The Lightweight IBM Cloud Garage Method for Data Science

Architectural Decisions Document Template

# Architectural Components Overview



IBM Data and Analytics Reference Architecture. Source: IBM Corporation

## Data Source

### Technology Choice

Intel Image Classification data set from Kaggle: <https://www.kaggle.com/puneet6060/intel-image-classification>

This is image data of Natural Scenes around the world.

This Data contains around 25k images of size 150x150 distributed under 6 categories: buildings, forest, glacier, mountain, sea and street.

The Train, Test and Prediction data is separated in each zip files. There are around 14k images in Train, 3k in Test and 7k in Prediction. This data was initially published on https://datahack.analyticsvidhya.com by Intel to host a Image classification Challenge.

### Justification

Not applicable as there is no choice in the data to be used in this project.

## Enterprise Data

### Technology Choice

Not required.

### Justification

No further data required other than data set from Kaggle.

## Streaming analytics

### Technology Choice

Not required.

### Justification

Data is one-off load. Data will be downloaded from Kaggle to local computer where a git repository will be created to contain it and transfer to Github. This repository will then be cloned on IBM Watson Studio cloud platform so that data integration can take place there.

## Data Integration

### Technology Choice

IBM Watson Studio

### Justification

Data integration will take place using Jupyter Notebooks on the IBM Watson Studio platform.

## Data Repository

### Technology Choice

IBM Watson Studio

### Justification

Data will be processed using Jupyter Notebooks on the IBM Watson Studio platform.

## Discovery and Exploration

### Technology Choice

IBM Watson Studio, Jupyter Notebooks using numpy, Python Image Library and Matplotlib.

### Justification

Images will be processed using the Python Image Library and converted to numpy arrays. Matplotlib will be used to visualize the data, primarily displaying images.

## Actionable Insights

### Technology Choice

IBM Watson Studio, Jupyter Notebooks, numpy, Keras/TensorFlow, pickle and Matplotlib.

### Justification

Numpy will be used to store image and label data as arrays, with Convolutional Neural Network models being built using Keras/TensorFlow. Pickle will be used to store objects so that they can be transferred between notebooks and sessions, and Matplotlib to visualize model metrics while developing models.

## Applications / Data Products

### Technology Choice

Model saved in h5 format.

### Justification

This format saves both model architecture and weights in a portable file format.

## Security, Information Governance and Systems Management

### Technology Choice

Not required

### Justification

Data used in the model is open source, with no privacy or security implications.

# Process Model

## Initial Data Exploration

### Task summary

Image and label data from Kaggle was downloaded to local machine where it was saved in a git repository. This repository was uploaded to Github, then cloned from there into IBM Watson Studio.

Training image data was converted into numpy arrays and stored in a list. A list of labels was created from the folder names in which the data was stored.

Frequency of images by label was checked to ensure the dataset was fairly evenly balanced between the 6 classes. Results: 'mountain': 2512, 'glacier': 2404, 'street': 2382, 'sea': 2274, 'forest': 2271, 'buildings': 2191.

Image shape was also checked as this needs to be the same for each image. Majority were 150 x 150, but 48 images in the training dataset were found to have slightly different sizes.

A function was developed to resize these images and the resized images visually inspected to ensure they were not too distorted.

### Assets created

Jupyter notebook: intelImageClassification.data\_exp

## Extract, Transform, Load (ETL)

### Task summary

An ETL pipeline was developed to accomplish the following tasks:

* Read in images from training and test folders and output as a list of images
* Create labels list from the folder names
* Get frequency counts by label to ensure that both training and test data were reasonably well balanced
* Take 3000 images and labels from the training dataset to use as a validation dataset when developing the model, leaving 11034 for training
* Convert images to a list of arrays, resizing any that were not 150 x 150 pixels
* Create a one-hot encoded list of arrays from labels to be used as the target in model training
* Save all data as pickle files to allow it to be transported between notebooks. This required splitting the training arrays into 2 files as the file size was too large to store in IBM Watson Studio.

### Assets created

Jupyter notebook: intelImageClassification.etl

## Feature Creation

### Task summary

There were 2 main transformations were applied at this stage as one-hot encoding of the target classes had already been completed in the ETL stage:

* Array data was normalized by dividing by converting to float32 and dividing by 255
* During the model definition stage it was identified that training models on arrays of size 150x150x3 would take an extremely long time. Therefore additional lists of training and validation arrays were created with sizes 30x30x3, 50x50x3 and 75x75x3.

### Assets created

Jupyter notebook: intelImageClassification.feature\_eng

## Model Definition

### Task summary

This task was very iterative. Initially I tried building CNN models based on architectures from various articles on the internet. These proved successful in accurately classifying images, but slow to train on 150x150x3 arrays. I also felt that there was little science behind any tweaks that I was making to the architecture.

I then decided to try to build a convolutional network from scratch, starting as simply and small as possible. I would then try to increase the complexity a step at a time, testing each main parameter separately. This assumes they work independently when in reality they may interact. However grid-searching all the possible combinations of parameters would take too long.

To speed the process up I went back the Feature Engineering task and built training and validation sets with smaller array sizes (see documentation for that stage). After some testing I settled on developing the model architecture with training and validation arrays of size 50x50x3.

The initial model built had 5 layers:

* A 16 neuron convolutional layer with kernel size 3
* A 10 window max-pooling layer which reduced the size of the 50x50 image data to a 5x5 matrix
* A flattening layer
* A dense fully-connnected layer with 16 neurons
* A final dense layer of 6 neurons to classify the images into the 6 classes

Relu activation functions were used in all layers except the final one where softmax was used. Please note activation functions were the only parameter which I did not vary, primarily because the vast majority of examples of image classifying CNNs I found on the internet used these activation functions.

The loss function used was categorical cross entropy and the metric used was accuracy. Again these were not varied for the same reason as above.

The initial optimizer used was Adadelta with batchsize 128. After 20 epochs the model was 72% accurate on the validation set with loss 0.77.

Once this model was fitted the following parameters and architecture changes were tested:

* Adding additional convolutional and fully connected layers. Adding both seemed to provide the best results, but the gains were fairly marginal. I decided to continue with 2 of each layer because I expected that when running on the full size arrays there would be more information to pull from them for classification.
* Increasing kernel size in convolutional layers from 3 to 5, testing each-layer separately and together. Again gains were marginal, but I decided to stay with kernel size 3 due to the trajectory of accuracy and loss in the validation data improving most at epoch 20.
* Pooling window sizes 5, 7, 10, 16 and 25. These reduce the size of the output of the convolutional layer to 10x10, 7x7, 5x5, 3x3 and 2x2 respectively. On training the smaller the window the higher the accuracy, but on validation it was less clear cut with little difference between 5, 7 and 10. I settled on 5 due to its performance on the training dataset, though this might risk over-fitting.
* Doubling neuron size in each layer to 32 separately. This didn’t seem to make any difference to accuracy or loss on the validation data, so I stayed at 16 neurons in each layer.
* *Batch-sizes 8, 16, 32, 64, 128 and 256. I could not get larger batch-sizes to run. Most articles I read on the internet indicated that the larger batch-size the better, but in intial epochs the model seemed to improve accuracy faster the smaller the batch size.*
* *Optimizers 'SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax' and 'Nadam'.*
* *At this point I ran a longer 30 epoch training with the optimal parameters identified so far. After epoch 10 validation loss started to creep up indicating the model was over-fitting.*
* *Adding dropout layers after each convolutional or dense layer (except the final classifying layer) to assess impact on reducing over-fitting.*
* *Changing dropout rate.*

*Finally I tested building a model with the same architecture and parameters with the full 150x150 sized arrays for a small number of epochs to check if it would work before heading into the model training stage.*

### Assets created

Jupyter notebook: intelImageClassification.model\_def

## Model Training

### Task summary

### Assets created

Jupyter notebook: intelImageClassification.model\_train

## Model Evaluation

### Task summary

### Assets created

Jupyter notebook: intelImageClassification.model\_evaluate

## Model Deployment

### Task summary

Jupyter notebook based on ETL to read in prediction dataset

Load model and run it over dataset

Output sample of n records with predicted label.

### Assets created

Jupyter notebook: intelImageClassification.model\_deploy