DEEP LEARNING ASSIGNMENT

FM9528A Banking Analytics

Word Count:2214

By:251253766

Objective
With this coursework, we aim to use LiDAR images of several neighborhoods in London, UK and try to predict the deprivation indices of those localities based on these images. The deprivation indices are [7]:

- Income: Measures the proportion of the population experiencing deprivation relating to low income.
- Employment: Measures the proportion of the working-age population in an area involuntarily excluded from the labour market.
- Education: Measures the lack of attainment and skills in the local population.
- Health: Measures the risk of premature death and the impairment of quality of life through poor physical or mental health.
- Crime: Measures the risk of personal and material victimization at the local level.
- Barriers to Housing: Measures the physical and financial accessibility of housing and local services.
- Living Environment: Measures the quality of both the indoor and outdoor local environment.

Approach
We have been provided with two sets of data
1. A dataset (csv format) that contains the geographic details of the neighborhoods
2. The corresponding LiDAR images of each of these neighborhoods

We will examine a couple of Deep Learning Models, viz, VGG16 and Renet50v2 to understand how well these images can predict deprivation indices. The index we will be exploring is **'Living Environment'.**

Hypothesis

LiDAR is a method for determining ranges (variable distance) by targeting an object with a laser and measuring the time for the reflected light to return to the receiver. LiDAR images are a freely available resource with multiple applications in various fields including but not limited to Agriculture, Archaeology, Geology and Climate Science, Remote Sensing, where it has been very successfully used with great results. While other forms of remote sensing and related imagery has been used in the domain of socio-economic analysis, LiDAR has not been explored in this domain before this research [1]

We use the Transfer Learning approach on models pre-trained on Imagenet weights. The images used in Imagenet are of a different type than aerial laser images, but we hope that our models will learn the pattern using the large number of images we are providing and will be able to predict the index to some degree of accuracy.

I am especially hopeful that we will have a decently low prediction error for Living Environment in comparison to the other indices because Living Environment can almost always be directly interpreted using the images of a particular area. LiDAR also captures some relevant information like the availability of green space, urban density which help in predicting the Living Environment. Previous work (Jean et al., Block et al.) on successfully predicting poverty using satellite imagery has been done [4,5] so it is possible to extend this work with LiDAR images.

My understanding is that the model prediction accuracy can be improved further if we use background geo-demographic information as input in addition to the images.

Deep Learning

This section describes the steps taken to predict the 'Living condition' of an area using the corresponding LiDAR image as input. We choose two pre-trained Convolutional Neural Network (CNN)from different families, viz. the VGG16 and the Resnet50 Version2.

Data:

We combine the CSV dataset and the LiDAR images into one single Pandas dataframe, and we split the dataframe into train and test set in a 70:30 ratio.

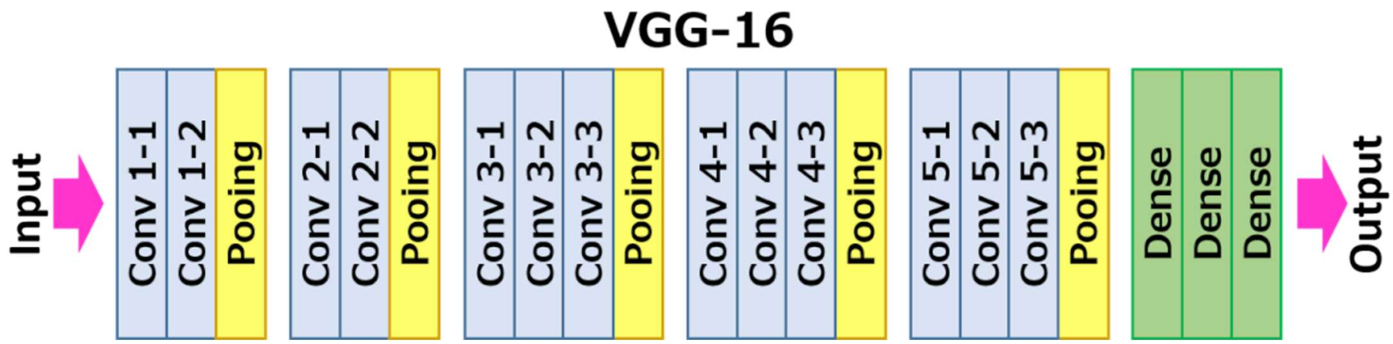
Train test split

```
# Create a train / test split
from sklearn.model_selection import train_test_split
train, test = train_test_split(DeprInd,
                               test_size = 0.3,
                               random_state = 251253766) # add student id as random state
```

Model Architecture

A. VGG16

VGG16 is a simple and widely used CNN Architecture used for ImageNet, a large visual database project used in visual object recognition software research [2]. VGG16 abbreviation for Visual Geometry Group, the group of researchers that developed this architecture, is named as such because it has 16 CNN layers. Below is the layer structure for VGG16.



VGG16 is a sequential model that has 16 stacked layers. Unlike earlier CNN models like AlexNet, VGG16 uses 3x3 sized kernels throughout the architecture. The final max-pooling layer is followed by 3 fully connected layers or dense layers and a final output layer with an activation function. This model was pre-trained on ImageNet data, and for this coursework, we will leverage these already trained weights and we follow the following steps to generate our model object:

a. We import the model VGG16 on the fly, but we do not want to include the top layers, so we set the option 'include_top = False'

b. We use the Imagenet weights as the weights for our model, but at this point the whole model is trainable. We do not want to train the whole model and will just customize the final few layers, so we copy our model object to a new model object and set everything to untrainable in this cloned copy.

c. We set the last two CNN layers as trainable, using the following code:

```
# Set layer as trainable.
CBModel.layers[15].trainable = True
CBModel.layers[16].trainable = True
```

d. We then add 2 dense layers with 64 neurons each and a final output layer with 1 neuron. The activation function added to each of these layers is **'relu'**.

e. We add 50% dropout in the dense layers to avoid overfitting, which neural networks are quite prone to do. The output layer also has Relu activation function, which sets all negative values to 0 which suits our needs because Living Environment has no negative values

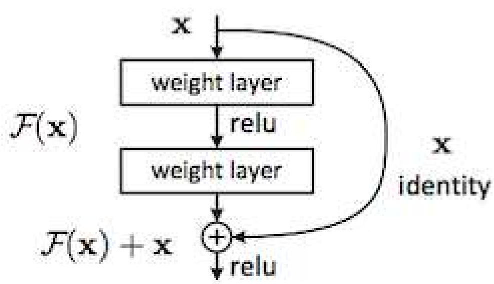
```
# We now add the new layers for prediction.
CBModel.add(Flatten(input_shape=model.output_shape[1:]))
CBModel.add(Dense(64, activation = 'relu'))
CBModel.add(Dropout(0.5))
CBModel.add(Dense(64, activation = 'relu'))
CBModel.add(Dropout(0.5))
CBModel.add(Dense(1, activation = 'relu')) # output is a linear function to predict living environment
```

f. With this architecture in place, we compile the model. We choose '**Adam**' as our optimizer, as it is an efficient, general-purpose optimizer that can be used for different applications. We set the learning rate low, at 1e-5, as we are using pre-trained models for the training to converge. Since we are predicting continuous variables ('living condition index'), we use **Mean Squared Error** as our loss function.

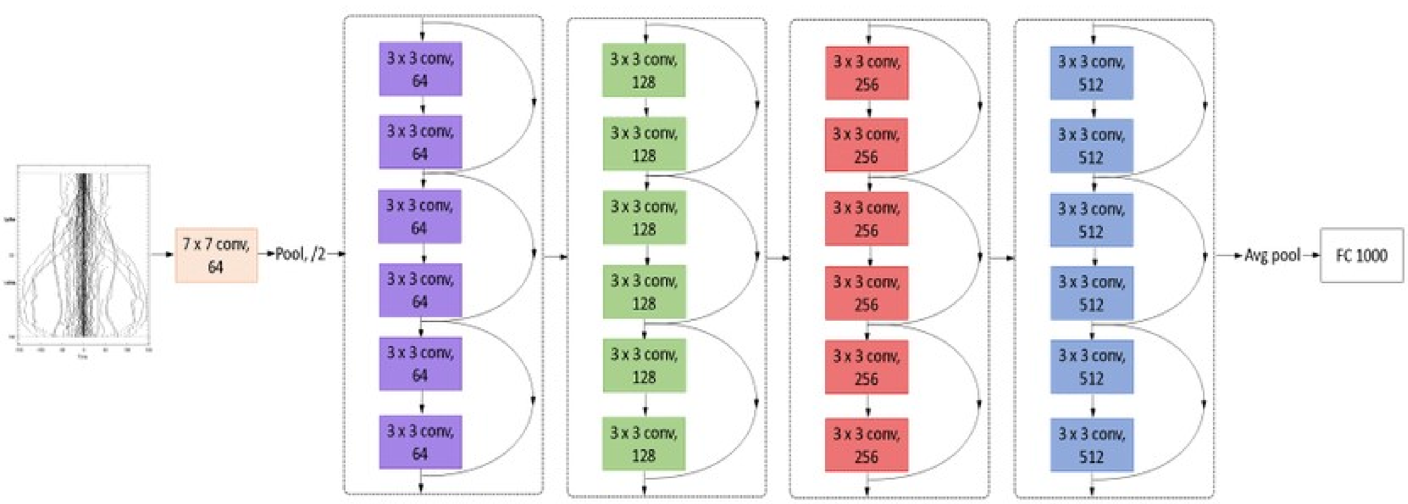
B. RESNET50V2

ResNet-50 model is a convolutional neural network (CNN) that is 50 layers deep. Resnet stacks residual blocks on top of each other to form a network. ResNet tackles two primary challenges faced by very deep neural nets- the problem of overfitting (and model complexity) and the problem of vanishing gradients. The vanishing gradient problem occurs when the backpropagation algorithm moves back through all the neurons of the neural net to update their weights [3]. The residual model is created by using skip connections, where the output of a particular layer (say layer A) is fed forward to a layer (say layer D) which is a few layers ahead of the current layer, skipping the layers in between (layers B and C). Layer D, therefore, gets input from layer C as well as layer A.

Residual layer:



Architecture:



Resnet comes with built-in batch normalization layers after every few CNN layers to avoid the data getting scaled up too much.

Resnet50 was pre-trained on Imagenet data, and we will use the Imagenet weights to train the model. We follow these steps:

a. The model is downloaded on the fly and saved as a base model, freezing all the weights and setting it as untrainable.
b. We then build the top layer by adding two dense layers with 64 neurons each and one output layer with one neuron. Each layer has the activation function **'relu'**.

```
# Input layer
inputs = keras.Input(shape=ImageSize + (3,),
                     name = 'image_only_input')

# Add the ResNet model, setting it to be untrainable.
# First we store it on a temporary variable.
x = base_model(inputs, training=False)

# Flatten to make it the same size as the original model
x = Flatten()(x)

# Now we actually add it to a layer. Note the way of writing it.
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)

# Add final output layer.
outputs = Dense(1, activation='relu')(x)

# Create the complete model object
ImageOnlyModel = keras.Model(inputs, outputs)
```

Model summary:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image_only_input (InputLaye  [(None, 224, 224, 3)]    0
 r)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten_1 (Flatten)         (None, 100352)            0

 dense_3 (Dense)             (None, 64)                6422592

 dropout_2 (Dropout)         (None, 64)                0

 dense_4 (Dense)             (None, 64)                4160

 dropout_3 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 1)                 65

=================================================================
Total params: 29,991,617
Trainable params: 6,426,817
Non-trainable params: 23,564,800
_________________________________________________________________
```

c. We now compile the model and as before, choose '**Adam**' as our optimizer, and set the learning rate at 1e-5. We choose **Mean Squared Error** as our loss function.

d. We create an Image Data Generator object (described in the next section) to add the images to the model in batches,
e. We do a warm start with two epochs.
f. Next, we set the base model as trainable. This way all its layers except the batch normalization layers will be set to trainable.

```
Model: "model"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image_only_input (InputLaye  [(None, 224, 224, 3)]    0
 r)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten_1 (Flatten)         (None, 100352)            0

 dense_3 (Dense)             (None, 64)                6422592

 dropout_2 (Dropout)         (None, 64)                0

 dense_4 (Dense)             (None, 64)                4160

 dropout_3 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 1)                 65

=================================================================
Total params: 29,991,617
Trainable params: 29,946,177
Non-trainable params: 45,440
_________________________________________________________________
```

g. We now add two callbacks and train the model.
EarlyStopping: This monitors the training and validation losses and automatically stops training the model once the validation error stays within 0.00001(tolerance) of the previous epoch flat for 3 epochs(patience).
Modelcheckpoint: Saves the weights of the best performing models. In case we need to retrain our model- we can always start by calling the last saved model and not have to start from scratch.

```
my_callbacks = [
    # Stop training if validation error stays within 0.00001 for three rounds.
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     min_delta=0.00001,
                                     patience=3),
    # Save the weights of the best performing model to the checkpoint folder.
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                       save_best_only=True,
                                       save_weights_only=True),
]

# Number of epochs
epochs = 20
```

Image Data Generation:

We will create Image Data Generators to be used for both our models, which take images from a directory, and feed them to the model as needed. The input size is always set to 224X224. We do data augmentation to improve the accuracy of prediction by making the models search more complex patterns. For VGG16, the generator has been set to rescale = 1./255 to normalize the inputs. For Resnet, the preprocessor normalizes the inputs, so we don’t rescale. For both models, we choose not to do any shearing, choose to zoom about 80-120%, as we think it might be beneficial to look more closely considering it is an aerial image and do both horizontal and vertical flips. We create a validation subset using 20% data from the train set.
The number of images in each set is:

```
Found 20565 validated image filenames.
Found 5141 validated image filenames.
Found 11017 validated image filenames.
```

VGG16:

```
train_datagen = ImageDataGenerator(
                                   rescale=1./255,
                                   shear_range=0,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   preprocessing_function=preprocess_input,
                                   validation_split = 0.2
                                   )

test_datagen = ImageDataGenerator(
                                   rescale=1./255,              #
                                   shear_range=0,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   preprocessing_function=preprocess_input,
                                   )
```

Resnet:

```python
# Define generators
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
                                   rescale=None,                    # I
                                   shear_range=0,                     #
                                   zoom_range=0.2,
                                   horizontal_flip=False,
                                   vertical_flip=False,
                                   preprocessing_function=preprocess_input,
                                   validation_split = 0.2
                                   )

test_datagen = ImageDataGenerator(
                                   rescale=None,                   # In
                                   shear_range=0,                     #
                                   zoom_range=0,                      #
                                   horizontal_flip=False,
                                   vertical_flip=False,
                                   preprocessing_function=preprocess_input,
                                   )
```

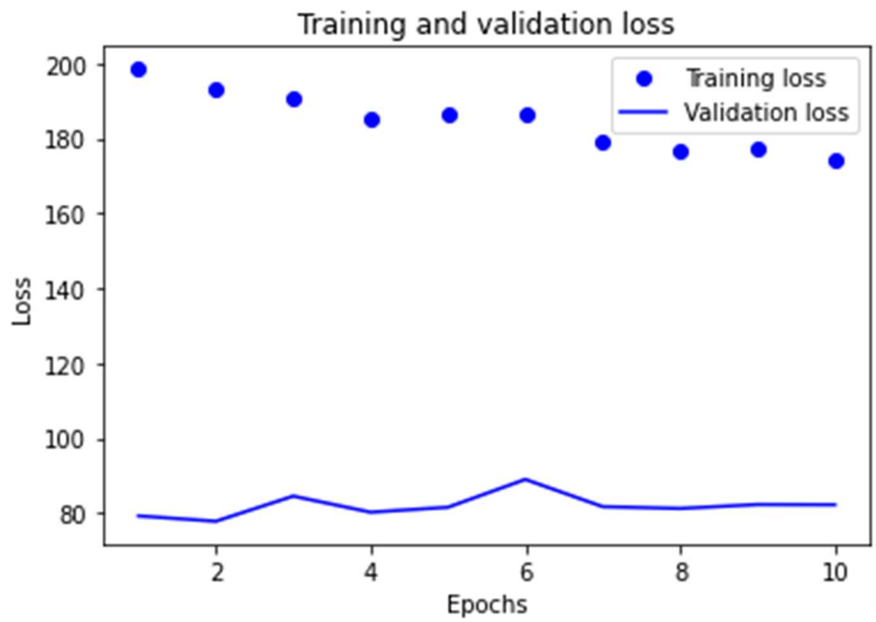
Training:

VGG16:

Here are the parameters chosen for training VGG:

```python
epochs = 10

# Train!
CBModel.fit(
            train_generator,
            epochs=epochs,
            validation_data=validation_generator,
            steps_per_epoch = train_generator.samples//train_generator.batch_size, # Usually ca
            validation_steps = validation_generator.samples//validation_generator.batch_size #
          )
```

We trained in batches of 5 epochs. The validation loss started to stabilize after about 18 epochs. We saw that it stayed steady for around 3 epochs, and we decided that this was the point where the model converged. The figure below shows the last 10 epochs.

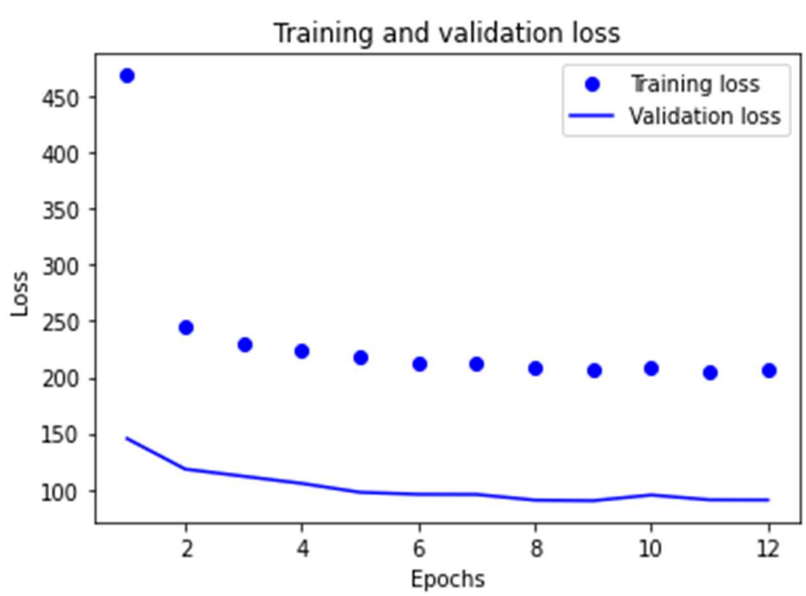


ResNet50:

Here, we kept the batch size 128, because Resnet is more memory intensive than VGG. Here are the parameters:

```
# Train!
ImageOnlyModel.fit(
                train_generator, # Pass the train generator
                epochs=epochs, # Pass the epochs
                validation_data=validation_generator, # Pass the validation generator
                steps_per_epoch = train_generator.samples//train_generator.batch_size,  # Usually cases / batch_size
                validation_steps = validation_generator.samples//validation_generator.batch_size,  # Number of validation steps. Again cases / batch_size
                callbacks=my_callbacks # Add the callbacks
                )
```

We started out wanting to train the model for 20 epochs. After the 12th epoch, however, the Callback terminated training since the model converged. We found that validation loss was the lowest for the 9th epoch, so we chose that as our optimum model.



Testing:

We tested our best performing models on the test set to predict the 'Living_Environment'. We use **Mean Squared Error**, **Root Mean Squared Error** and **Normalized Mean Squared Error** as our metrics to measure performance because we are tackling a regression problem. The latter will be helpful to compare the performance of the model for different indices (e.g, crime, education) and understand which index is best predicted using these models.

Model	Training Loss(MSE)	Validation Loss(MSE)	Testing Error(MSE)	Testing Error (RMSE)	Normalised RMSE(test)
VGG16	174.4736	82.1808	165.84	12.88	0.149
Resnet	206.1017	89.9827	155.33	12.46	0.145

We conclude that for our data, Resnet50V2 is performing marginally better than VGG and is giving us lower test errors.

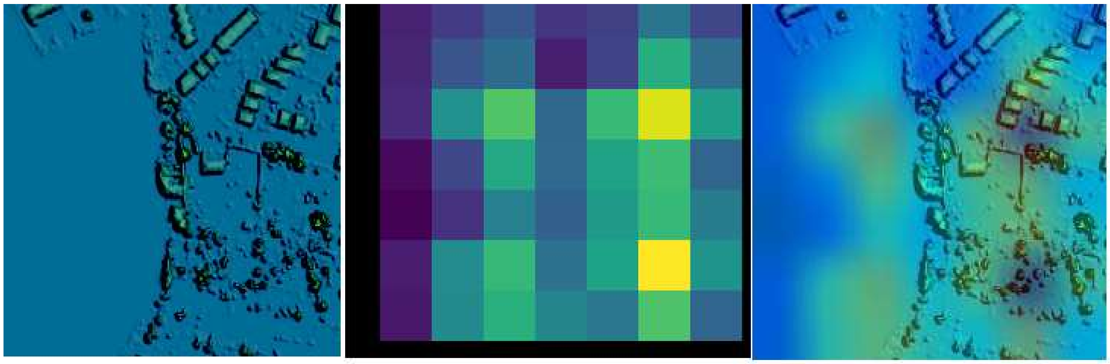
Interpretation using GradCAM

GradCAM(Gradient Class Activation Mapping) is a technique to introduce explainability in our NN models. It allows us to understand where our models are focusing and how they are interpreting the information in the images.

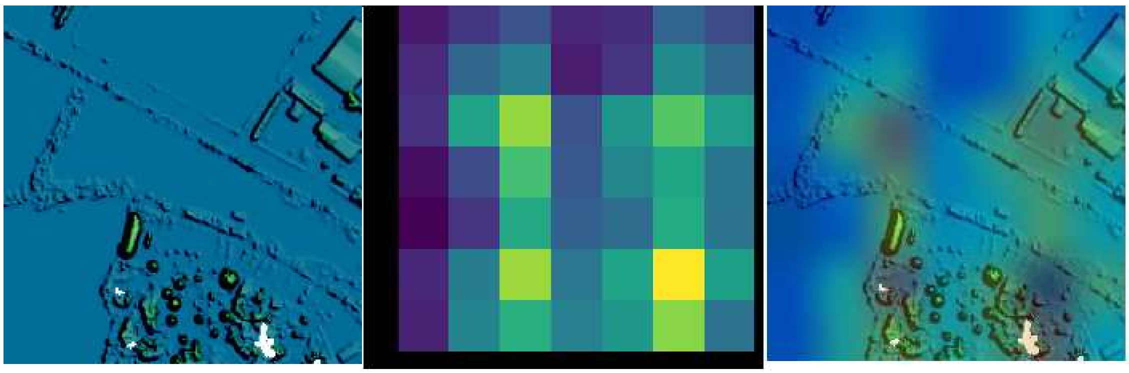
We clone the last convolutional layer of our model and add the top layers from our model and let's say we name it regression model. We then calculate the gradient of the output of the regression model (predicted value of deprivation index) with respect to the feature map of our last convolutional layer. We take the mean intensity of the gradient over a specific feature map channel, multiply each channel with that weight, and then take the channel wise mean to derive the final heatmap.
The heatmap tells us exactly which features or areas of the image the model gives the highest weights to and considers important.

Following are 10 images from the overall range of the index 'Living Environment'. We have provided below the original LiDAR image, the heatmap generated by GradCam and the heatmap superimposed on the actual image to understand what parts of the images the model has focused on. The GradCAM was run on Resnet, our better performing model on that model, focusing on the final convolutional layer.

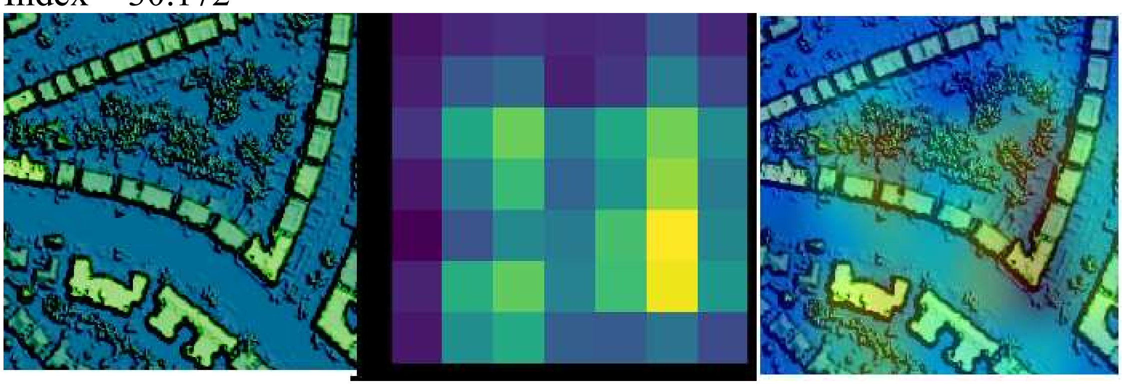
Index = 5.45



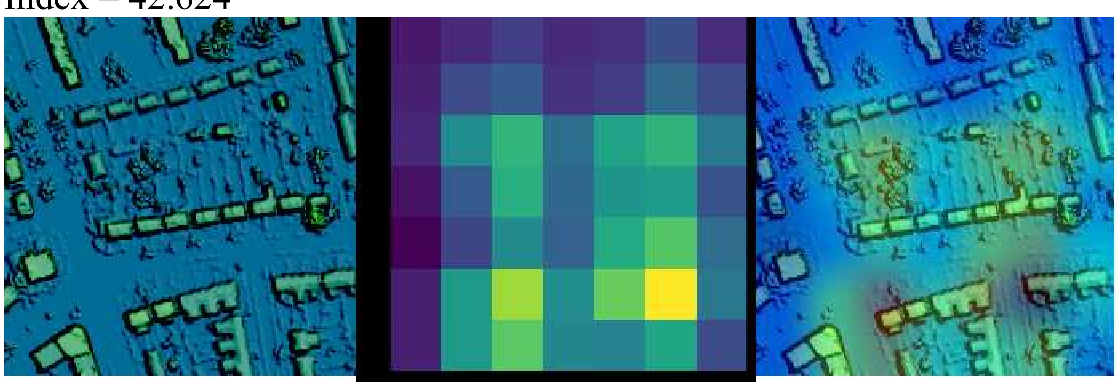
Index = 16.13



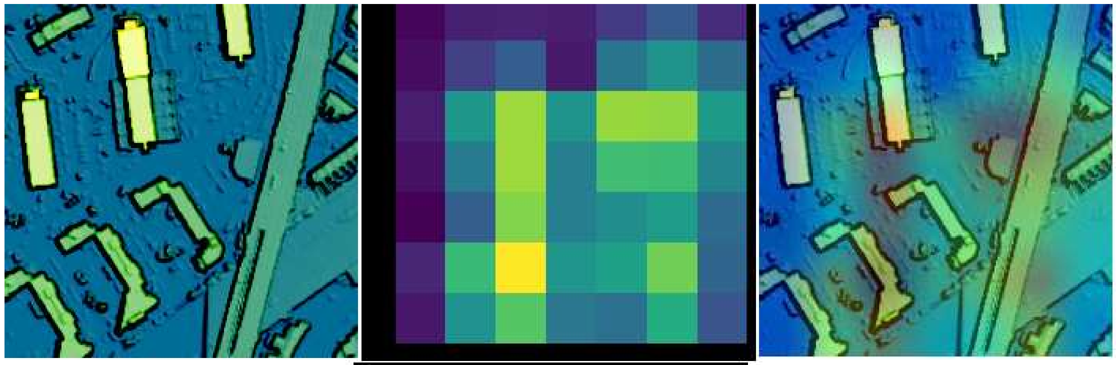
Index = 30.172



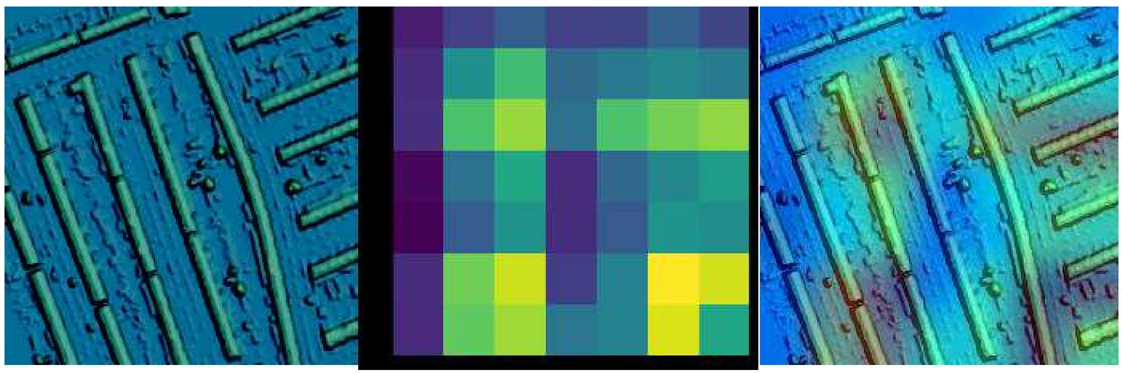
Index = 42.624



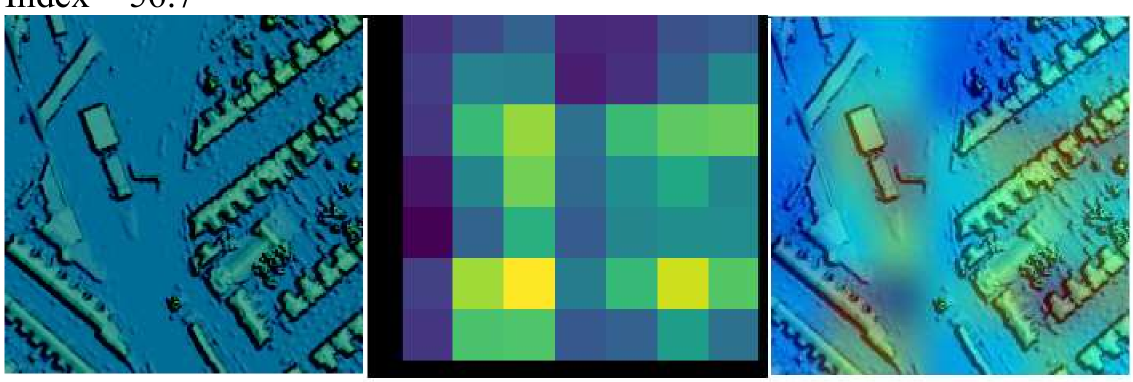
Index = 47.933



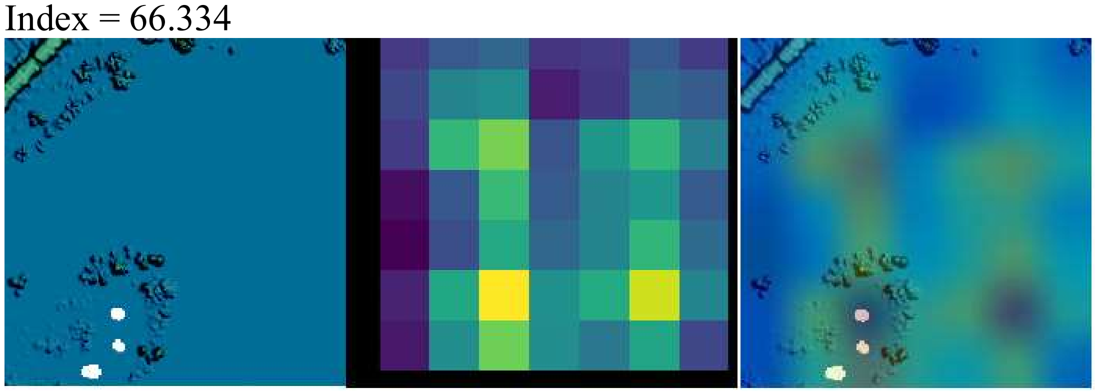
Index = 52.091



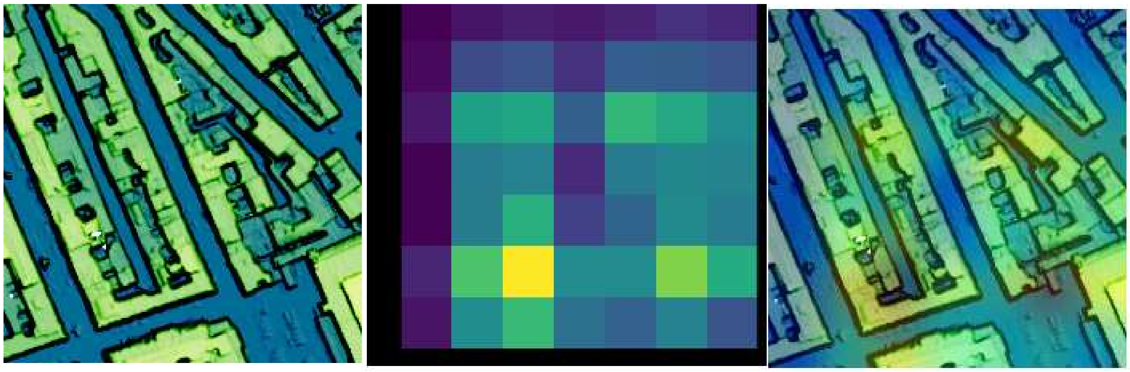
Index = 56.7



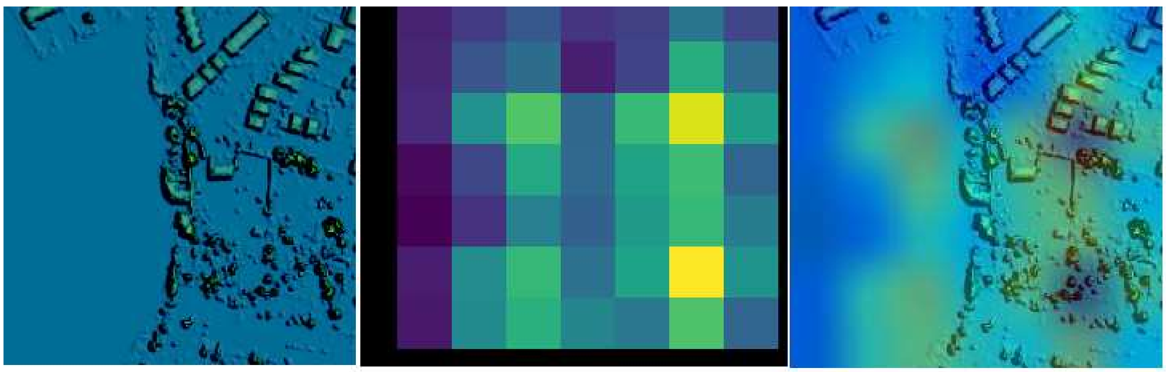
Index = 66.334



Index = 79.648



Index = 91.602



The model appears to focus on areas with higher building concentration, and images with high building density have been predicted to have a higher value of Living environment, which means lower quality of the said index.

The conclusion is not too off base, considering higher building density means less availability of open space. The caveat here is, these pictures are captured aerially, from a considerable height, so it is difficult to capture the indoor living environment. To achieve higher accuracy and lesser bias, the models should consider taking the background geo-demographic information as input in addition to the LiDAR images.

Ethical implications of using LiDAR images in socio-economic predictions

When we talk about ethical considerations, we often think about privacy, security, bias and fairness amongst others. Aerial LiDAR does not collect PII data [8], so individual privacy might not be the biggest concern here. There is however some concern regarding the models learning the bias based on the data collected unless the models are trained with data that fit the socio-demographic context of a particular area.

While some interpretations can be correct based on the images, some, for example, crime rates can be biased interpretation based on the diversity and representation of the sample. For example- are all densely populated areas crime-prone? This is where we need to be mindful of data collection bias.

For example, in many studies about predicting poverty, ownership of cars is one of the predictor variables. While this information might be able to correctly predict the living condition of a community in the suburb of an industrialized nation, it is hardly a good predictor when it comes to heavily urban communities or comparatively less industrialized economies.

LiDAR images find wide usage in various disciplines including but not limited to geology, archaeology, climate sciences, and as we saw above in the coursework can be a quite promising tool to predict deprivation indices. Considering that it is open data, we need to start thinking about who might use the data and to what end.

I think we can all agree that data, especially big data, is an incredibly powerful tool, and in the wrong hands can turn into a weapon of destruction. We need to think about implementing changes that give more agency to the individual, enforce the practice of informed consent, minimize the collection of personal information while still collecting overall community information, and bring transparency in every step of the process, from data collection, to use to storage policies.

References:

1. Bravo, C et al.,2021.Deep residential representations: Using unsupervised learning to unlock elevation data for geo-demographic prediction
2. Tinsy John Perumanoor.What is VGG16? Introduction to VGG16” https://medium.com/@mygreatlearning/what-is-vgg16-introduction-to-vgg16-f2d63849f615
3. Nick McCullum.The Vanishing Gradient Problem in Recurrent Neural Networks https://nickmccullum.com/python-deep-learning/vanishing-gradient-problem/#what-is-the-vanishing-gradient-problem
4. Jean, N., Burke, M., Xie, M., Davis, W.M., Lobell, D.B., Ermon, S., 2016. Combining satellite imagery and machine learning to predict poverty.
5. J. Block et al., "An Unsupervised Deep Learning Approach for Satellite Image Analysis with Applications in Demographic Analysis," 2017 IEEE 13th International Conference on e-Science (e-Science), 2017, pp. 9-18, doi: 10.1109/eScience.2017.13.
6. Panchal, Shubham,2021. Grad-CAM: A Camera For Your Model’s Decision

7. “ English indices of deprivation” https://www.gov.uk/government/collections/english-indices-of-deprivation

8. https://www.thefastmode.com/expert-opinion/19182-lidar-is-playing-a-leading-role-in-the-development-of-smart-cities

Appendix:

1. Link to Collab Notebook:
https://colab.research.google.com/drive/1zKzROi17Y0kkfzcBEOkWa-I9UoK_w44E#scrollTo=K24odNK20iqJ

2. Code for the coursework:

We start by importing Keras

```
import numpy as np
import h5py as h5py
import PIL
import tensorflow as tf

# Others
import numpy as np
from sklearn.model_selection import train_test_split
import pandas as pd


# For AUC estimation and ROC plots
from sklearn.metrics import roc_curve, auc


# Image and directories
import cv2
import os

#tensorflow and keras
import tensorflow.keras as keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow import keras
# Plots
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
!wget·"https://uwoca-my.sharepoint.com/:u:/g/personal/cbravoro_uwo_ca/Ea8hL1Qqz-1DqXPUkFg3_OkBkT_oOJ5EdvwX1YU_afWF1w?downloa
!mv·/content/Ea8hL1Qqz-1DqXPUkFg3_OkBkT_oOJ5EdvwX1YU_afWF1w?download=1·/content/drive/MyDrive/data.tar.gz
!tar xvzf /content/drive/MyDrive/data.tar.gz
```

Reading the Deprivation index file

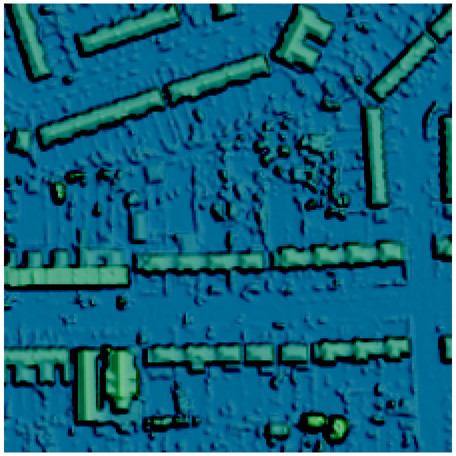
```
DeprInd = pd.read_csv('/content/EmbeddingData_C3_9528.csv')
DeprInd.describe()
```

	id	income	employment	education	health	crime	barriers	living_environme
count	36723.000000	36723.000000	36723.000000	36723.000000	36723.000000	36723.000000	36723.000000	36723.0000
mean	36922.291997	0.117011	0.077774	12.995546	-0.578348	0.211408	32.301551	25.6455
std	16424.963636	0.072170	0.043844	10.240666	0.718902	0.588681	10.587134	10.7296
min	2619.000000	0.006000	0.003000	0.013000	-3.215000	-2.354000	6.910000	5.4500
25%	24161.500000	0.057000	0.043000	4.481000	-1.089000	-0.199000	24.283000	17.5270
50%	37072.000000	0.103000	0.067000	10.925000	-0.563000	0.221000	31.270000	24.3500
75%	50001.500000	0.163000	0.102000	19.086000	-0.039000	0.627000	39.506000	31.9970
max	74739.000000	0.437000	0.317000	58.976000	1.570000	2.377000	70.456000	91.6020

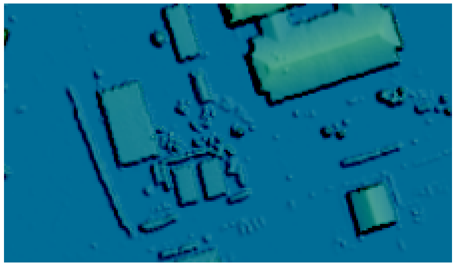
```
DeprInd.head()
```

	id	LSOA11CD	LSOA11NM	SOAC11CD	SOAC11NM	MSOA11CD	MSOA11NM	LAD17CD	LAD17NM	LACCD	LAC
0	48552	E01000759	Bromley 034A	8a	Affluent communities	E02000160	Bromley 034	E09000006	Bromley	1a1r	Rural-Urb Frin
1	46571	E01000759	Bromley 034A	8a	Affluent communities	E02000160	Bromley 034	E09000006	Bromley	1a1r	Rural-Urb Frin
2	21161	E01000487	Brent 006E	7b	Young ethnic communities	E02000098	Brent 006	E09000005	Brent	4a1r	Ethnica Diver Metropolit Livi

```
from IPython.display import Image
#Image(filename='/content/LIDAR/LIDAR_0.png')
Image(filename='/content/LIDAR/LIDAR_46276.png')
```



```
Image(filename='/content/LIDAR/LIDAR_10010.png')
```



```
# # Image Parameters
# ImageSize = (224,224)
# BatchSize = 128
```

Flow from dataframe- combine LIDAR images with the deprivation index dataset

```
ImagePath = '/content/LIDAR/'
DeprInd['path'] = [os.path.join(ImagePath + 'LIDAR_'+ str(i) + '.png') for i in DeprInd.id]
DeprInd.head()
```

```
DeprInd.shape
```

```
(36723, 19)
```

```
DeprInd.to_csv("finalfile.csv",index=False)
```

Train test split

```
# Create a train / test split
from sklearn.model_selection import train_test_split
train, test = train_test_split(DeprInd,
                               test_size = 0.3,
                               random_state = 251253766) # add student id as random state
```

```
print(train.shape)
print(test.shape)
```

```
(25706, 19)
(11017, 19)
```

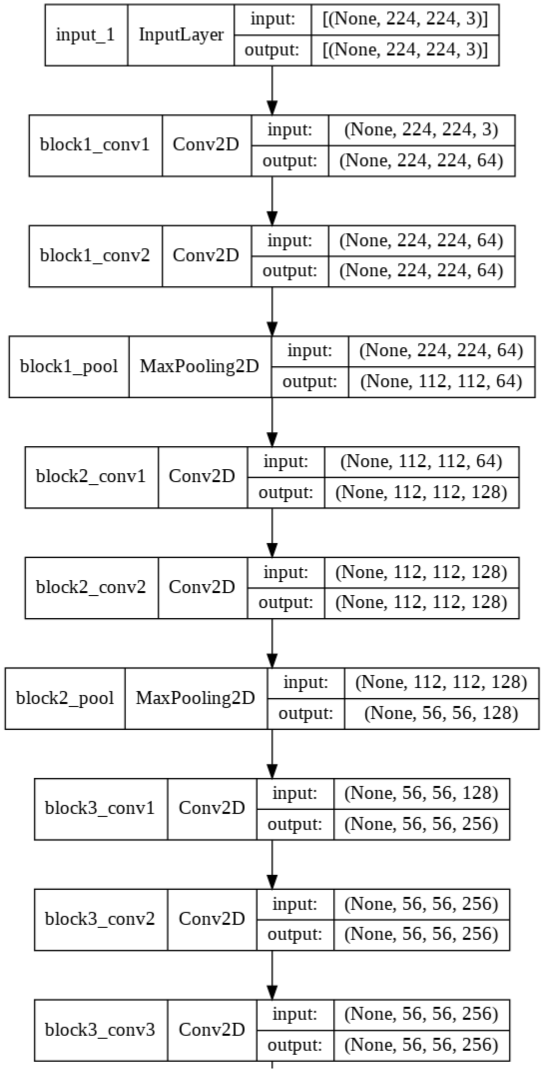
Double-click (or enter) to edit

Create base VGG16 model

```
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
model = VGG16(weights = 'imagenet',      # The weights from the ImageNet competition
              include_top = False,       # Do not include the top layer, which classifies.
              input_shape= (224, 224, 3) # Input shape. Three channels, and BGR
             )
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_
58892288/58889256 [==============================] - 0s 0us/step
58900480/58889256 [==============================] - 0s 0us/step
```

```
from tensorflow.keras.utils import plot_model
from IPython.display import Image

plot_model(model, show_shapes=True, show_layer_names=True, to_file='GraphModel.png')
Image(retina=True, filename='GraphModel.png')
```



Creating model with untrinable layers and new dense layers

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
```

```
# Create new model
```

```
CBModel = Sequential()

# Copy the layers to our new model. This needs to be done as there is a bug in Keras.
for layer in model.layers:
    CBModel.add(layer)

# Set the layers as untrainable
for layer in CBModel.layers:
    layer.trainable = False

# Set layer as trainable.
CBModel.layers[15].trainable = True
CBModel.layers[16].trainable = True
```

block5_pool | MaxPooling2D | input: | (None, 14, 14, 512)

```
# We now add the new layers for prediction.
CBModel.add(Flatten(input_shape=model.output_shape[1:]))
CBModel.add(Dense(64, activation = 'relu'))
CBModel.add(Dropout(0.5))
CBModel.add(Dense(64, activation = 'relu'))
CBModel.add(Dropout(0.5))
CBModel.add(Dense(1, activation = 'relu')) # output is a linear function to predict living environment


# What does the model look like?
CBModel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584
```

```
 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 64)                1605696

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 64)                4160

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 16,324,609
Trainable params: 6,329,537
```

```
Non-trainable params: 9,995,072
_________________________________________________________________
```

Compiling the model

```
# Compiling the model!
#import tensorflow.keras as keras

opt = optimizers.Adam(learning_rate=1e-5,            # Learning rate needs to be tweaked for convergence and be small!
                      decay=1e-3 / 200    # Decay of the LR 10^-3 / 1 / 50 / 100 / 200
                      )
CBModel.compile(loss=keras.losses.MeanSquaredError(), # Regression Loss
                optimizer=opt,
              )
```

Image data generation- VGG16

```
# prepare data augmentation configuration. One for train, one for test.
target_size = (224, 224)
batch_size = 256
DataDir = '/content/LIDAR/'


# Define generators
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
                                  rescale=1./255,                         # Inputs are scaled in the preprocessing function
                                  shear_range=0,                          # Shear?
                                  zoom_range=0.2,                           # Zoom? 0.2 means from 80% to 120%
                                  horizontal_flip=True,                     # Flip horizontally?
                                  vertical_flip=True,                      # Flip vertically?
                                  preprocessing_function=preprocess_input,  # VGG expects specific input. Set it up with thi
                                  validation_split = 0.2                    # Create a validation cut?
```

```
                                  )

    test_datagen = ImageDataGenerator(
                                  rescale=1./255,                        # Inputs are scaled in the preprocessing function
                                  shear_range=0,                          # Shear?
                                  zoom_range=0.2,                           # Zoom? 0.2 means from 80% to 120%
                                  horizontal_flip=True,                     # Flip horizontally?
                                  vertical_flip=True,                      # Flip vertically?
                                  preprocessing_function=preprocess_input,  # VGG expects specific input. Set it up with thi:
                                  )

    # Point to the data and **give the targets**. Note the "raw" class_mode
    train_generator = train_datagen.flow_from_dataframe(train,
                                                    directory='.',  # Look from root directory
                                                    x_col='path',   # Path to images
                                                    y_col='living_environment',  # Target
                                                    target_size=target_size, # Same as last lab
                                                    batch_size=batch_size,
                                                    shuffle=True,
                                                    class_mode='raw',
                                                    subset='training',
                                                    interpolation="bilinear"
                                                   )

    validation_generator = train_datagen.flow_from_dataframe(train,
                                                    directory='.',
                                                    x_col='path',
                                                    y_col='living_environment',
                                                    target_size=target_size,
                                                    batch_size=batch_size,
                                                    shuffle=True,
                                                    class_mode='raw',
                                                    subset='validation',
                                                    interpolation="bilinear"
                                                   )

    test_generator = test_datagen.flow_from_dataframe(test,
                                                  directory='.',
```

```
                                                  x_col='path',
                                                  y_col='living_environment',
                                                  target_size=target_size,
                                                  batch_size=batch_size,
                                                  shuffle=False,
                                                  class_mode='raw',
                                                  interpolation="bilinear"
                                                  )
```

```
Found 20565 validated image filenames.
Found 5141 validated image filenames.
Found 11017 validated image filenames.
```

```
train_generator.samples//train_generator.batch_size
validation_generator.samples//validation_generator.batch_size
# np.amax([validation_generator.samples // validation_generator.batch_size, 1])
```

```
20
```

Loading the data file and splitting into train and test

Training

Priming the model with 2 epochs.

```
# Number of epochs
```

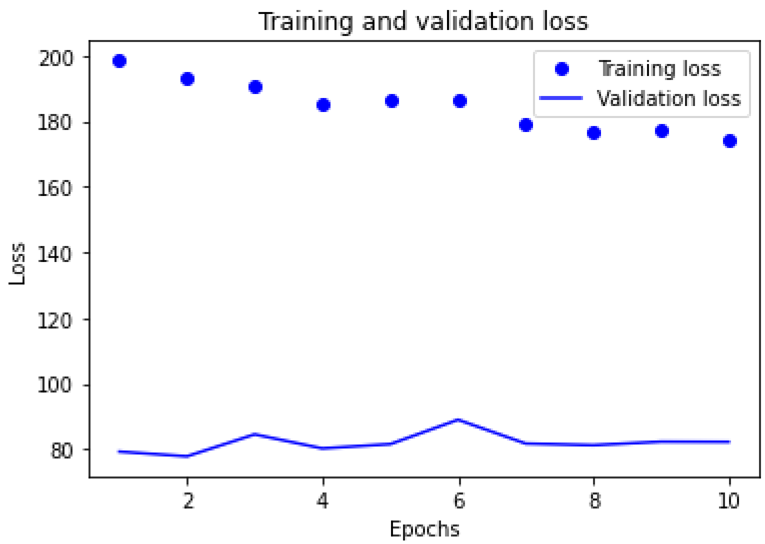
```
epochs = 10

# Train!
CBModel.fit(
            train_generator,
            epochs=epochs,
            validation_data=validation_generator,
            steps_per_epoch = train_generator.samples//train_generator.batch_size, # Usually cases / batch_size = 160.
            validation_steps = validation_generator.samples//validation_generator.batch_size # Number of validation steps. A
          )
```

```
Epoch 1/10
80/80 [==============================] - 316s 4s/step - loss: 198.5642 - val_loss: 79.1979
Epoch 2/10
80/80 [==============================] - 305s 4s/step - loss: 193.5176 - val_loss: 77.7606
Epoch 3/10
80/80 [==============================] - 304s 4s/step - loss: 191.0668 - val_loss: 84.4767
Epoch 4/10
80/80 [==============================] - 303s 4s/step - loss: 185.1550 - val_loss: 80.1827
Epoch 5/10
80/80 [==============================] - 304s 4s/step - loss: 186.3592 - val_loss: 81.4977
Epoch 6/10
80/80 [==============================] - 304s 4s/step - loss: 186.7633 - val_loss: 88.9467
Epoch 7/10
80/80 [==============================] - 303s 4s/step - loss: 179.1846 - val_loss: 81.6659
Epoch 8/10
80/80 [==============================] - 304s 4s/step - loss: 176.4626 - val_loss: 81.1754
Epoch 9/10
80/80 [==============================] - 304s 4s/step - loss: 177.0326 - val_loss: 82.2359
Epoch 10/10
80/80 [==============================] - 313s 4s/step - loss: 174.4736 - val_loss: 82.1808
<keras.callbacks.History at 0x7fca5e13f850>
```

Checking the convergence plot before actual training

```
loss = CBModel.history.history['loss']
val_loss = CBModel.history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
```

```
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



To train the non dense layers

Saving model

```
# Activating Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remou
```

```
# Saving the model
CBModel.save('/content/drive/MyDrive/VGG16_FM9528A_V1.h5')
```

```
#Loading
CBModel = keras.models.load_model('/content/drive/MyDrive/VGG16_FM9528A_V1.h5')
```

Restoring the best model

```
# Applying to the test set with a generator.
test_generator.reset()

# Get predictions
output_vgg = CBModel.predict(test_generator)
```

```
output_vgg
```

```
array([[21.389135],
       [42.07809 ],
       [20.046844],
       ...,
       [19.769209],
       [20.185917],
       [28.400835]], dtype=float32)
```

Double-click (or enter) to edit

```
test_generator.labels
```

```
array([11.187, 48.228, 11.657, ..., 15.814, 11.387, 22.131])
```

Root Mean Squared Error

```
# Root Mean Squared Error
def root_mean_squared_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.sqrt(np.mean((y_true - y_pred)**2))
```

```
def mean_squared_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean((y_true - y_pred)**2)
```

```
rmse_vgg = root_mean_squared_error(test_generator.labels, output_vgg)
print('The root mean squared error over the test for VGG model is %.2f' % rmse_vgg)
mse_vgg = mean_squared_error(test_generator.labels, output_vgg)
print('The mean squared error over the test for VGG model is %.2f' % mse_vgg)
```

```
The root mean squared error over the test for VGG model is 12.88
The mean squared error over the test for VGG model is 165.84
```

Resnet

```
# Import base model. Using ResNet50v2.
from tensorflow.keras.applications.resnet_v2 import ResNet50V2, preprocess_input
```

```
# Import model with input layer
base_model = ResNet50V2(weights = 'imagenet',     # The weights from the ImageNet competition
                        include_top = False,       # Do not include the top layer, which classifies.
                        input_shape= (224, 224, 3) # Input shape. Three channels.
                        )
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50v2_weights_tf_dim_orc
```

```
94674944/94668760 [==============================] - 2s 0us/step
94683136/94668760 [==============================] - 2s 0us/step
```

```
# Parameters
ImageSize = (224,224)
BatchSize = 128


# Set the base model to untrainable.
base_model.trainable = False


# Create the full model using the Model API

# Input layer
inputs = keras.Input(shape=ImageSize + (3,),
                        name = 'image_only_input')

# Add the ResNet model, setting it to be untrainable.
# First we store it on a temporary variable.
x = base_model(inputs, training=False)

# Flatten to make it the same size as the original model
x = Flatten()(x)

# Now we actually add it to a layer. Note the way of writing it.
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)

# Add final output layer.
outputs = Dense(1, activation='relu')(x)

# Create the complete model object
ImageOnlyModel = keras.Model(inputs, outputs)
```

```
# This is what the model looks like now.
ImageOnlyModel.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image_only_input (InputLaye  [(None, 224, 224, 3)]    0
 r)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten (Flatten)           (None, 100352)            0

 dense (Dense)               (None, 64)                6422592

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 64)                4160

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 29,991,617
Trainable params: 6,426,817
Non-trainable params: 23,564,800
_________________________________________________________________
```

```
base_model.summary()
```

```
 '')

 conv5_block2_2_pad (ZeroPaddin  (None, 9, 9, 512)   0           ['conv5_block2_1_relu[0][0]']
 g2D)

 conv5_block2_2_conv (Conv2D)   (None, 7, 7, 512)    2359296     ['conv5_block2_2_pad[0][0]']

 conv5_block2_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block2_2_conv[0][0]']
 ization)
```

```
conv5_block2_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block2_2_bn[0][0]']
n)

conv5_block2_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block2_2_relu[0][0]']

conv5_block2_out (Add)         (None, 7, 7, 2048)   0           ['conv5_block1_out[0][0]',
                                                                 'conv5_block2_3_conv[0][0]']

conv5_block3_preact_bn (BatchN  (None, 7, 7, 2048)  8192        ['conv5_block2_out[0][0]']
ormalization)

conv5_block3_preact_relu (Acti  (None, 7, 7, 2048)  0           ['conv5_block3_preact_bn[0][0]']
vation)

conv5_block3_1_conv (Conv2D)   (None, 7, 7, 512)    1048576     ['conv5_block3_preact_relu[0][0]'
                                                                ]

conv5_block3_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_1_conv[0][0]']
ization)

conv5_block3_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_1_bn[0][0]']
n)

conv5_block3_2_pad (ZeroPaddin  (None, 9, 9, 512)   0           ['conv5_block3_1_relu[0][0]']
g2D)

conv5_block3_2_conv (Conv2D)   (None, 7, 7, 512)    2359296     ['conv5_block3_2_pad[0][0]']

conv5_block3_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_2_conv[0][0]']
ization)

conv5_block3_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_2_bn[0][0]']
n)

conv5_block3_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block3_2_relu[0][0]']

conv5_block3_out (Add)         (None, 7, 7, 2048)   0           ['conv5_block2_out[0][0]',
                                                                 'conv5_block3_3_conv[0][0]']

post_bn (BatchNormalization)   (None, 7, 7, 2048)   8192        ['conv5_block3_out[0][0]']

post_relu (Activation)         (None, 7, 7, 2048)   0           ['post_bn[0][0]']
```

```
==================================================================================================
Total params: 23,564,800
Trainable params: 0
Non-trainable params: 23,564,800
__________________________________________________________________________________________________
```

```
# Compiling the model! Note the learning rate.
opt = optimizers.Adam(learning_rate=1e-6,            # Learning rate needs to be tweaked for convergence and be small!
                      decay=1e-3 / 200    # Decay of the LR 10^-3 / 1 / 50 / 100 / 200
                      )
ImageOnlyModel.compile(loss=keras.losses.MeanSquaredError(), # This is NOT a classification problem!
                      optimizer=opt
                       )
```

Data generation for Resnet

```
# Define parameters

target_size = (224, 224)
batch_size = 128
DataDir = '/content/LIDAR/'

# Define generators
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
                                  rescale=None,                         # Inputs are scaled in the preprocessing function
                                  shear_range=0,                          # Shear?
                                  zoom_range=0.2,                           # Zoom? 0.2 means from 80% to 120%
                                  horizontal_flip=False,                     # Flip horizontally?
                                  vertical_flip=False,                      # Flip vertically?
                                  preprocessing_function=preprocess_input,  # ResNet expects specific input. Set it up with
                                  validation_split = 0.2                    # Create a validation cut?
                                  )

test_datagen = ImageDataGenerator(
```

```
                                  rescale=None,                        # Inputs are scaled in the preprocessing function
                                  shear_range=0,                          # Shear?
                                  zoom_range=0,                           # Zoom? 0.2 means from 80% to 120%
                                  horizontal_flip=False,                     # Flip horizontally?
                                  vertical_flip=False,                      # Flip vertically?
                                  preprocessing_function=preprocess_input,  # VGG expects specific input. Set it up with thi
                                  )

# Point to the data and **give the targets**. Note the "raw" class_mode
train_generator = train_datagen.flow_from_dataframe(train,
                                                    directory='.',  # Look from root directory
                                                    x_col='path',   # Path to images
                                                    y_col='living_environment',  # Target
                                                    target_size=target_size, # Same as last lab
                                                    batch_size=batch_size,
                                                    shuffle=True,
                                                    class_mode='raw',
                                                    subset='training',
                                                    interpolation="bilinear"
                                                   )

validation_generator = train_datagen.flow_from_dataframe(train,
                                                    directory='.',
                                                    x_col='path',
                                                    y_col='living_environment',
                                                    target_size=target_size,
                                                    batch_size=batch_size,
                                                    shuffle=True,
                                                    class_mode='raw',
                                                    subset='validation',
                                                    interpolation="bilinear"
                                                   )

test_generator = test_datagen.flow_from_dataframe(test,
                                                  directory='.',
                                                  x_col='path',
                                                  y_col='living_environment',
                                                  target_size=target_size,
```

```
                                            batch_size=batch_size,
                                            shuffle=False,
                                            class_mode='raw',
                                            interpolation="bilinear"
                                            )
```

```
Found 20565 validated image filenames.
Found 5141 validated image filenames.
Found 11017 validated image filenames.
```

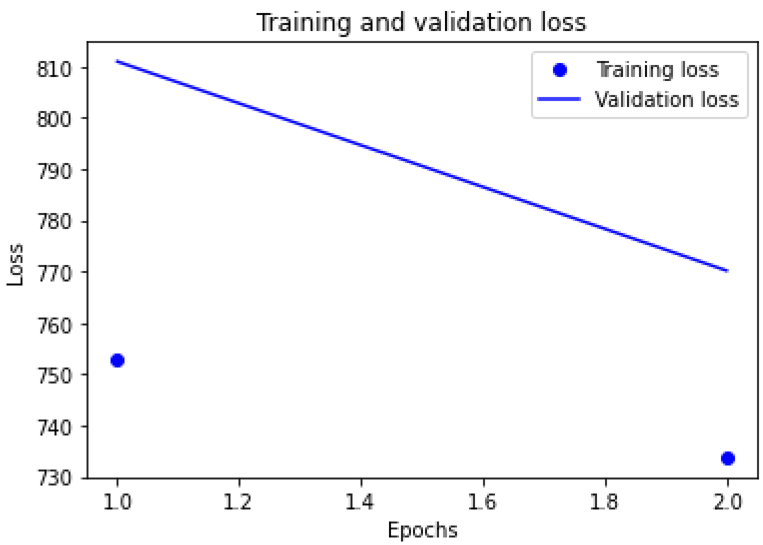
Warm start and priming of model

```
# Number of epochs
epochs = 2

# Train!
ImageOnlyModel.fit(
                  train_generator,
                  epochs=epochs,
                  validation_data=validation_generator,
                  steps_per_epoch = 3, # Usually cases / batch_size = 3.
                  validation_steps = 1 # Number of validation steps. Again cases / batch_size = 1.
                  )
```

```
Epoch 1/2
3/3 [==============================] - 52s 24s/step - loss: 752.8106 - val_loss: 811.0211
Epoch 2/2
3/3 [==============================] - 8s 3s/step - loss: 733.7498 - val_loss: 770.1473
<keras.callbacks.History at 0x7fc6022956d0>
```

```
# Plotting training history.
loss = ImageOnlyModel.history.history['loss']
val_loss = ImageOnlyModel.history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
```

```
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



We will now train non dense layers

```
base_model.trainable = True

# Recompile as we changed things.
ImageOnlyModel.compile(loss=keras.losses.MeanSquaredError(), # This is NOT a classification problem!
                      optimizer=opt
                       )

ImageOnlyModel.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image_only_input (InputLaye  [(None, 224, 224, 3)]    0
```

```
 r)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten (Flatten)           (None, 100352)            0

 dense (Dense)               (None, 64)                6422592

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 64)                4160

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 29,991,617
Trainable params: 29,946,177
Non-trainable params: 45,440
_________________________________________________________________
```

Call backs and model training

```
# Define callbacks
#checkpoint_path='/checkpoints/ImageOnlyModel.{epoch:02d}-{val_loss:.2f}.h5'
checkpoint_path='/content/drive/MyDrive/checkpoints/ImageOnlyModel.{epoch:02d}-{val_loss:.2f}.h5'

checkpoint_dir=os.path.dirname(checkpoint_path)
checkpoint_dir
```

```
'/content/drive/MyDrive/checkpoints'
```

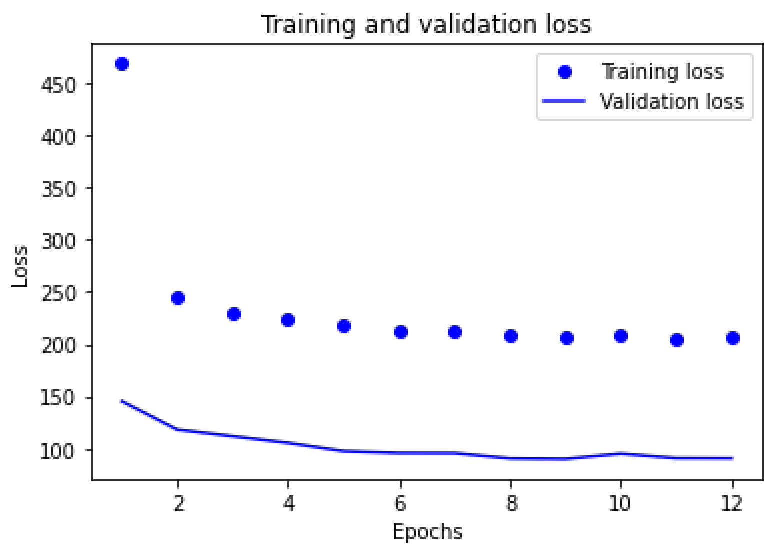
```
# Define callbacks
#checkpoint_path='checkpoints/ImageOnlyModel.{epoch:02d}-{val_loss:.2f}.h5'
checkpoint_path='/content/drive/MyDrive/checkpoints/ImageOnlyModel.{epoch:02d}-{val_loss:.2f}.h5'

checkpoint_dir=os.path.dirname(checkpoint_path)
```

```
my_callbacks = [
    # Stop training if validation error stays within 0.00001 for three rounds.
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     min_delta=0.00001,
                                     patience=3),
    # Save the weights of the best performing model to the checkpoint folder.
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                       save_best_only=True,
                                       save_weights_only=True),
]

# Number of epochs
epochs = 20

# Train!
ImageOnlyModel.fit(
                  train_generator, # Pass the train generator
                  epochs=epochs, # Pass the epochs
                  validation_data=validation_generator, # Pass the validation generator
                  steps_per_epoch = train_generator.samples//train_generator.batch_size,  # Usually cases / batch_size
                  validation_steps = validation_generator.samples//validation_generator.batch_size,  # Number of validation 
                  callbacks=my_callbacks # Add the callbacks
                  )
```

```
Epoch 1/20
160/160 [==============================] - 331s 2s/step - loss: 469.0233 - val_loss: 145.3185
Epoch 2/20
160/160 [==============================] - 325s 2s/step - loss: 245.2115 - val_loss: 118.0729
Epoch 3/20
160/160 [==============================] - 323s 2s/step - loss: 229.7025 - val_loss: 111.7264
Epoch 4/20
160/160 [==============================] - 323s 2s/step - loss: 224.7601 - val_loss: 105.3560
Epoch 5/20
160/160 [==============================] - 322s 2s/step - loss: 218.5129 - val_loss: 97.4457
Epoch 6/20
160/160 [==============================] - 323s 2s/step - loss: 211.7424 - val_loss: 95.6826
Epoch 7/20
160/160 [==============================] - 324s 2s/step - loss: 212.7580 - val_loss: 95.5370
Epoch 8/20
```

```
160/160 [==============================] - 326s 2s/step - loss: 207.9579 - val_loss: 90.4813
Epoch 9/20
160/160 [==============================] - 324s 2s/step - loss: 206.1017 - val_loss: 89.9827
Epoch 10/20
160/160 [==============================] - 325s 2s/step - loss: 209.2825 - val_loss: 95.0437
Epoch 11/20
160/160 [==============================] - 324s 2s/step - loss: 204.4840 - val_loss: 90.7474
Epoch 12/20
160/160 [==============================] - 326s 2s/step - loss: 207.7153 - val_loss: 90.6158
<keras.callbacks.History at 0x7fcadb652510>
```

```
# Plotting training history.
loss = ImageOnlyModel.history.history['loss']
val_loss = ImageOnlyModel.history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Restoring best model

```
# Load the weights. THIS REQUIRES FIRST CREATING THE LOGIC.
ImageOnlyModel.load_weights('/content/drive/MyDrive/checkpoints/ImageOnlyModel.09-89.98.h5') # replace this name with the be
```

Test prediction

```
# Applying to the test set with a generator.
test_generator.reset()

# Get probabilities
output = ImageOnlyModel.predict(test_generator)
```

```
test_generator
```

```
<keras.preprocessing.image.DataFrameIterator at 0x7fd84617d890>
```

```
import sys
#import numpy
np.set_printoptions(threshold=sys.maxsize)
```

```
display(output)
```

```
test_generator.labels
```

```
array([11.187, 48.228, 11.657, ..., 15.814, 11.387, 22.131])
```

```
rmse_resnet = root_mean_squared_error(test_generator.labels, output)
print('The root mean square error over the test for Resnetmodel is %.2f' % rmse_resnet)
mse_resnet = mean_squared_error(test_generator.labels, output)
print('The mean squared error over the test for VGG model is %.2f' % mse_resnet)
```

```
The root mean square error over the test for Resnetmodel is 12.46
The mean squared error over the test for VGG model is 155.33
```

```
max_test = np.max(test_generator.labels)
min_test = np.min(test_generator.labels)
#norm_rmse_vgg = rmse_vgg/(max_test-min_test)
norm_rmse_resnet = rmse_resnet/(max_test-min_test)
#print("normalised rmse for vgg over test is %.3f" % norm_rmse_vgg)
print("normalised rmse for resnet over test is %.3f" % norm_rmse_resnet)
```

```
normalised rmse for resnet over test is 0.145
```

```
base_model.summary()
```

```
ImageOnlyModel.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 image_only_input (InputLaye  [(None, 224, 224, 3)]    0
 r)

 resnet50v2 (Functional)     (None, 7, 7, 2048)        23564800

 flatten (Flatten)           (None, 100352)            0

 dense (Dense)               (None, 64)                6422592

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 64)                4160

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 1)                 65
```

```
=================================================================
Total params: 29,991,617
Trainable params: 29,946,177
Non-trainable params: 45,440
_________________________________________________________________
```

GradCam

```
ImageOnlyModel.layers[1].get_layer(last_conv_layer_name)
```

```
<keras.layers.convolutional.Conv2D at 0x7fd8467a0a50>
```

```
# The explainer. Gotten from https://keras.io/examples/vision/grad_cam/
def make_gradcam_heatmap(
    img_array, model, last_conv_layer_name, classifier_layer_names
):
    from tensorflow import keras
    import tensorflow as tf
    # First, we create a model that maps the input image to the activations
    # of the last conv layer. This layer is located at model.layers[1] as the
    # ResNet model is the first "layer" of the ImageOnlyModel. Modify as needed.

    last_conv_layer = model.layers[1].get_layer(last_conv_layer_name)
    last_conv_layer_model = keras.Model(model.layers[1].inputs, last_conv_layer.output)

    # last_conv_layer = base_model.get_layer(last_conv_layer_name)
    # last_conv_layer_model = keras.Model(base_model.inputs, last_conv_layer.output)

    # Second, we create a model that maps the activations of the last conv
    # layer to the final class predictions
    regression_input = keras.Input(shape=last_conv_layer.output.shape[1:])
    x = regression_input
    for layer_name in classifier_layer_names:
        try:
            x = model.get_layer(layer_name)(x)
```

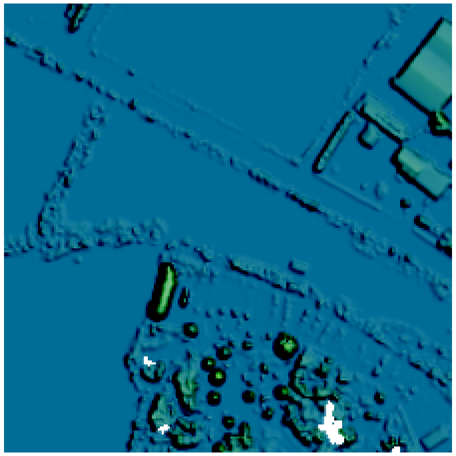
```
        except:
            x = model.layers[1].get_layer(layer_name)(x)
    regression_model = keras.Model(regression_input, x)



    # Then, we compute the gradient of the top predicted class for our input image
    # with respect to the activations of the last conv layer

    with tf.GradientTape() as tape:
        # Compute activations of the last conv layer and make the tape watch it
        last_conv_layer_output = last_conv_layer_model(img_array)
        tape.watch(last_conv_layer_output)
        # Compute predictions
        top_class_channel = regression_model(last_conv_layer_output)



    # This is the gradient of the top predicted class with regard to
    # the output feature map of the last conv layer
    grads = tape.gradient(top_class_channel, last_conv_layer_output)

    # This is a vector where each entry is the mean intensity of the gradient
    # over a specific feature map channel
    pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))

    # We multiply each channel in the feature map array
    # by "how important this channel is" with regard to the regression
    last_conv_layer_output = last_conv_layer_output.numpy()[0]
    pooled_grads = pooled_grads.numpy()
    for i in range(pooled_grads.shape[-1]):
        last_conv_layer_output[:, :, i] *= pooled_grads[i]

    # The channel-wise mean of the resulting feature map
    # is our heatmap of activation
    heatmap = np.mean(last_conv_layer_output, axis=-1)
    # print(last_conv_layer_output)
    # print(heatmap)
```

```
    # print(np.max(heatmap))
    # For visualization purpose, we will also normalize the heatmap between 0 & 1
    heatmap = np.maximum(heatmap, 0) / np.max(heatmap)

    return heatmap

# # Set the layers. # trying with vgg
# last_conv_layer_name = "block5_conv3"
# classifier_layer_names =  ["block5_pool",
#                             "dense",
#                             "dense_1",
#                             "dense_2",]


last_conv_layer_name = "post_relu"
#last_conv_layer_name = "conv5_block3_out"

classifier_layer_names =  ["flatten",
                           "dense",
                           "dropout",
                           "dense_1",
                           "dropout_1",
                           "dense_2",]
```

Loading random images and plotting heatmaps

```
# Display
from IPython.display import Image
import matplotlib.pyplot as plt
import matplotlib.cm as cm
%matplotlib inline

# Get the image in the right size
def get_img_array(img_path, size = (224, 224)):
    import tensorflow as tf
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=size)
    # `array` is a float32 Numpy array of shape (299, 299, 3)
```

```
    array = tf.keras.preprocessing.image.img_to_array(img)
    # We add a dimension to transform our array into a "batch"
    # of size (1, 224, 224, 3)
    array = np.expand_dims(array, axis=0)
    array = preprocess_input(array)
    return array

# Get an image
img_path = '/content/LIDAR/LIDAR_64774.png' # pick an image
data = get_img_array(img_path)

# Plot it
display(Image(img_path))
```

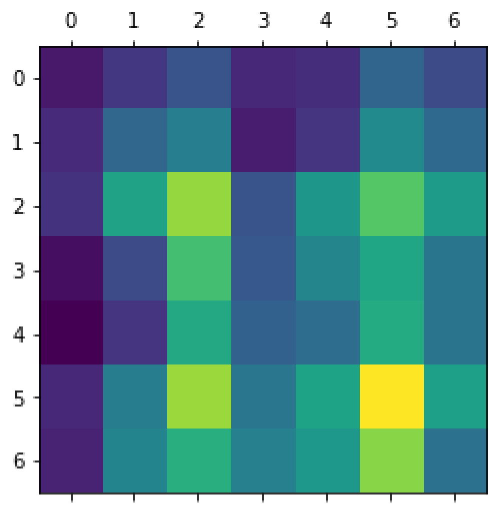


GradRam estimate

```
#Plot the heatmap! trying with vgg
heatmap = make_gradcam_heatmap(
    preprocess_input(data), ImageOnlyModel, last_conv_layer_name, classifier_layer_names
)
# heatmap = make_gradcam_heatmap(
#     preprocess_input(data), ImageOnlyModel, last_conv_layer_name, classifier_layer_names
# )
```

```
#Display heatmap

print(heatmap)

plt.matshow(heatmap)
plt.show()
```

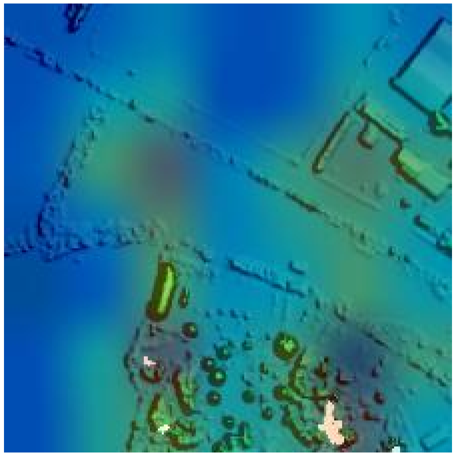
```
[[0.09705488 0.188017   0.2843762  0.14159058 0.15692163 0.34596252
  0.25399876]
 [0.14867824 0.3525952  0.44636166 0.10878056 0.18164127 0.49411422
  0.36233315]
 [0.17501602 0.58807176 0.8452461  0.2849584  0.54350185 0.7443358
  0.5656265 ]
 [0.06603655 0.2550343  0.71168214 0.29863393 0.47405562 0.6005398
  0.40790316]
 [0.03181586 0.18213327 0.616528   0.33383268 0.3792645  0.62592554
  0.40378892]
 [0.14077865 0.44248793 0.8551589  0.4104448  0.59493035 1.
  0.579672  ]
 [0.12425995 0.4670375  0.6403909  0.45414543 0.54945207 0.82881
  0.39616168]]
```



Superimposing heatmap on image

```
# We load the original image
```

```
    # We load the original image
    img = keras.preprocessing.image.load_img(img_path)
    img = keras.preprocessing.image.img_to_array(img)

    # We rescale heatmap to a range 0-255
    heatmap = np.uint8(255 * heatmap)

    # We use jet colormap to colorize heatmap
    jet = cm.get_cmap("jet")

    # We use RGB values of the colormap
    jet_colors = jet(np.arange(256))[:, :3]
    jet_heatmap = jet_colors[heatmap]

    # We create an image with RGB colorized heatmap
    jet_heatmap = keras.preprocessing.image.array_to_img(jet_heatmap)
    jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
    jet_heatmap = keras.preprocessing.image.img_to_array(jet_heatmap)

    # Superimpose the heatmap on original image
    superimposed_img = jet_heatmap * 0.4 + img
    superimposed_img = keras.preprocessing.image.array_to_img(superimposed_img)

    # Save the superimposed image
    save_path = "Lidar49078.jpg"
    superimposed_img.save(save_path)

    # Display Grad RAM
    display(Image(save_path))
```



✓ 0s completed at 1:16 AM