University of Zurich[UZH]

[Scooby Data Doo]

Applied Business Modelling and Analytics

Neural Networks and Deep Learning

Group Project

Authors

Mesut Ceylan, Tran Phan, Xiaoao Song

Supervisor

Robert Earle

Date of Submission: 28th June 2019

Final Project Report:
Applied Business Modelling and Analytics
Neural Networks and Deep Learning

Team: Scooby Data Doo
Members:
1. Tran Phan
2. Mesut Ceylan
3. Xiao'ao Song

1. Running Enviroment and supportted package

Python: 3.6

Python libraries:

- Numpy
- Matplotlib
- Pandas
- sklearn

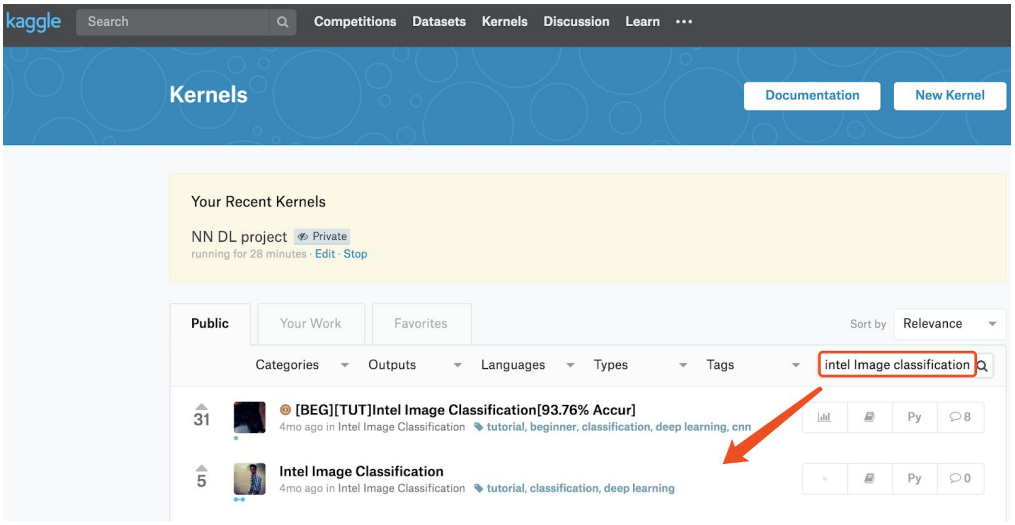
Other dependencies:

- tensorflow (1.12.0)
- Keras 2.2
- cv2 (openCV)

2. How to access data set and run the script

How to access data:

Go to kaggle with your account, type *intel Image classification* in the search box and you will be able to see the data set.



How to import the data set:

Type the below code in your project to use the data set we showed above.

```
import os
print(os.listdir("../input/seg_train/seg_train/"))
```

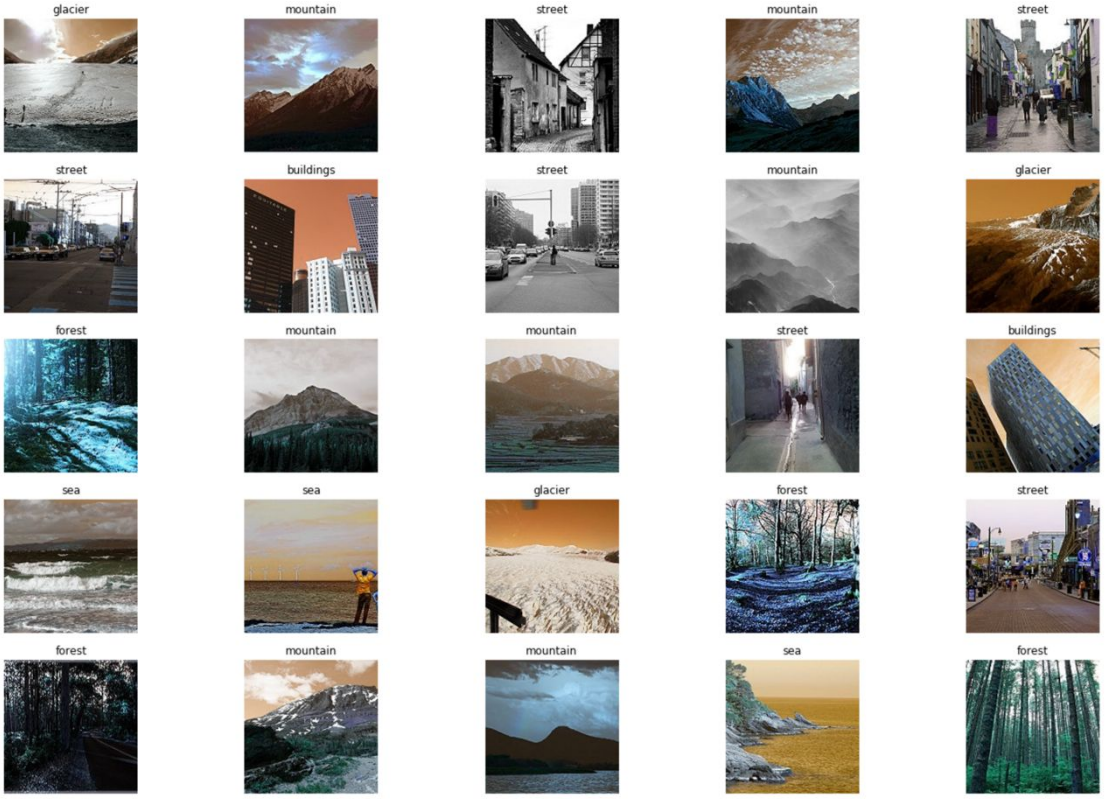
How to Run our project:

It is very recommended to run our notebook script on Kaggle unless you want to obtain the data set offline. To run the script on Kaggle, just upload our script onto Kaggle and run it cell by cell as you normally did in your Jupyter notebook.

3. Data Description

The dataset that we chose contains images of natural scenes around the world. It has around 25000 images of size 150 x 150 and the data is distributed under 6 categories which are buildings, forests, glaciers, mountain, sea, and street. The training, testing and prediction data are already separated into zip files. There are around 14000 images in train data folder, 3000 in test data folder, and 7000 in prediction folder.

Below are some images from the dataset:



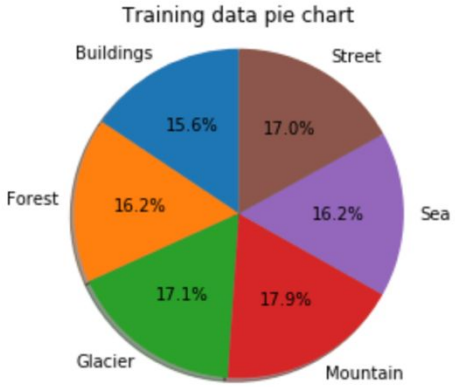
With this dataset, we label the images corresponding to their content and use the train dataset to train a convolutional neural network by using Keras. The trained neural network can then classify an input image as building, glacier, street, sea, or forest with high precision. The test set and prediction set are input to the trained neural network and some performance metrics of the model are recorded.

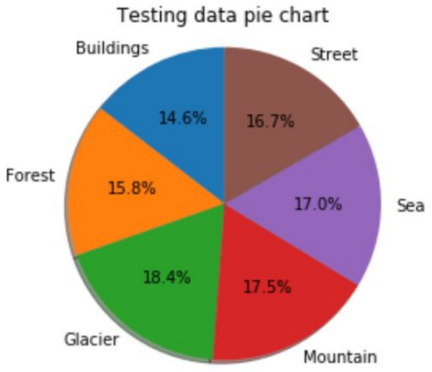
4. Exploratory Data Analysis

The features are images of scenes around the world with the resolution of 150x150 each image.
The labels are buildings - 0, forest - 1, glacier - 2, mountain - 3, sea - 4, and street – 5.

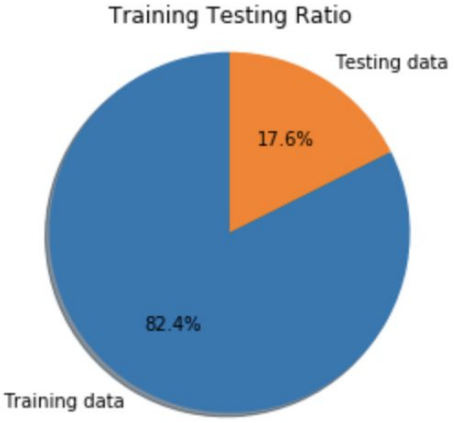
Using matplotlib, we have analysed and present the ratio between the categories in the data.
We can observe in the pie charts below that the amount of data of each category is almost

equal to each other. Therefore, we decide there is no need to balance the amount of data between categories since all categories are balanced very well.





As the data is already separated into train set, test set. We calculate the ratio between these two sets. Besides, there are also 7301 images which are used for prediction of the final model.



5. Data Preprocessing

The first step in the data pre-processing is to create a data frame as the input of the convolutional neural network. Each image in the corresponding categorized folder is assigned the corresponding label. With 0 is assigned to images in the buildings folder, 1 for images in the forest folder, 2 for images in the glacier folder, 3 for images in the mountain folder, 4 for images in the sea folder, and 5 for images in the street folder.

To ensure that all images in the dataset have the same size and to speed up the training process, we decided to resize all images to the size of 100x100. We chose this resolution for all images because with this resolution, the natural scenes in the images are still recognizable and smaller size of images could have us reduce the training time significantly and we could utilize more time for hyper-parameters tuning and metrics generating. We did not choose to resize images to 50x50 because we tried to do so and it decreases the accuracy of our model. The reason could be that some patterns in images become undetectable at that resolution.

Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the learning process. For that reason, we also normalize the pixel values so that each pixel value has a value between 0 and 1. This can be achieved by dividing all pixel's values

by the largest pixel value which is 255. This is performed across all channels, regardless of the actual range of pixel values that are present in the image. With the pixel values in range 0-1, images can still be viewed normally.

Data shuffling is the last step that we do in data pre-processing. This step is done as networks learn the fastest from the most unexpected sample. It is a good practice to do data shuffling since it reduces variance and make sure that models remain general and overfit less.

Besides all of these pre-processing steps, we also thought about data augmentation. We realised that natural image data can exist in many conditions such as sunny, snow, damp, etc. Thus, if our convolutional neural network does not understand the fact that certain landscapes can exist in a variety of conditions, it might mistakenly classify frozen lakeshores as glaciers or wet fields as swamps. This problem can be mitigated by creating new images with different season effects on existing images. There are some tools available for that purpose such that:

- Changing seasons using a CycleGAN (Source: https://junyanz.github.io/CycleGAN/),
- Deep Photo Style Transfer. Notice how we could generate the effect we desire on our dataset. (Source: https://arxiv.org/abs/1703.07511)

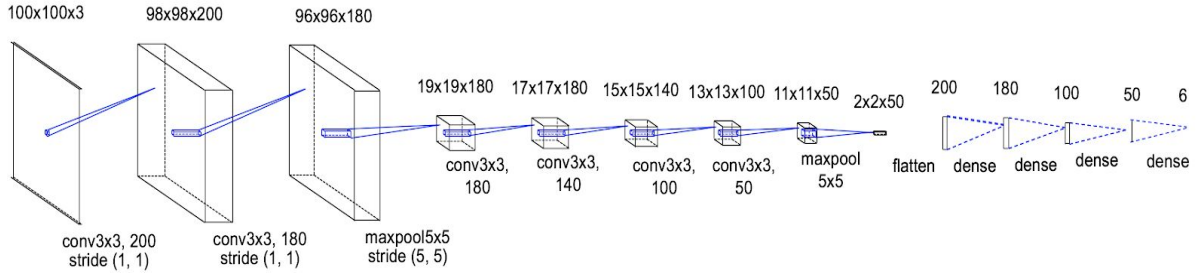
We did not dive deep into that because in our project, the result is satisfiable. Admittedly, applying these tools could make our neural network more intelligent.

6. Data Splitting

Since the data is split into train set and test set by default, we take 30% of the train set to use as validation set.

```
Number of training images: 14034
Number of testing images: 3000
Number of images for prediction: 7301
```

7. Network Architecture



Referring from other famous CNN architectures such as LeNet, AlexNet, we build our network architecture with group of convolutional and pooling layers into blocks. In our model, we have two groups of convolutional and pooling layers. The convolutional layers have decreasing number of filters. We separate convolutional and pooling layers group with the classifier part of the network by the flatten layer which transform the features into an array of features before it enters into the dense layers.

As in AlexNet, we use of the ReLU activation function after convolutional layers and softmax for the output layer, we use of Max Pooling instead of Average Pooling as it has

better performance in classification task. We also use dropout regularization after the fully connected layers.

8. Hyper-parameters Tuning

a. Optimizer Selection

Optimizer selection on neural network structure yet is a fundamental step in achieving accurate results. Optimizers are responsible to optimize the objective function of the model by minimizing or maximizing the outcome. As our main goal is to decrease the difference between actual results and predicted results in our neural network model, we need to minimize the cost function with optimal weight values. To be able to choose the best weight values via selecting the best optimizer, we tested four different optimizer to check their suitability on our neural network structure. Optimizers that we tested are Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam), RMSProp. Again, we fixed the other hyper-parameter at the beginning of the testing. Results regarding optimizer selection part and model outcomes can be found in Table 1 below.

Optimizer	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
SGD	0,70	0,83	0,68	0,83
Adam	0,89	0,34	0,80	0,63
RMSprop	0,72	0,90	0,76	0,80

Table 1*. Training and Validation data results with respect to different optimizers.*

We run the model for each optimizer by using their default parameters. We observed Adaptive Moment Estimation (Adam) outperformed the other optimizers both on training and validation scores. Therefore, we decided to continue our analysis by selecting Adam as optimizer for our model.

b. Activation Function Selection

Activation function is a crucial part of the neural network structure. Because activation function is responsible for transforming input data into output data. The function decides particular neuron will be fired or not by calculating weight between them. In this way, function acts as bridge between next neuron and the neuron that connected with it, then function decides whether next neuron should consider output of the particular neuron or not. Although there are many different functions available for use in neural network, we analyzed the most commonly used ones, particularly Rectified Linear Units (ReLU), Sigmoid and TanH and Exponential functions. Since each activation function is unique, we also included activation function selection part in the hyper-parameter tuning section. Our approach is maintaining same activation function for all network structure. On our analysis, we modified our code with related activation function and test each activation function one by one.

```
model.add(Layers.Conv2D(200,kernel_size=(3,3),activation='relu',input_shape=(IMAGE_SIZE,IMAGE_SIZE,3)))
model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
model.add(Layers.Conv2D(140,kernel_size=(3,3),activation='relu'))
model.add(Layers.Conv2D(100,kernel_size=(3,3),activation='relu'))
model.add(Layers.Conv2D(50,kernel_size=(3,3),activation='relu'))
model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
model.add(Layers.Flatten())
model.add(Layers.Dense(180,activation='relu', kernel_initializer='normal'))
model.add(Layers.Dense(100,activation='relu', kernel_initializer='normal'))
model.add(Layers.Dense(50,activation='relu', kernel_initializer='normal'))
model.add(Layers.Dropout(dropout_rate))
model.add(Layers.Dense(6,activation='softmax'))
```

Figure. Screenshot of part of the convolutional neural network structure.

function	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
relu	0,89	0,34	0,80	0,63
sigmoid	0,18	1,79	0,18	1,79
tanh	0,77	0,65	0,75	0,70
exponential	0,16	1,79	0,15	1,79

***Table 2**. Training and Validation data results with respect to different activation functions.*

We conduct our analysis on activation function by fixing other hyper-parameters such as number of epochs, batch size, dropout rate, pooling size, optimizer. Not surprisingly, we observed ReLU activation outperformed the other activation functions. Since our task is image classification, we were expecting that ReLU function will perform way better than other activation function. Therefore, we continued our project with selecting ReLU function as activation function. One can see performance of other functions on training and validation on Table 2.

c. Learning Rate Decay

In addition to activation function and optimizer selection, we also examined our model with different value learning rate decay parameter. As it is mentioned above, we selected to use Adam optimizer in our model. To analyze how the model performs under different optimizer learning rate parameter, we tested different values of decay and observe its effect on model accuracy and loss values. We started our analysis with relatively bigger values and slowly decrease the decay value. One can observe model outcome differences at Table 3 below.

```
def my_model(dropout_rate=0.3, decay=0.003):

    model = Models.Sequential()

    model.add(Layers.Conv2D(200,kernel_size=(3,3),activation='relu',input_shape=(IMAGE_SIZE,IMAGE_SIZE,3)))
    model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
    model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
    model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(140,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(100,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(50,kernel_size=(3,3),activation='relu'))
    model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
    model.add(Layers.Flatten())
    model.add(Layers.Dense(180,activation='relu', kernel_initializer='normal'))
    model.add(Layers.Dense(100,activation='relu', kernel_initializer='normal'))
    model.add(Layers.Dense(50,activation='relu', kernel_initializer='normal',activity_regularizer=l1(0.01)))
    model.add(Layers.Dropout(dropout_rate))
    model.add(Layers.Dense(6,activation='softmax'))
```

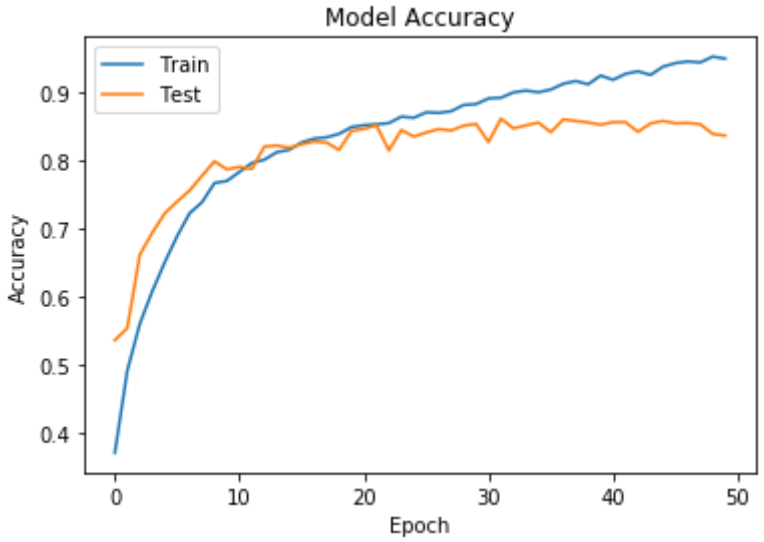
Decay	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
0,2	0,49	1,18	0,54	1,12
0,1	0,50	1,19	0,55	1,12
0,01	0,79	0,62	0,79	0,59
0,005	0,82	0,53	0,82	0,53
0,003	**0,86**	**0,41**	**0,84**	**0,47**
0,001	0,88	0,36	0,84	0,50
0,0005	0,90	0,28	0,82	0,59

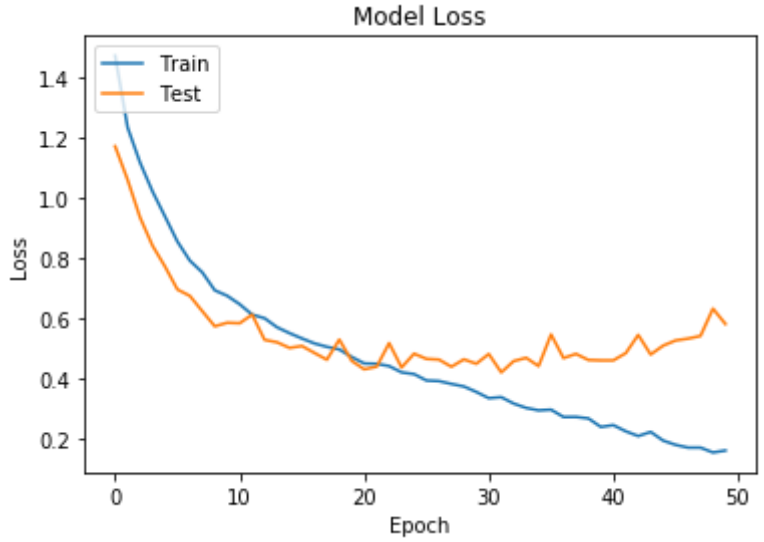
***Table 3**. Training and Validation data results with respect to different learning rate decay values.*

End of our analysis, we observed that the best learning rate decay value for our model is 0.003, generating 0.86 and 0.41 accuracy and loss values on training set, 0.84 and 0.47 on validation set. We observed that when we lower the decay value until some point, we were able to reduce the overfitting of the model by lowering validation loss value down to 0.47. As we decrease the value from 0.003 to 0.001 and even to 0.0005, model again started to overfit more. Therefore, we stopped testing more values since we obtained the optimal level of decay value.

d. Epoch Tuning

First we do hyper-parameters tuning for the epoch. We set the epoch to 50 and documented the accuracy and loss on both train set and test set.





We could observe that the accuracy on the **training** set still increases after epoch 50th, while the accuracy on **validation** set fluctuates around 80% from epoch 20th. As for the loss, the loss on **training** set keep decreasing and it reaches below 0.2 at epoch 50th, but the loss on **validation** set fluctuates about 0.5 and have increasing trend after epoch 35th.
With this observation, we decided to choose the configuration for epoch as 35 in the final train.

e. Network Weight Initialization Tuning

The next parameter that we choose to tune is the initialization of network weights. For the reason that different distribution of value of weights will affect the training time as well the optimization process. We tried training the model with different initialization schemes for the network weights in keras such as “uniform”, “lecun_uniform”, and “normal”.

```
model.add(Layers.Dense(180,kernel_initializer='lecun_uniform', activation='relu'))
```

With different settings, we will have different initial network weights at the beginning of the training and the result we got is as below:

Initial Weight	Training Accuracy	Training Loss	Validation Accuracy	Validation loss
uniform	0,786	0,617	0,8069	0,5616
normal	0,814	0,5571	0,8236	0,5107
lecun_uniform	0,8324	0,498	0,7953	0,6279

Table 4. Results with respect to different network weight initialization

With this result, “normal” setting has the best result on the validation set. So we choose “normal” as our initial weights setting.

f. Dropout Rate Tuning

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. We tried dropout rate from 0 to 0.5 and the result was shown below(noted that in Keras, dropout number is the probability of dropping):

drop rate	accuracy training	loss of training	test accuracy	loss of test set	delta
0,0	0,872	0,356	0,809	0,554	0,063
0,1	0,876	0,357	0,843	0,452	0,033
0,2	0,860	0,395	0,833	0,485	0,027
0,3	0,864	0,404	0,857	0,451	0,006
0,4	0,853	0,438	0,841	0,473	0,013
0,5	0,847	0,470	0,839	0,458	0,008

***Table 4**. Training and Testing data results with respect to different dropout rate*

With this result shown above, the dropout rate at 0.3, 0.4, 0.5 are both acceptable because their accuracy difference between **training** and **testing** set is rather small.

g. Stride Tuning

Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. We also tried stride tuning for 2,3,4 and 5. The result was shown below:

Stride	accuracy training	loss of training	test accuracy	loss of test set	delta
5	**0,85**	**0,47**	**0,84**	**0,46**	**0,008**

4	**0,84**	**0,46**	**0,85**	**0,45**	**-0,001**
3	0,87	0,36	0,84	0,47	0,028
2	0,93	0,22	0,84	0,57	0,094

***Table 5**. Training and Testing data results with respect to stride*

With this result shown above, the stride at 5 or 4 are both acceptable because their accuracy difference between **training** and **testing** set is rather small. We will prefer stride at 5 because under the same accuracy level, stride 5 is more computation efficient.

9. Overfitting Reduction

To reduce overfitting, we reduce the images' size from 150x150 to 100x100. This step would reduce the features to be learnt and detected by the CNN significantly.

Methods that we used for overfitting reduction as follows:

- Dropout
- Activity Regularization with L1 Penalty
- Obtaining More Data

Besides, we also apply dropout at the last dense layer and data augmentation. Dropout is one of the methods that we used in our model in order to avoid or at least reduce overfitting within neural network. Dropout is a technique that removes neurons on particular layer according to probability that can be set. In our model, we examined different dropout rates with respect to accuracy and loss values both on training and validation sets. What we observed is that Dropout Rate with 0.3 generates the best results among 0.0, 0.1, 0.2, 0.3, 0.4 and 0.5.

In addition to Dropout as one of the regularization methods, we also decided to introduce Activity Regularization method within our network to tackle overfitting. Activity Regularization is approach that can be used for reducing overfitting, thus providing better model with respect to generalization. It works as function that regularize output of the certain layer of the network. We introduced L1 penalty to penalize sum of weights.

```
def my_model(dropout_rate=0.3, decay=0.003):

    model = Models.Sequential()

    model.add(Layers.Conv2D(200,kernel_size=(3,3),activation='relu',input_shape=(IMAGE_SIZE,IMAGE_SIZE,3)))
    model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
    model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
    model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(140,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(100,kernel_size=(3,3),activation='relu'))
    model.add(Layers.Conv2D(50,kernel_size=(3,3),activation='relu'))
    model.add(Layers.MaxPool2D(pool_size=(5,5), strides=(5,5)))
    model.add(Layers.Flatten())
    model.add(Layers.Dense(180,activation='relu', kernel_initializer='normal'))
    model.add(Layers.Dense(100,activation='relu', kernel_initializer='normal'))
    model.add(Layers.Dense(50,activation='relu', kernel_initializer='normal',activity_regularizer=l1(0.01)))
    model.add(Layers.Dropout(dropout_rate))
    model.add(Layers.Dense(6,activation='softmax'))
```

We obtained the following results after introducing Activity Regularization method with differing lambda values: 0.01, 0.1 and 1. Outcomes can be found on Table 6.

Lambda Value	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
0,01	0,86	0,59	0,84	0,58
0,1	0,18	1,79	1,79	0,17
1	0,18	1,79	1,79	0,17

***Table 6**. Training and Validation data results with respect to different activation regularization lambda values.*

In addition to dropout and penalizing activities of neurons, we decided to apply data augmentation in order to obtain more data and reduce overfitting. We used ImageDataGenerator to generate more images and tried using two parameters
1. featurewise_center: set input mean to 0 over the dataset.
2. featurewise_std_normalization: divide inputs by std of the dataset, feature-wise.

```
datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    )
datagen.fit(train_data)
# Concatenating the old data with the augmented data
result_x  = np.concatenate((train_data, train_data1), axis=0)
result_y  = np.concatenate((labels, labels1), axis=0)
# # fits the model on batches with real-time data augmentation:
trained = model.fit_generator(datagen.flow(result_x, result_y, batch_size=35),
                    steps_per_epoch=len(result_x) / 32, epochs = 20)
```

Using fit_generator function to train the model, after 20 epochs, we got 0.89 accuracy, 0.325 loss on training set and 8.88 accuracy, 0.336 loss on validation set. This shows that by applying data augmentation, we reduce loss and thus, reduce overfitting.

10. Model Outcomes Before Hyperparameter Tuning and After It

At this section of our report, we wanted to highlight the changes on model outcomes before conducting hyperparameter tuning and after it.

Before tuning hyper parameters, our neural network has outcome as following: 0.97 and 0.79 accuracy on **training** and **testing** sets respectively. However, there is a huge difference on loss values. We obtained 0.08 loss on **training** set and 1.17 on **testing** set. This difference is obviously result of overfitting of the model.

However, when we run our model after hyperparameter tuning that explained above, we obtained 0.91 and 0.85 accuracy on **training** and **testing** sets respectively. In addition, we obtained 0.26 and 0.48 loss values on **training** and **testing** sets. Introduced outcomes of the two models can be found in Table 7 below.

	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
Before Tuning	0,97	0,08	0,79	1,17
After Tuning	0,92	0,26	0,85	0,48

Table 7*. Training and Validation data results with respect to tuned and not tuned neural networks.*

Finally, we observed the model differences on confusion matrix as well. At left matrix that belongs to neural network without hyperparameter tuning, we observed that the model is not able to predict glaciers and mountains very well. If we compare this outcome with the right matrix that belongs to neural network with hyperparameter tuning, we can see that model with hyperparameter tuning performed better.

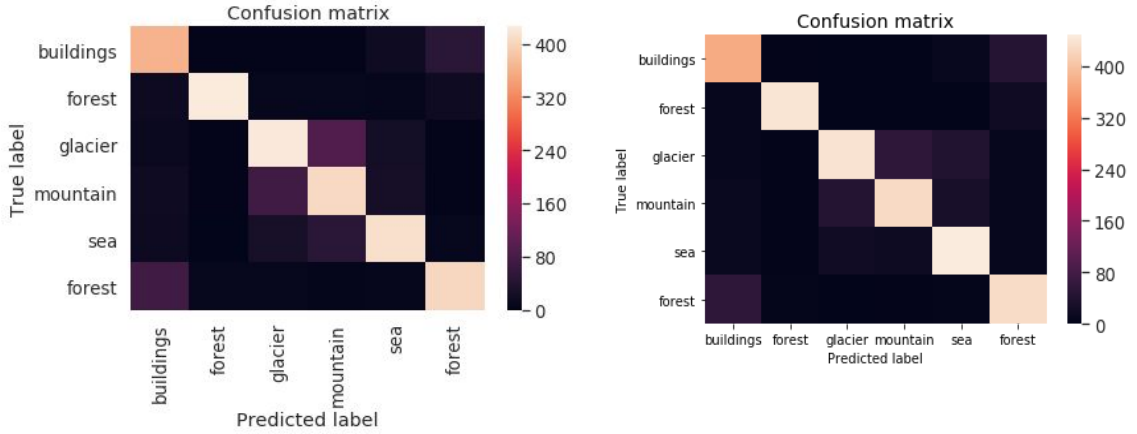
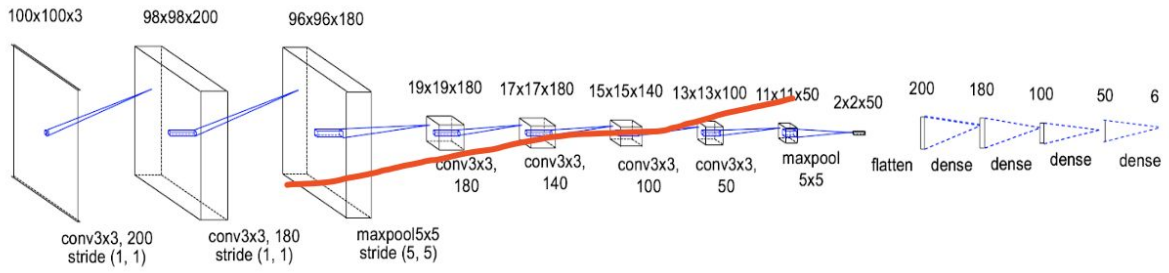


Figure C. *Confusion matrix of two models. Left: Model without hyperparameter tuning. Right: Model with hyperparameter tuning.*

11. Different Network Comparison

We now compare our model with another network with different structure, the code and figure are shown below:



```
model = Models.Sequential()
model.add(Layers.Conv2D(200,kernel_size=(3,3),activation='relu',input_shape=(IMAGE_SIZE,IMAGE_SIZE,3)))
model.add(Layers.Conv2D(180,kernel_size=(3,3),activation='relu'))
model.add(Layers.MaxPool2D(5,5))
# model.add(Layers.MaxPool2D(5,2))

# model.add(Layers.MaxPool2D(5,2))
model.add(Layers.Flatten())
model.add(Layers.Dense(180,activation='relu'))
model.add(Layers.Dense(100,activation='relu'))
model.add(Layers.Dense(50,activation='relu'))
model.add(Layers.Dropout(dropout_rate))
model.add(Layers.Dense(6,activation='softmax'))
model.compile(optimizer=Optimizer.Adam(lr=0.0001),loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.summary()
```

As we can see, under the same setting, the only difference between these two network is that the later network is simplified a lot by removing out several layers.

	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
Full Network	0,92	0,26	0,85	0,48
Simple Network	0,98	0.07	0.78	1.03

The result showed that our full network is better than simple network. The simple network have a serious problem with overfitting.

Effort Report:
We distributed the tasks equally in both exam questions and the final project.

Team: Scooby Data Doo

Reference

https://www.kaggle.com/rdmisal/intel-image-classification

https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/