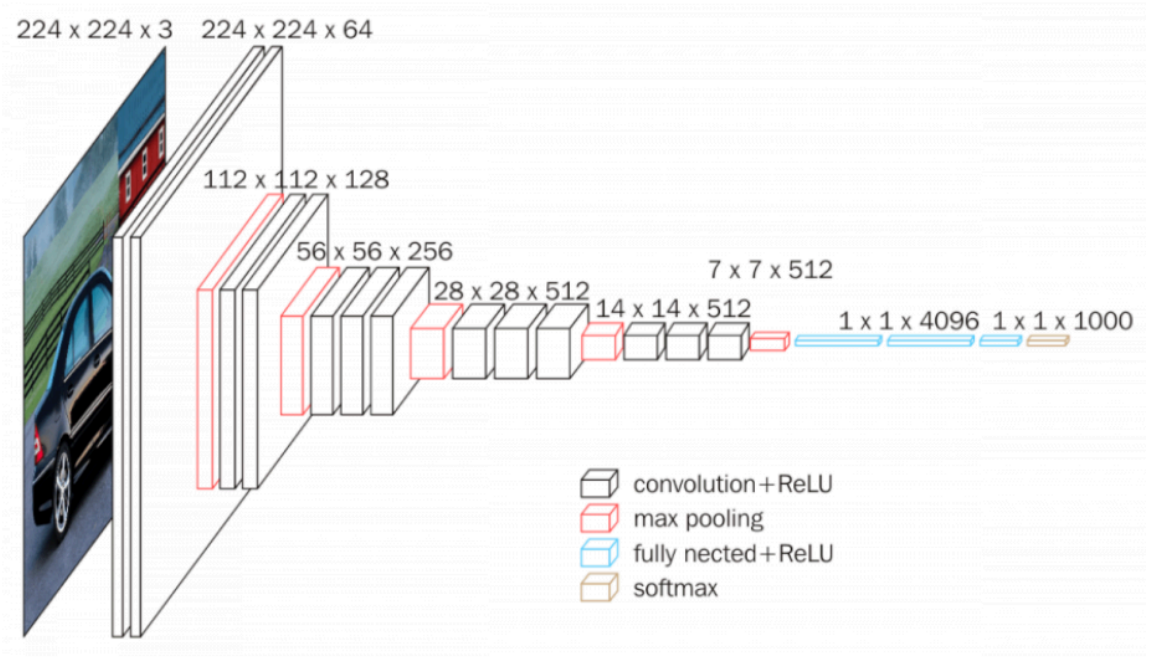
DenseNet 121

In comparison to the previously explained network architecture, DensNet121 is a far more complex and deeper network architecture. This was the result of an observation by researchers which found that CNNs can have a higher number of layers, a higher accuracy, and better training times when the layers are closely connected. This is achieved by designing the architecture to be feed-forward with full connections between each layer and its predecessor. The result of this is that all feature maps of all previous layers are used as input for the current layer. Then the result of this layer along with all the other feature map information is passed onto the next layer. This allows for stronger feature propagation and reuse. As with the VGG-16, another goal of this configuration is to reduce the total number of required parameters. These densely connected layers are arranged in four so-called dense blocks which are separated from each other by transition layers which is where pooling operations are performed. One common problem of these very deep network structures is known as the vanishing gradients as the information gets lost on the way through the network both forward and backward. Residual networks (ResNets) also aim at solving this problem however while they combine features using summation before passing on the feature maps, DenseNet instead combines the feature maps through concatenation which is its big advantage. The network architecture can be seen in the following image:

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

VGG-16

This network architecture was developed by the visual geometry group at the Department of Engineering Science in Oxford. The paper and model architecture were developed to compete in the ImageNet classification challenge in 2014. Their research resulted in two public CNN models that can be used through the Keras API. One variant which is called VGG16, has 16 trainable layers and the larger model called VGG19 has 19 trainable layers. Both architectures use a small, but stacked, 3 by 3-pixel convolutional filter, which was fixed at the beginning of the group's research as it was their goal to determine the effect of increasing the depth of the network. The layer count, and therefore the complexity, was increased until desirable results were reached. At the time when the paper was published, the small filter size was a novel approach compared to the other competitive models that achieved solid results at the competitions in 2012 and 2013 which used 11x11 or 7x7 filter sizes. The idea behind the smaller filter size was to limit the total parameter count when increasing the number of layers. The input images should be 224x224 RGB images which are then passed through the convolutional layers. The hidden layers use ReLU activation functions and some of the convolutional layers are followed by max-pooling layers that use the typical 2x2 window size that was described earlier. The following image shows the architecture of the network.



5. Training

Approach

Our approach to the training cycles of our selected models can be best described as continuous trial and error. After initially running the first few attempts at our model training scripts locally we quickly figured out that a true parameter grid search as we learned about in class and applied in our exercises would not be feasible. Sadly this was still true even when running the models in Google Colab on remote GPU systems as training that would last too long would usually run into a session timeout.

However, our move to Google Colab still enabled us to somewhat quickly prototype multiple training configurations for each architecture during the late hours of the day when the GPUs usually became available in the free tier. This way we were able to adjust parameters such as the layers, learning rate, and optimizer values between the runs of each model. This enabled us to try a few configurations even without a proper grid search. However, ideally, this same approach would have worked better in a more controlled and automated fashion.

Initially, we took an experimental approach to our model training as we had to learn how to set up our training script and the different ways these large networks can be trained. From building your own network to training proven architectures from scratch or using pre-trained weights to initialize the models there are many different ways to approach the training which we had to learn about first.

Transfer Learning

Since we lacked the computational resources to train one of the previously described network architectures completely from scratch for long enough, we instead applied transfer learning. A side note we did try hand-building the VGG16 architecture as the layers are fairly simple, but even after letting it run for multiple hours and a high number of epochs without early stopping it was lacking the desired prediction power, which is why the results were omitted. Instead, we quickly learned that it makes sense to initialize the models with the pre-trained weights which were trained on much more powerful systems, larger datasets, and for a substantial amount of time.

At first, this happened implicitly due to the tutorials and code samples that can be found online when researching ways to use Keras with TensorFlow in Python. Most of them take into account that most people trying to learn about using these technologies will likely not be able to do so on powerful systems right away, which is why they take a transfer learning approach which allows the user to reach results fairly quickly, especially when classifying content that is closely related to the ImageNet dataset. Typical tutorial examples that quickly reach high results are cats vs. dogs or flower type classification.

However, after some research and getting a better understanding of the code samples and training process we started to look specifically into transfer learning approaches for our problem. Knowing this, it was easier to find information on the layer structures to apply on top of our models as well as the range of learning rates and optimizer values we should be looking at.

Transfer learning describes the process of using an existing model that was pre-trained for a different task, such as classifying the images from the ImageNet dataset, and trying to use this pre-existing “knowledge” of the model for a new, ideally related challenge. This is done by freezing either all the layers or some number of layers of the pre-trained architecture before extending it with some trainable layers. The idea behind this is that we do not want to train the millions of weights and parameters of the existing deep learning architecture and to instead only train a smaller number of layers in a short amount of time and with fewer computational resource requirements. Ideally, we want to be able to transfer as much of the pre-learned weights from the previous domain to the new domain.

There are various ways to approach this process, as in some tutorials the entire network architecture is trained for a couple of epochs before the weights are frozen, while others apply transfer learning right away and freeze all layers from the start. The number of layers can also differ a lot as we came across samples that used as little as one dense layer and pooling layer ahead of their final output layer for the classification, while others built entire small architectures consisting of multiple convolutional layers and dense blocks.

Another important aspect of transfer learning is that a small learning rate should be used as we just want to fine-tune the existing weights to our new problem. Additionally, it can make sense to apply regularization through using dropouts to limit overfitting. This proved to be very valuable in our use case as the complex models would otherwise overfit the training data within less than 5 epochs.

Training Tools

We have mentioned some of the tools that we used for our model training in this paper already such as the Keras API and some of the functions that come with it. Additionally, our use of Google Colab was briefly described. Throughout the training process, we decided to get the pro version on one of our Google accounts for less limited access to the GPUs.

During the training itself, we made use of multiple callback tools which we introduced into our training script after learning about them in various online tutorials. These callback functions can be added to the mode.fit() method and are then included in the training process. The two main ones that are part of many tutorials are the EarlyStopping and the ModelCheckpoint function. Both brought significant improvements as before implementing them we ran into issues with losing progress whenever the training script would crash after a certain amount of epochs resulting in our custom code for saving the model after the set amount of epochs are completed not being run. Additionally, we had to let the training run through completely for our set amount of epochs even if the model was already overfitting as we were lacking the stopping mechanism.

The Early Stopping function solves one of these issues by interrupting the training if no more significant progress has been made for a set amount of epochs. The latter can be passed as a parameter as patience, which describes how many epochs without an improvement in the monitored metric, in our case the validation loss, the model will allow. Therefore, if the patience is set to 3, the training will be stopped after three epochs without any improvement to the validation loss, but if there is another improvement after two epochs for instance, then the counter will reset. The goal is to not waste computational resources and time on a model

that is not improving based on the validation, even though the training metrics are still getting better, which can be a sign of overfitting. This becomes especially obvious on complex networks such as DenseNet as it will be able to turn into a look-up table of the training data if given enough time or a small dataset. In this case, we had the accuracy jump up to above 95% for the training data while the validation results were not improving anymore at all, which is exactly when this early stopping mechanism came in to stop the process.

The images below show the brief setup of the early stopping and its effect on the training, in the case of our VGG16 model we used patience of 3 epochs after tinkering with the value throughout training cycles and discovering that with this architecture we were able to get close to our maximum results quickly and saw little to no improvements afterward, but it also did not have the same overfitting tendencies as DenseNet:

```
train_early_stopper = EarlyStopping(
    monitor = 'val_loss',
    patience = 3,
    )
```

```
Epoch 12/30
350/350 [==============================] - ETA: 0s - loss: 0.4183 - accuracy: 0.8102
Epoch 00012: val_loss improved from 0.48186 to 0.48041, saving model to models/vgg16_model_best_sgd.hdf5
350/350 [==============================] - 65s 186ms/step - loss: 0.4183 - accuracy: 0.8102 - val_loss: 0.4804 - val_accuracy: 0.7766
Epoch 13/30
350/350 [==============================] - ETA: 0s - loss: 0.4053 - accuracy: 0.8177
Epoch 00013: val_loss did not improve from 0.48041
350/350 [==============================] - 55s 157ms/step - loss: 0.4053 - accuracy: 0.8177 - val_loss: 0.4942 - val_accuracy: 0.7763
Epoch 14/30
350/350 [==============================] - ETA: 0s - loss: 0.3997 - accuracy: 0.8195
Epoch 00014: val_loss did not improve from 0.48041
350/350 [==============================] - 55s 157ms/step - loss: 0.3997 - accuracy: 0.8195 - val_loss: 0.4922 - val_accuracy: 0.7694
Epoch 15/30
350/350 [==============================] - ETA: 0s - loss: 0.3896 - accuracy: 0.8248
Epoch 00015: val_loss did not improve from 0.48041
350/350 [==============================] - 55s 157ms/step - loss: 0.3896 - accuracy: 0.8248 - val_loss: 0.4972 - val_accuracy: 0.7638
```

This screenshot shows the early stopping kicking in after the validation did not improve for three epochs, which led to the training getting interrupted.

Additionally, the image above also shows the second tool in action as the ModelCheckpointer can be seen after the first epoch in the screenshot. This saves the model file which includes both the current weights and the architecture of all epochs where the validation loss improved. However, it is important to note that in our configuration the previous checkpoint was overwritten to limit the amount of model files within the Google Colab environment and Google Drive. This can be a problem when the patience is set far too high and the validation loss may improve a tiny bit after for example 5 epochs of no improvement which will trigger a new checkpoint getting saved. This by itself does not have to be a problem, but in the worst case, your existing checkpoint will get overwritten with a model that has already been overfitting for multiple epochs. Knowing this we decreased the patience setting for our model training as we had it much higher at the beginning leading to the checkpoint mechanism overwriting a decent model with an overfitting one that improved its validation loss by a very very small amount, triggering the new save file. Once again the setup was quite simple and can be seen below.

```
train_checkpointer = ModelCheckpoint(
    filepath = 'models/dense2_model_best_adam.hdf5',
    monitor = 'val_loss',
    save_best_only = True,
    mode = 'auto',
    verbose=1
)
```

These two callbacks are only examples and what we ended up using, but there are more functionalities out there such as the display of the current train/test error curve after each epoch. However, this required a keyboard input after each epoch to close the graph and continue the execution, which is why it was removed again as we just decided to go off of the textual information in the logs and since we wanted to leave the training running for longer on its own.

Training approach and continuous improvements

This section will provide some details about the trial and error approach we took towards our training cycles as well as the adjustments we made based on the knowledge we gained through looking more into transfer learning.

Our very first training setup and run was not explicitly transfer learning, as at the time we did not know as much about it. Instead, it was important to us to get up and running to figure out some important things such as how long the training takes, can we get the model we chose, which at first was just DenseNet, to train properly and will it be able to yield results within our problem domain. Therefore, the initial setup did not freeze all of the layers and only applied two additional dense layers on top of the existing architecture. Furthermore, a relatively high learning rate of 0.01 was used for the Adam optimizer. In hindsight, this initial setup was prone to overfit and this is exactly what happened as DenseNet achieved a 98% accuracy on the training data while the validation accuracy was nowhere near as high. At the time we used even fewer images per continent and did not yet use early stopping, which is why the training continued, even due to the overfitting.

Even though this initial approach can be seen as a total failure, it taught us a lot as it was the primary trigger to look deeper into transfer learning as well as other model architectures. This was the point in time when we started researching as well as trying multiple other architectures that were available within Keras which included ResNet 50, ResNet50v2, VGG16 and VGG19, MobileNet, and InceptionNet. After some research and test runs, we decided to use VGG16, because it showed some potential for our problem while also being far less complex compared to our existing choice, the DenseNet architecture. Our third architecture of choice was the popular ResNet50, which sits in the middle between the other two networks when it comes to its depth and it proved to be a popular choice in many tutorials as well as papers. We did look into its predecessor, version two of the model, but similarly to some blog posts, we found online, our experience with ResNet50v1 was better. The same reason was behind our choice of VGG16 over VGG19.

From here on out we took a more systematic approach to making adjustments to our training script as with all three models we started to try multiple runs with varying learning rates as well as optimizers. The learning rates we ended up landing on are far smaller than our initial values which were 0.01 for the Adam optimizer and 0.001 for the SDG optimizer. Our final learning rates we decided to stick with were 0.0005 for ResNet50 and 0.0001 for VGG16.

The second kind of adjustment we made between cycles was to change the number of layers we applied on top of the frozen layers from our chosen model architecture. As previously mentioned, the layer count started out fairly small with as little as three additional layers, but this proved to be not enough trainable parameters for our problems as it led to severe underfitting with the accuracy on both the training and test validation being in the mid-50s in percent. Unfreezing more layers, such as half of the DenseNet architecture was not the right approach as this reversed the severe underfitting into essentially immediate overfitting within the first 5 epochs.

Therefore we chose to increase the layer count of our transfer learning model, basing our decisions for the layer types and amount off of the previous runs as well as online resources. While some approaches used multiple convolutional blocks, most of the ones we found used dense layers along with pooling and dropouts. This is why we started experimenting with different amounts of dropout depending on the model architecture as well as different configurations for the densely connected layers. An example can be seen in the image below which is part of our ResNet50 training script:

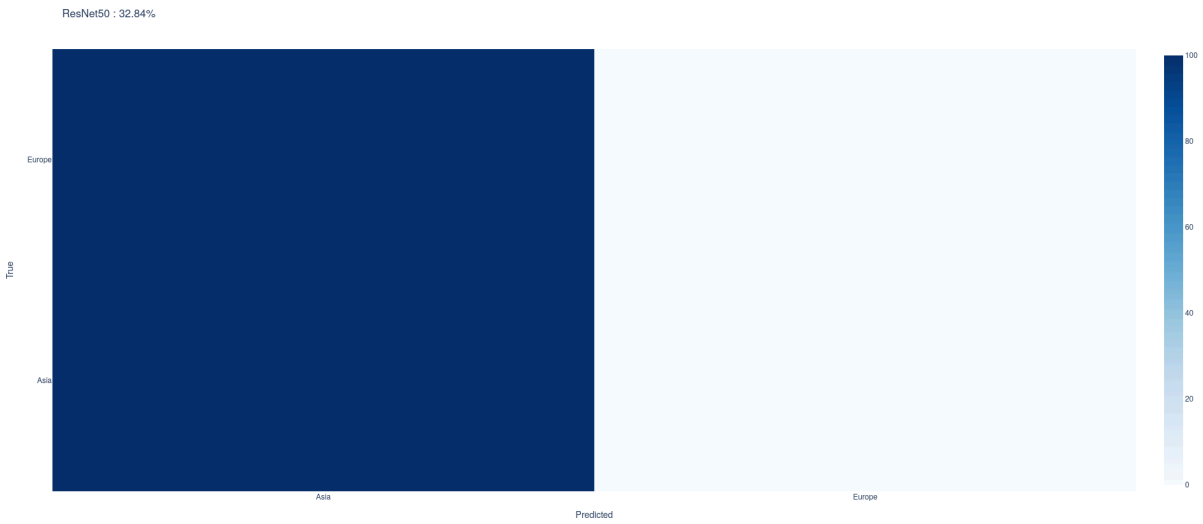
```
# Load pre-trained ResNet50v2 model with the ImageNet weights,
#fully connected top layer isn't included because we will do transfer learning
resnet50_model = ResNet50(
    include_top=False,
    weights='imagenet',
    input_shape=(IMG_HEIGHT, IMG_WIDTH,3)
)

# Freeze the layers in the pretrained model as we do not want to train them anymore
for layer in resnet50_model.layers:
  layer.trainable = False

# Create our transfer model that will use the pretrained resnet as input
transfer_model = Sequential()
transfer_model.add(resnet50_model)
transfer_model.add(GlobalAveragePooling2D())
transfer_model.add(BatchNormalization())
transfer_model.add(Dense(512, activation='relu'))
transfer_model.add(BatchNormalization())
transfer_model.add(Dense(256, activation='relu'))
transfer_model.add(BatchNormalization())
transfer_model.add(Dropout(0.5))
transfer_model.add(Dense(128, activation='relu'))
transfer_model.add(Dropout(0.5))
transfer_model.add(BatchNormalization())
transfer_model.add(Dense(64, activation='relu'))
transfer_model.add(Dropout(0.3))
transfer_model.add(BatchNormalization())
transfer_model.add(Dense(2,activation='softmax'))
transfer_model.summary()
```

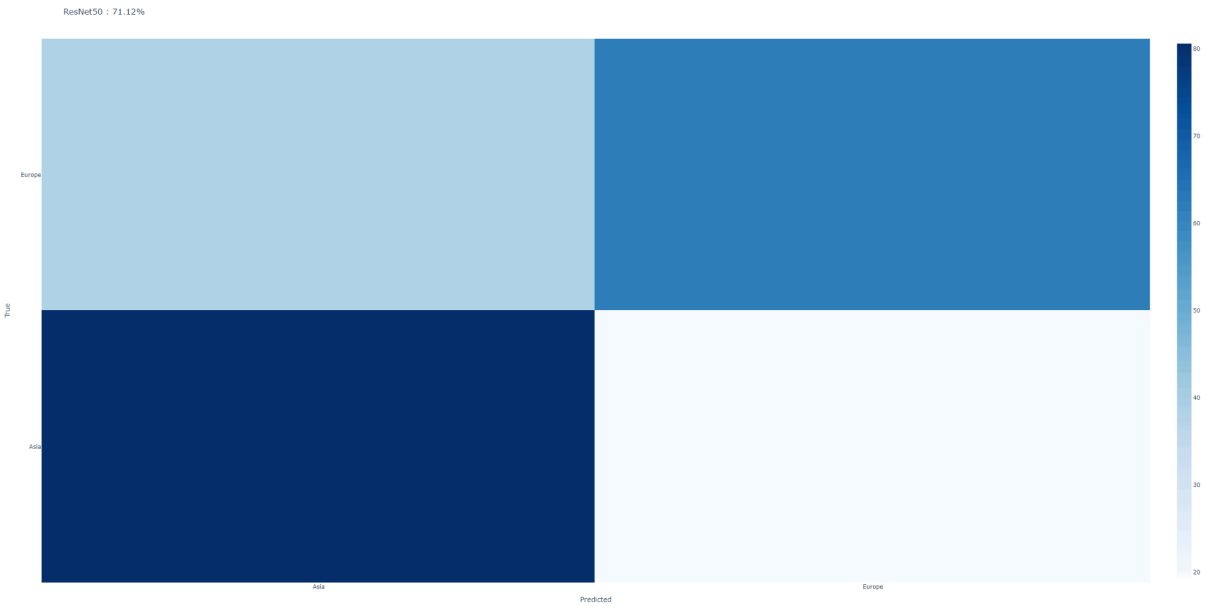
An important part of this is that the last dense layer has to be passed the number of classes we wish to classify, which in our case is 2. The number of dropout layers varied between 0 and as much as 5, while its dropout rates varied from 0.1-0.7. The dimensionality of the output space was changed multiple times as well for the dense layers, but using too many layers or values that were too high just ran into overfitting again.

A lot of online sources also used a Flatten() layer of the Keras API after its first layer after the frozen layer, however, this was the source of one of our major problems with ResNet. The layer does exactly what its name would suggest as it takes an input and flattens it as the example from Keras shows:

```
>>> x = keras.Input(shape=(10, 64))
>>> y = keras.layers.Flatten()(x)
>>> y.shape
(None, 640)
```

Multiple code samples and articles used this instead of a pooling layer, especially with Resnet and DenseNet which is why we chose to try to implement it as well. It did not seem to harm our training and validation set accuracies, but this is exactly why it is important to properly evaluate the model as our held-back test set with its confusion matrix and ROC curve revealed a huge problem. The accuracy was completely one-sided, meaning that depending on the runs and seeds it was either everything predicted as Europe or as Asia. This problem was mentioned during our presentation and after some more experimenting it seems like that the Flatten layer was the cause behind the issue. It was replaced with a GlobalAveragePooling layer, which is the setup we used for our VGG training script, and as the confusion matrices below show it helped with the issue.
The first matrix is with the Flatten layer, which just led to a classifier that in this specific case nearly exclusively predicted Asia, while the matrix below shows a better result where the classifier was able to achieve around 80.5% on our test set for Asia and 61.6% on the Europe images. This is still quite imbalanced, but a far better result than the model using the Flatten layer.

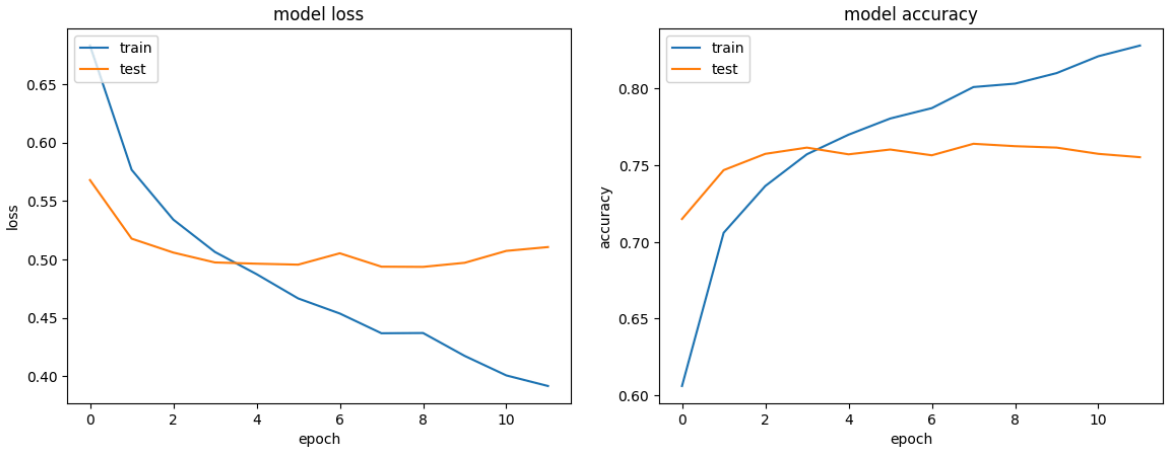




A similar issue was encountered with DenseNet, which we initially also utilized a Flatten layer for, but in this case, it was not resolved by simply switching it out for a pooling layer. We experimented with multiple different setups with different layer structures, freezing just parts of the network architecture and deliberately using the 5th convolutional block to also train its weights. Different random seeds and batch sizes were also explored. Additionally, we tried using a different activation function than ReLU and also switched out the optimizer's parameter values, but in the end, we were unable to find a solution to this issue. Even when using the identical structures for our layers that are added on top of the completely frozen architectures and using the identical code as with the other model architectures for data loading as well as visualization we still ran into the issue. Therefore the results are not representative nor meaningful, but our struggles with this model architecture still initiated some research and brought us a fair share of learnings as it was the most prone to overfitting, which is why we had to read up on regularization.

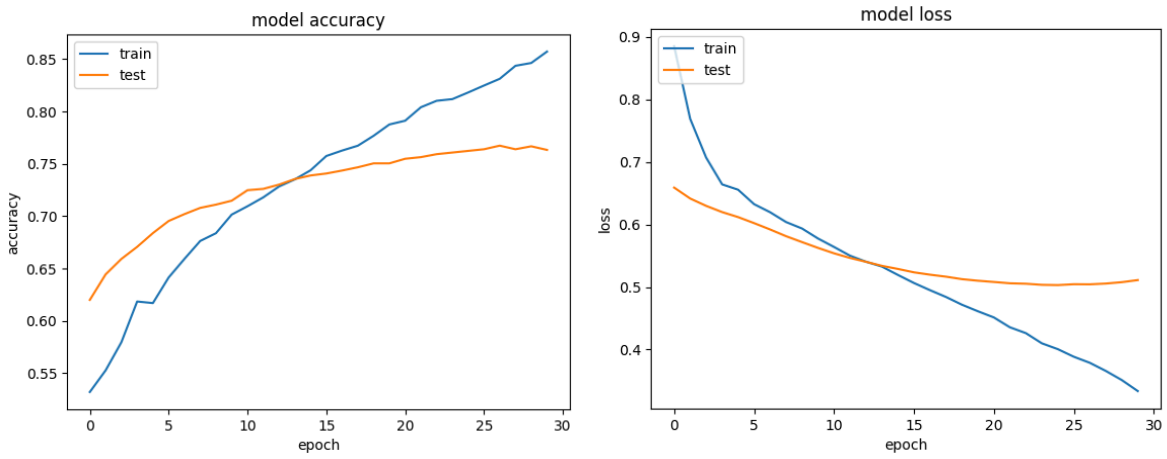
Lastly, even with applying regularization, we were unable to fully avoid overfitting which is likely due to our relatively small dataset in combination with our complex network architectures. As expected, DenseNet was able to overfit the most drastically by becoming a near-perfect look-up table for our training data if given enough time or unfrozen layers to work with. The same can be said about ResNet50, however, it does not do so with the same power as DenseNet, meaning it does take a couple more epochs, but is still easily able to achieve 95% accuracy on the training data within the 30 epochs if no early stoppage or regularization is applied. To no surprise, even VGG16 started to overfit at some point even with its smaller depth and complexity, but it was still by far the easiest model architecture to work with. The graph below shows the development of the loss values during training for the VGG16 model, which shows that it doesn’t take long until overfitting occurs in our final setup, but even with more aggressive regularization measures our resulting accuracies remained about the same, even if the point of the two lines intersecting came at a later point in time.

VGG16 loss curves from training in relation to its accuracy curve



Here we can see that while the accuracy difference between train and validation shows a clear gap at the point where the overfitting starts, the total accuracy does not rise as quickly as with the more complex models. Additionally, it is worth mentioning that while it did take fewer epochs for VGG16 to reach, what we deemed its full potential for our approach after a lot of trial and error, it does have a much longer training time for the first epoch compared to ResNet as it took 1180 seconds to complete the first epoch, while ResNet needed only 158 seconds.

ResNet50 loss curves from training in relation to its accuracy curve



The graphs above show that it did take ResNet50 more epochs before the two curves intersect, but at the point of overfitting it will do so strongly even with regularization. Without it, we had ResNet50 achieving results in the mid 90% for accuracy on the training data while drastically underperforming on the validation data within the first 10 epochs. As mentioned earlier, the initial training startup for ResNet is faster and the epochs following it took usually around 60 seconds to complete within a GPU-supported Google Colab environment. The curves were after the early stoppage occurs or the set amount of epochs is reached.

6. Evaluation

Confusion Matrix

A confusion matrix is a tabular representation used in machine learning to evaluate the performance of a classification model. It compares predicted and actual outcomes, breaking down results into four categories: true positives (correctly predicted positive), true negatives (correctly predicted negative), false positives (incorrectly predicted positive), and false negatives (incorrectly predicted negative). The matrix allows for a quick assessment of a model's precision, recall, and overall accuracy. It is particularly beneficial because it provides a clear snapshot of a model's strengths and weaknesses, aiding in the identification of areas for improvement. By offering a detailed breakdown of classification results, the confusion matrix enhances the interpretability and diagnostic capabilities of machine learning models.

test_ds: test dataset splitted from the original dataset
true_labels_int = true class names represented as integers (Asia: 0, Europa: 1)
predictions_binary: predictions from the model mapped as integers (Asia: 0, Europa: 1)

```
# Get model predictions
predictions = dense_model.predict(test_ds)

# Convert predictions to binary (0 or 1) based on a threshold
predictions_binary = (predictions[:, 1] > 0.5).astype(np.int32)

# Compute confusion matrix
confusion_matrix = tf.math.confusion_matrix(labels=true_labels_int, predictions=predictions_binary, num_classes=2)

# Normalize confusion matrix to percentage
cm_percentage = confusion_matrix / tf.math.reduce_sum(confusion_matrix, axis=1, keepdims=True) * 100
```

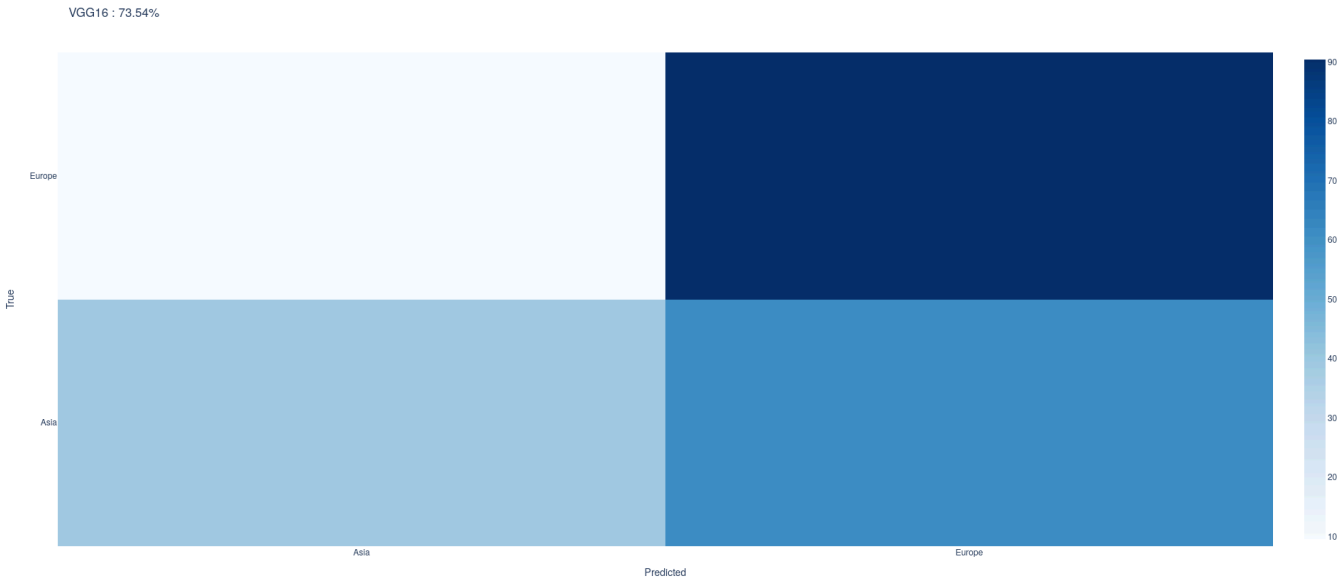
ROC Curve

The Receiver Operating Characteristic (ROC) curve is a graphical tool for assessing the performance of binary classification models. It plots the true positive rate against the false positive rate at different decision thresholds. The curve visually illustrates the trade-off between sensitivity and specificity. A diagonal line represents random chance, and a curve above it signifies better-than-random performance. The steeper the curve, the better the model. The point at the top-left corner represents perfect classification. The area under the ROC curve (AUC) quantifies overall performance, with a higher AUC indicating better discrimination. ROC curves are useful for selecting an optimal operating point based on the desired trade-off. They provide insights into a model's ability to distinguish between classes, aiding model evaluation and comparison.

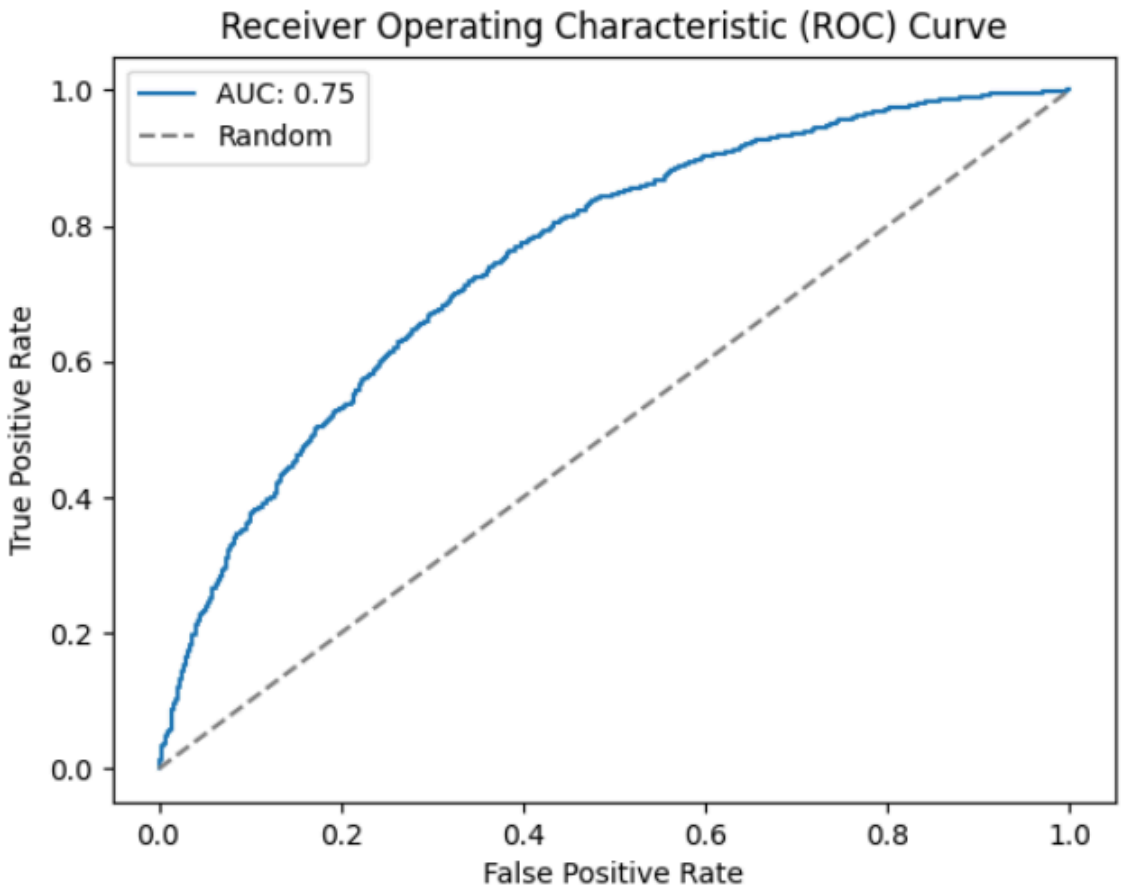
true_labels_int = true class names represented as integers (Asia: 0, Europa: 1)
predictions = predictions made from the model with the test dataset

```
auc = roc_auc_score(true_labels_int, predictions[:, 1])

fpr, tpr, _ = roc_curve(true_labels_int, predictions[:, 1])

plt.plot(fpr, tpr, label=f'AUC: {auc:.2f}')
plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
```

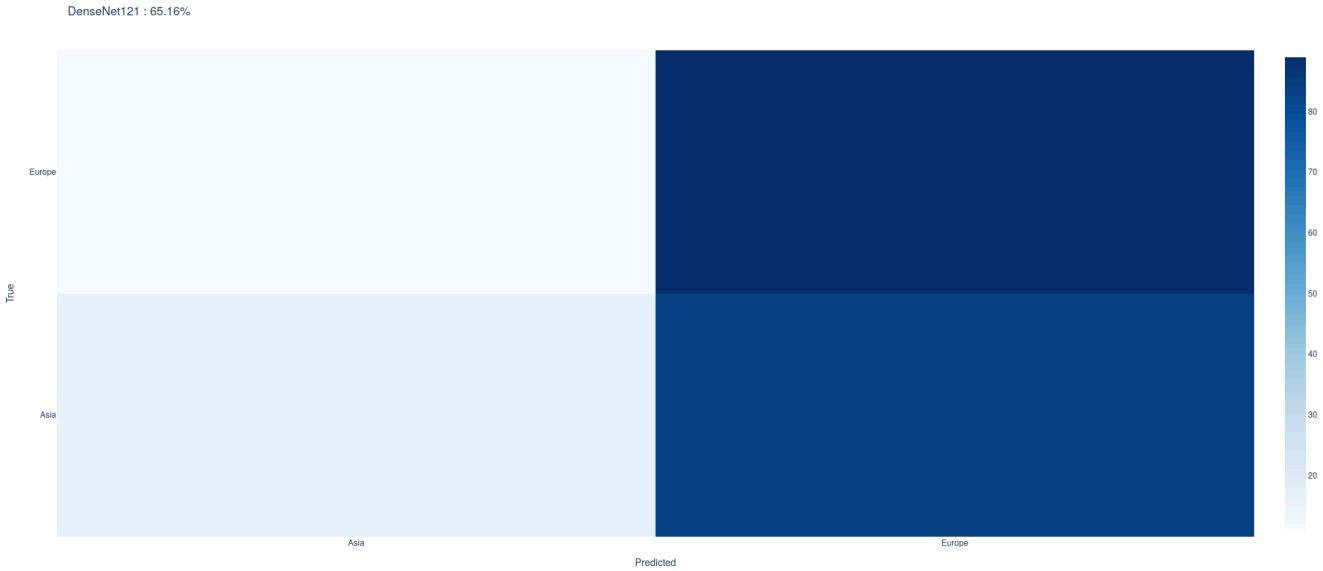
VGG - center cropped Images



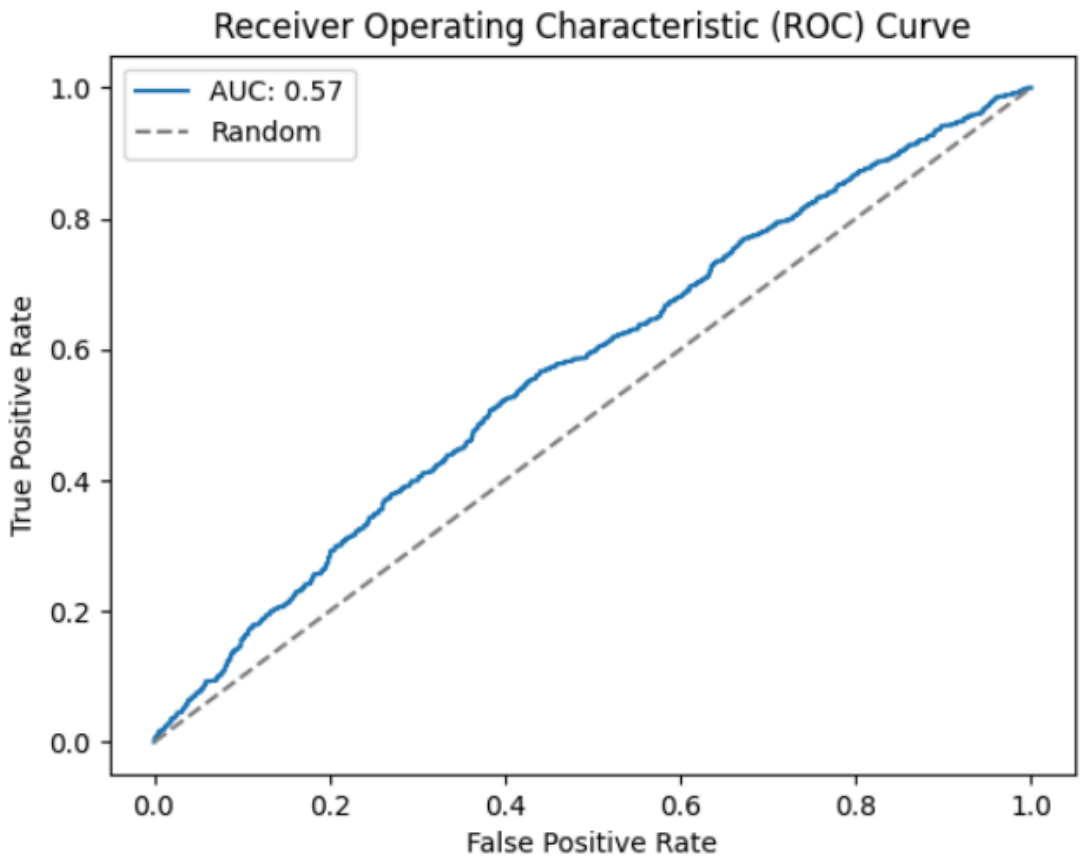
The VGG model tends to overfit “Europe” as the correct prediction of Europe works well (~90%) but the prediction of “Asia” only gets ~40% right.



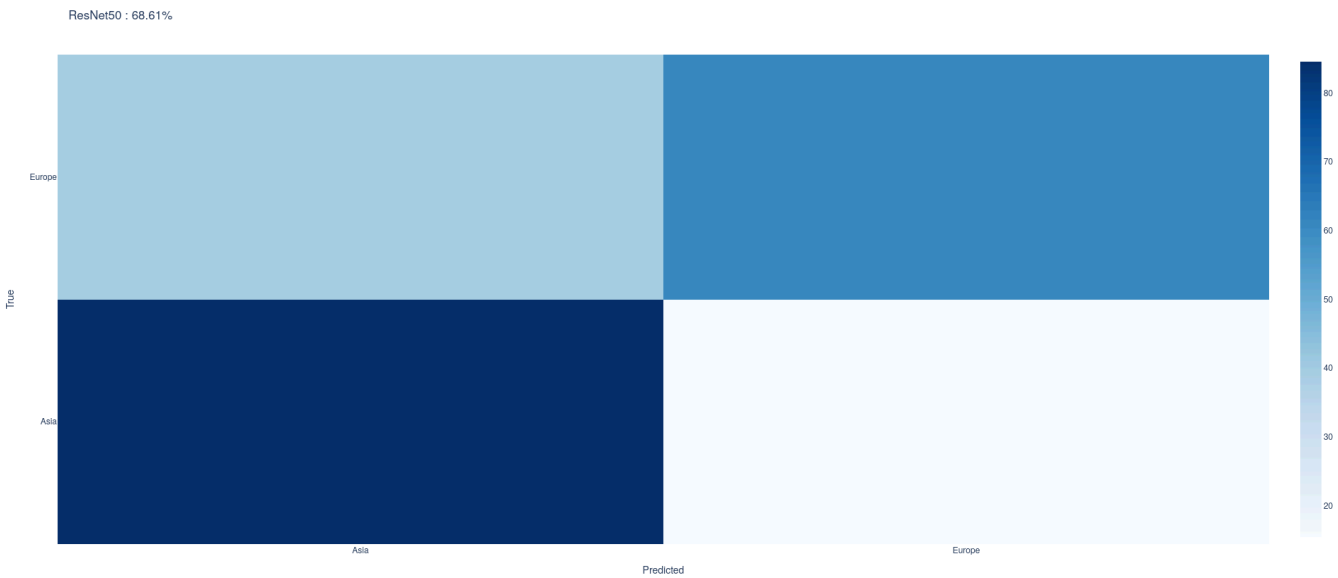
DenseNet - center cropped Images

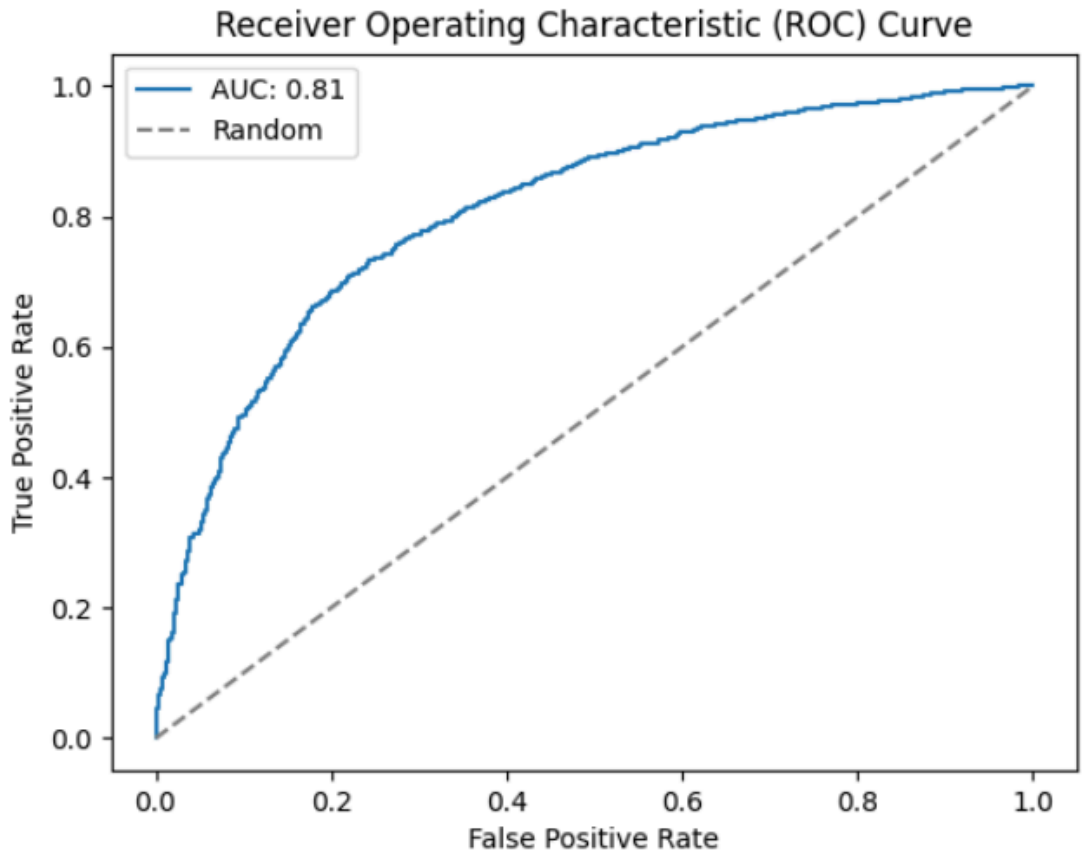


Similar result for the DenseNet model but it mostly predicted Europe even though Asia was the correct class.

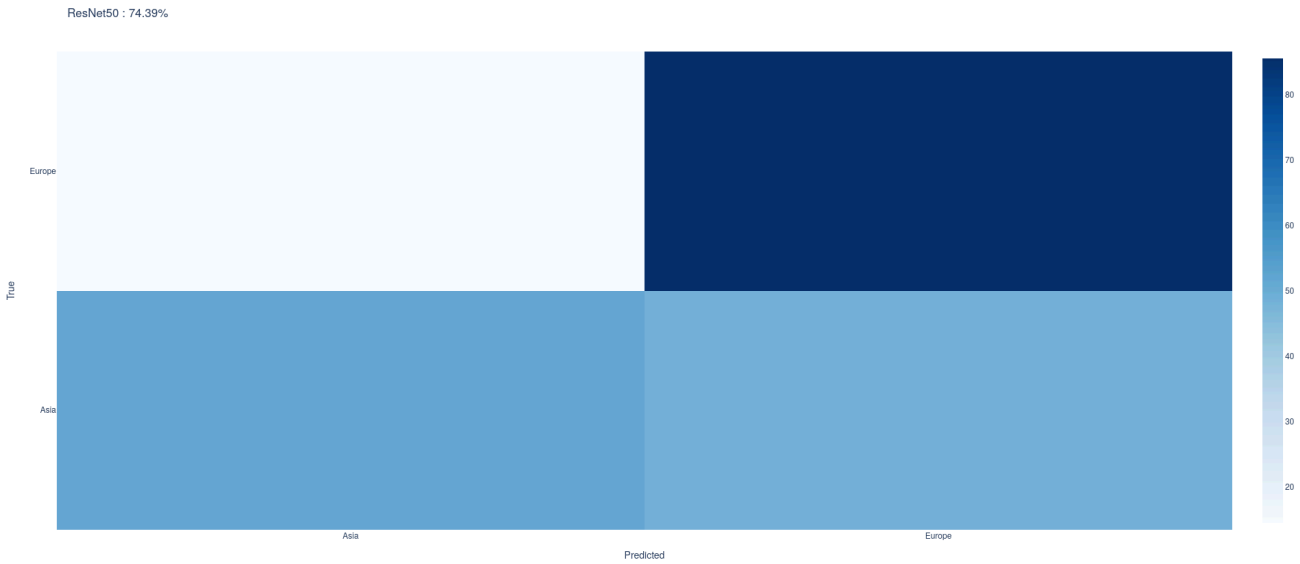


ResNet50 - non cropped Images

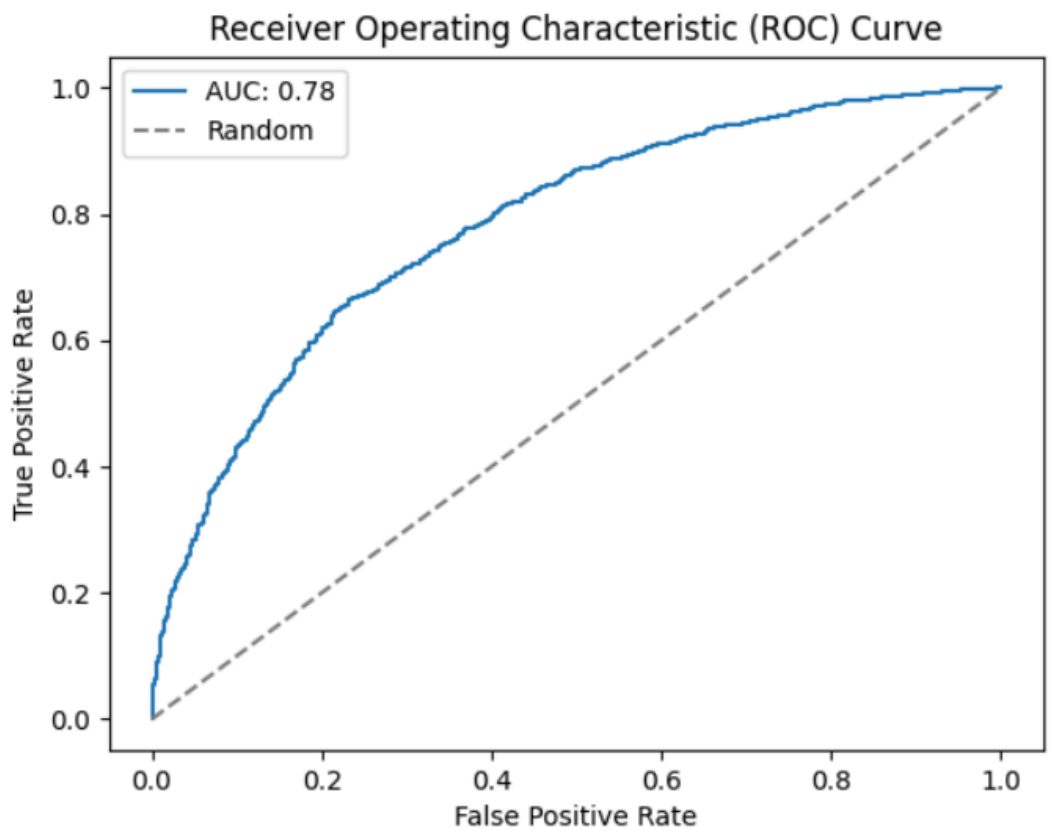




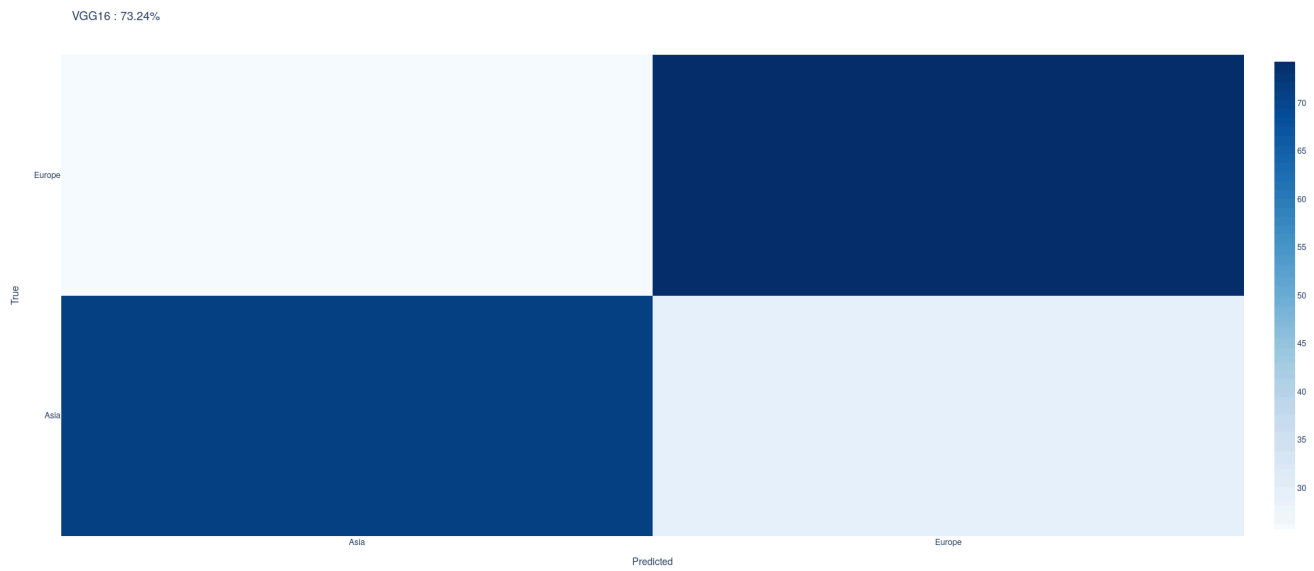
ResNet - center cropped Images



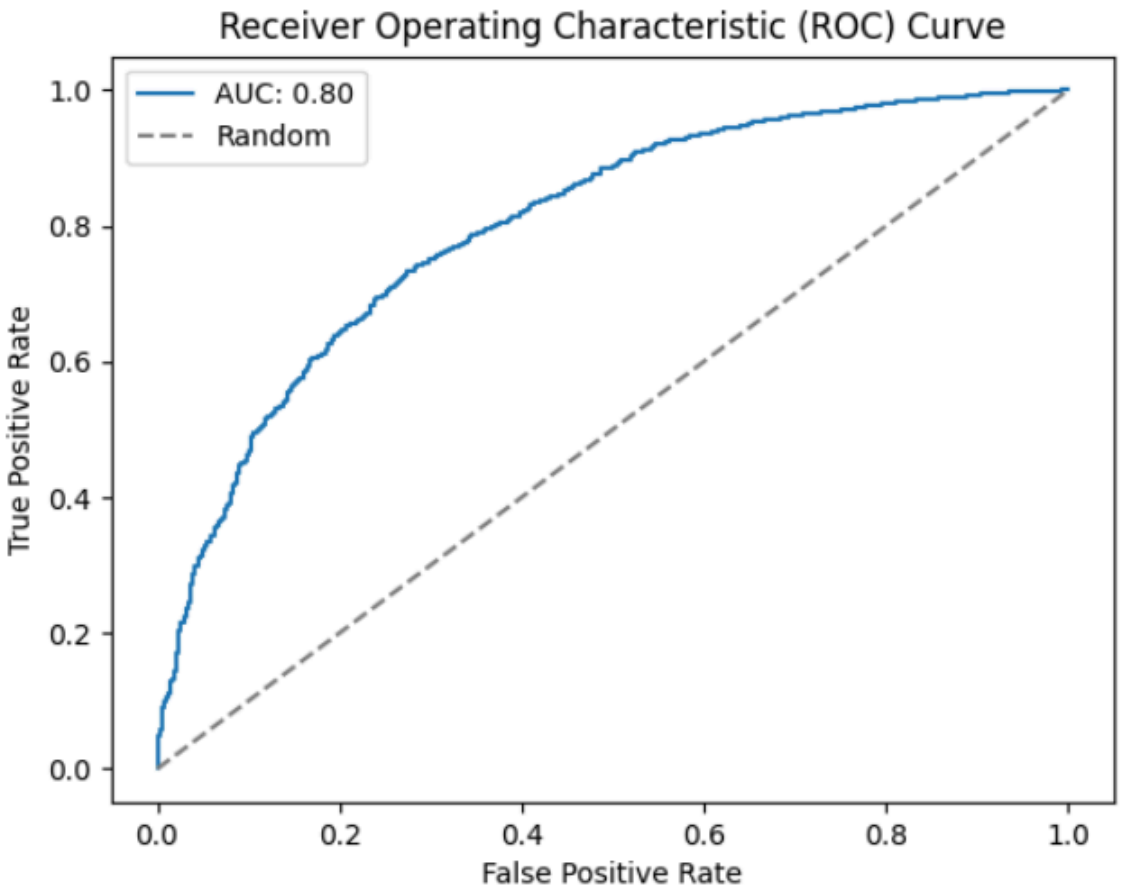
ResNet can predict Europe with ~75% accuracy.



VGG - non cropped Images



The VGG model with non-cropped images has the best performance of all models we have evaluated. With an overall accuracy of 73.24% correct predictions.



7. Lessons learned

Transfer Learning is its own Science

One of the primary takeaways from our project on creating a machine learning model to distinguish between two continents, Asia and Europe, is the realization that transfer learning is a distinct and intricate field. While it holds immense potential for leveraging pre-trained models and adapting them to specific tasks, it requires a deep understanding of the source and target domains.

Compute Resources and Image Data are vital

Another critical lesson learned was the significance of compute resources and the availability of an extensive dataset. We faced limitations in both computational power and the quantity of images during the training phase. This hindered the model's ability to generalize effectively, resulting in suboptimal performance.

Overfitting Challenges with Complex Network Architectures

We also quickly realized the difficulty in avoiding overfitting when working with complex, layered architectures like our neural network models. While intricate models may demonstrate high accuracy on the training set, their generalization to unseen data becomes

a delicate balancing act. We encountered scenarios where the model exhibited excellent performance on the training data but struggled to maintain the same level of accuracy on validation and test sets.

8. Ideas for improvement

Places 365 Dataset

To improve the performance of our machine learning model, we propose the use of the Places365 dataset to filter the images to specific scenarios before training. The Places365 dataset contains a diverse collection of images annotated with scene categories. There are already some pre-trained CNN models available that we could make good use of our use case. Our approach would be to train our model only with images from an urban area and filter out images from a rural area or vice versa.

YOLO (You Only Look Once)

Incorporating the YOLO algorithm into our model could be a promising way to improve it. If we consider the use of YOLO for object recognition, we can expect precise identification and localization of key features in Google Street View images that are characteristic of specific continents, such as unique architectural styles, landmarks or street signs. The bounding box annotations generated by YOLO would make it possible to extract detailed spatial information about relevant objects and provide a wealth of fine-grained visual cues to augment our dataset in the future.

Capture Panorama

Another idea to improve our model would be to use 360-degree panorama images from Google Street View instead of individual screenshots. The challenge here is to obtain such data cost-effectively, as Google Street View charges money per API call. Exploring open-source street views or using crowdsourced platforms could be a cost-effective alternative. Once different 360-degree images are available, the model's training data would provide a more comprehensive representation, which could improve the accuracy of continence detection. Another challenge for such an extension would also be the additional computational cost needed for training.

9. Conclusion

While we did endure a fair share of struggles with this project and the problem we chose to apply ML on, we can take away some key learnings from the research as well as the practical application that went into this paper. Firstly, it is clear that image classification is far more challenging than just throwing a complex neural network at a small-scale dataset and expecting good results right away. This is especially true if the classification problem goes beyond the scope of the typical online tutorial examples, such as cats vs. dogs. This however is the reason behind our other learnings about working with different neural network architectures, utilizing pretrained weights for transfer learning, and applying regularization to combat the complex models overfitting our relatively small dataset. Our results with VGG16

also show that it does not always need an even deeper, and more complex network, as it yielded better results than expected while not needing a lot of epochs to achieve them.

Lastly, we acknowledge that while we did reach our goal of creating a classifier that can classify better than a dummy classifier, which would have been one that reached 50% given our two balanced classes, there are still a lot of things we could improve on. Mainly the size of the dataset could be increased and the data augmentation as well as pre-processing concerning picking out a specific image area of interest rather than the complete image could be beneficial to the overall performance.

10. Sources

Dataset and Tools
https://www.kaggle.com/datasets/ubitquitin/geolocation-geoguessr-images-50k/code
https://pypi.org/project/pycountry-convert/
https://www.kaggle.com/code/shuvostp/split-folders-for-train-test-val-split-of-images

Model architectures
CNNs:
https://www.ibm.com/topics/convolutional-neural-networks
https://developers.google.com/machine-learning/practica/image-classification/convolutional-neural-networks
https://towardsdatascience.com/using-convolutional-neural-network-for-image-classification-5997bfd0ede4
https://www.youtube.com/watch?v=jztwpsIzEGc&t=3458s
https://www.youtube.com/watch?v=J1jhfAw5Uvo&t=920s
https://www.youtube.com/watch?v=LKMi8Daf2ts

VGG:
https://keras.io/api/applications/vgg/
https://arxiv.org/abs/1409.1556
https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/
https://towardsdatascience.com/transfer-learning-with-vgg16-and-keras-50ea161580b4
https://www.kaggle.com/code/kasmithh/transfer-learning-using-keras-vgg-16
https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918
https://www.geeksforgeeks.org/vgg-16-cnn-model/

DenseNet:
https://keras.io/api/applications/densenet/
https://www.kaggle.com/datasets/pytorch/densenet121
https://arxiv.org/abs/1608.06993
https://bouzouitina-hamdi.medium.com/transfer-learning-with-keras-using-densenet121-fffc6bb0c233
https://www.kaggle.com/code/sujoykg/keras-densenet-with-regularization-fine-tuning
https://www.kaggle.com/code/xhlulu/densenet-transfer-learning-iwildcam-2019

ResNet50:
https://keras.io/api/applications/resnet/
https://arxiv.org/abs/1512.03385
https://medium.com/@nitishkundu1993/exploring-resnet50-an-in-depth-look-at-the-model-architecture-and-code-implementation-d8d8fa67e46f
https://medium.com/@kenneth.ca95/a-guide-to-transfer-learning-with-keras-using-resnet50-a81a4a28084b
https://www.kaggle.com/code/suniliitb96/tutorial-keras-transfer-learning-with-resnet50

https://hamdi-ghorbel78.medium.com/a-guide-to-transfer-learning-with-keras-using-resnet50-453934a7b7dc
https://www.kaggle.com/code/koshirosato/classification-of-small-datasets-using-resnet
https://datagen.tech/guides/computer-vision/resnet-50/
https://deci.ai/blog/resnet50-how-to-achieve-sota-accuracy-on-imagenet/

Other:
https://medium.com/nerd-for-tech/yolo-you-only-look-once-65ea86104c51
https://builtin.com/data-science/transfer-learning
http://places.csail.mit.edu/index.html