Indrasena kallam

Capstone 2 Final Report
Springboard Data Science Bootcamp

Classifying Land Use in Satellite Imagery with Deep Learning

Overview

Satellite imagery is crucial for many applications, including agriculture, city planning, natural resource management, environmental monitoring, and disaster response. In recent years, there has been significant growth in the availability of high-resolution satellite imagery, however this immense quantity of data requires processing to make it useful for analysis.

Common processing tasks for satellite imagery include land use classification, object detection, semantic segmentation and object counting. Since satellite imagery can cover large areas, the main business cases for applying computer vision models to automate these tasks is that they can reduce overall costs and processing time, and potentially increase the accuracy of interpretations. Automated image processing enables mapping land uses across extensive areas that are impractical to delineate by hand. Access to digitally classified land use data facilitates applications such as tracking land use changes over time, and identifying regions of land use that are at risk.

For decades, the remote sensing community has applied algorithmic techniques and classic machine learning methods (random forests, SVM, etc) to automate common satellite image processing tasks, such as classifying land uses and detecting surface features. More recently, there has been increased adoption of deep learning techniques that apply neural networks to learn classification and detection tasks.

DeepSat-6 Dataset

For this project, I have applied deep learning to satellite image classification by developing convolutional neural networks (CNNs) for classification of satellite image tiles. Two approaches were implemented: building a custom CNN from scratch and using a pre-trained CNN. Both CNNs were trained on a benchmarking dataset (Deepsat-6) and achieved over 96% accuracy on held-out data.

The DeepSat-6 dataset contains pre-labeled image tiles extracted from the National Agriculture Imagery Program (NAIP) dataset. The full NAIP dataset covers the entire US at 1-meter resolution over 4 bands: red, green, blue and near-IR. The DeepSat-6 dataset consists of 28x28 pixel non-overlapping tiles extracted from NAIP imagery over different regions of California. The labels are one-hot encoded vectors for the following 6 land cover classes: barren land, trees, grassland, roads, water and buildings. Both training and test datasets are available, and the datasets have been randomized.

Data Acquisition

I decided to use the Google environment for this project. The first challenge was to bring the data from Kaggle into Colab using the Kaggle API. This process involved: obtaining an API key, using shell commands within Colab to move the key to the appropriate location in the Colab file system, installing the Kaggle public API python package, and downloading the DeepSat-6 dataset via the package’s command line tools.

Since the VMs in Colab file system don’t persist after each session, I need to copy the dataset to my personal Google drive to make it accessible for future work. The copy was done by mounting google drive to the VM and running shell commands from within Colab. The DeepSat-6 dataset contained the data in both csv and MATLAB (.mat) formats. The MATLAB format, since it is more compact, easy to load with SciPy, and in a format close to that expected by Keras.

Data Inspection and Reworking:

Method 1: Directly accessing from mat files:

After obtaining the data, I used scipy.io. readmat() to load the dataset into a dictionary. The dictionary contained key-value pairs for the test and training data, test and training labels, and a set of annotations for translating the labels into human-readable categories. The data values were all stored as arrays.

A series of tests were applied to verify the quality of the raw data. These included testing that each image was assigned to a single category, that there were no null values in the image arrays, and that all pixel values were between 0 and 255. No data quality issues were identified.

The imagery data required reworking to convert it to the format expected by TensorFlow. These steps included:

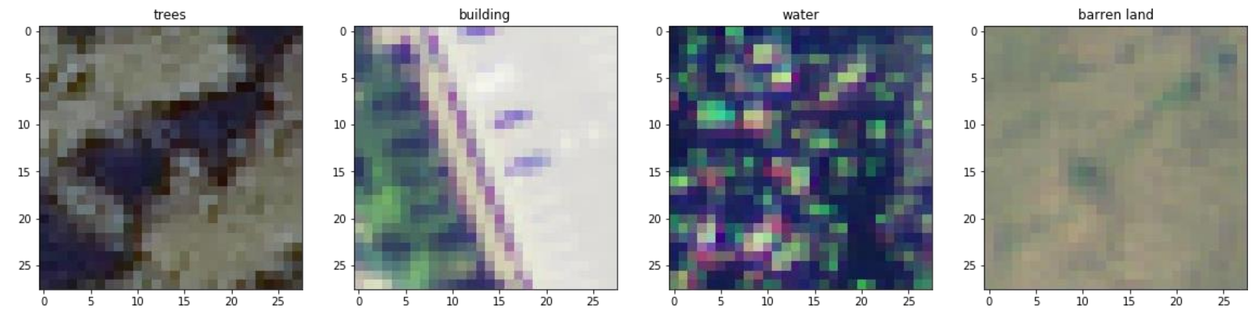
1.Re-dimensioning the multidimensional image arrays from the original shape (rows, columns, channels, samples) to the channels-last convention expected by TensorFlow (samples, rows, columns, channels).

2) Transposing the label arrays.

3) Creating a pandas Series of the training labels (as text) indexed by image number.

Method 2: Directly calling method

Directly calling the data from the csv files and which are stored in the names of train.csv and test.csv
After these manipulations, individual image tiles were easy to plot and identify by assigned class:



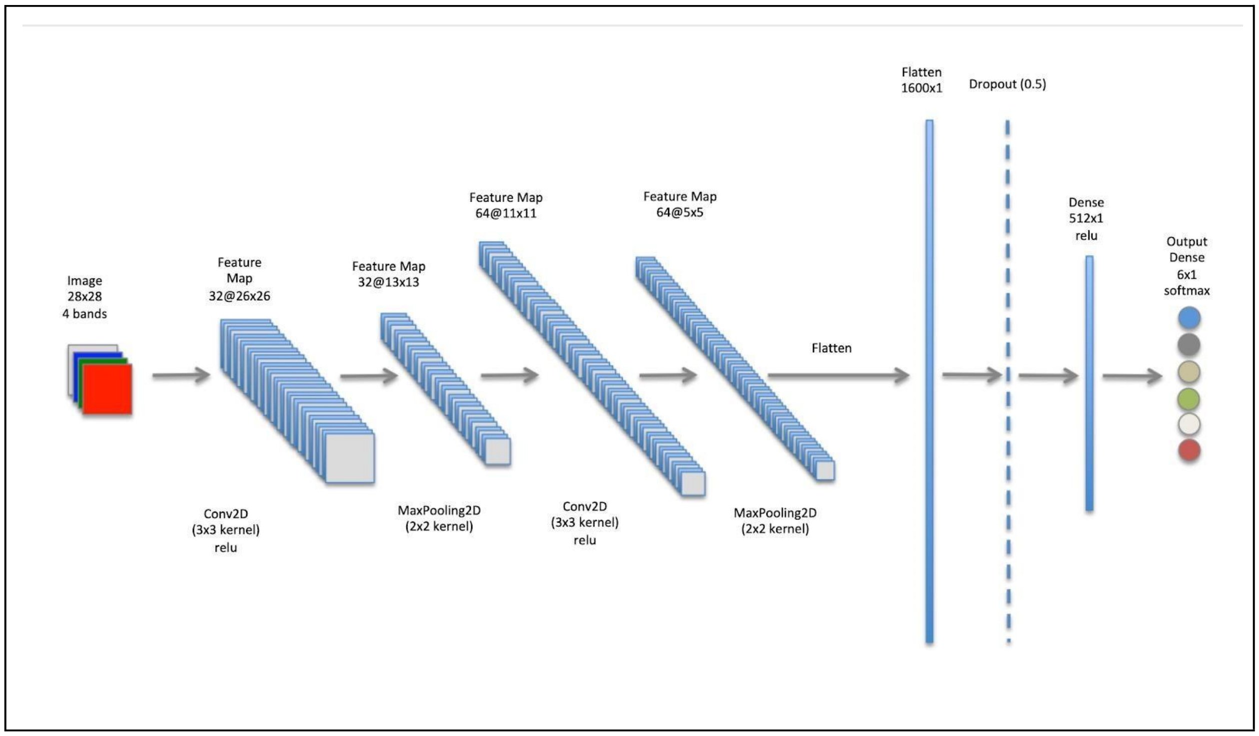
Application of Deep Learning

The convolutional neural network (CNN) is a deep learning algorithm commonly used in computer vision applications. Implementing CNNs were applied to develop classification models for the DeepSat-6 data. The “baseline” CNN uses a simple CNN architecture. Model were implemented in Python on Google Colab using the Keras interface to TensorFlow and the general methods described in Deep Learning with Python.

Classification models implement a series of convolution and pooling steps (the convolutional base) followed by a densely connected classifier to produce the final output. The convolutions learn a hierarchy of local patterns at different scales, while the dense layers learn global patterns.

Baseline Model

The baseline model was implemented using TensorFlow's Keras API with the following architecture:



Baseline CNN design

The input to the baseline CNN is tensor representing a 28x28 4-band image tile. The first convolutional layer applies 32 convolutions (with a 3x3 kernel) to the input image followed by the rectified linear unit (relu) activation function to introduce non-linearity into the model. The result is a 3D tensor (feature map) with dimensions 26x26X32. Each of the two max pooling operations down samples the feature maps by a factor of 2. This down sampling allows each convolution stage to learn patterns at a different scale. Due to the small extent of the input image tiles, only two stages of convolutions and max pooling were applied.

After the second max pooling operation, the input data has been transformed to a feature map with dimensions 5x5x64. This representation is flattened before being input a densely-connected classifier consisting of a fully-connected dense layer followed by a fully-connected dense output layer. The final dense layer estimates the class of the image, by applying the 'softmax' activation function to generate a probability of the input image being in each class. The class with the highest probability is assigned to the image.

As shown above, a dropout layer with a dropout rate of 0.5 was inserted after the flatten layer. The dropout layer randomly zeros out a portion of the output features of the prior layer during training and is used as a regularization tool to prevent overfitting.

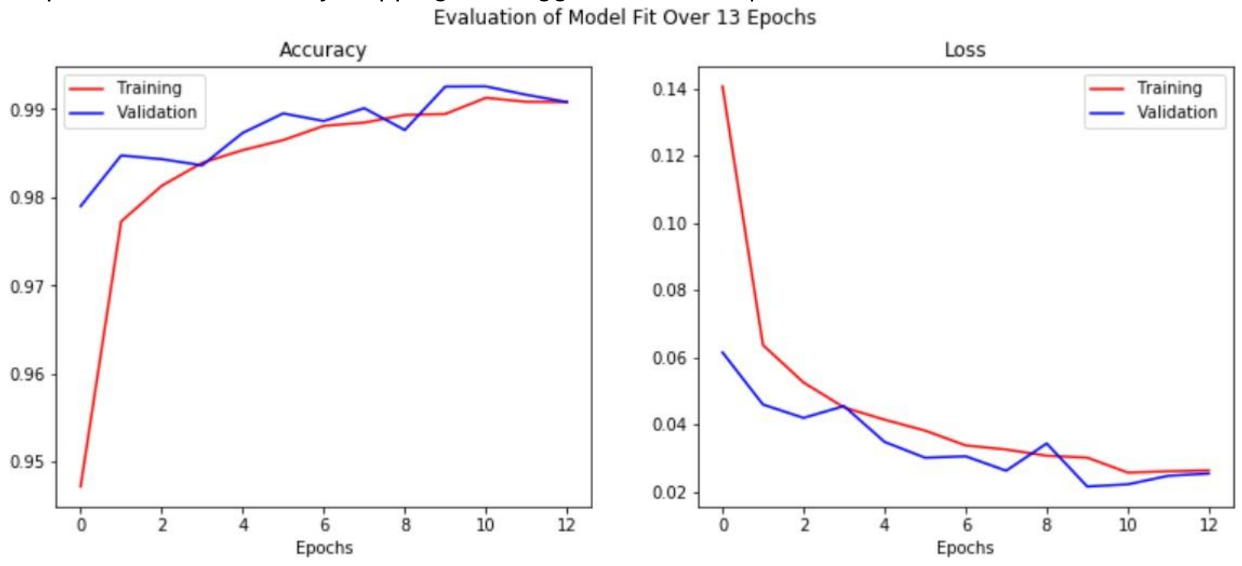
This model has a total of 842,470 parameters. Approximately 20,000 of these parameters are the weights to be trained in the convolution filters and the bias values to be applied after the convolutions. The majority of parameters are the weights of the first fully connected dense layer of the classifier.

Compiling a model in Kera’s involves selecting a loss function, optimizer function, and output metrics to be reported during training. Since this is a multi-class classification problem, the categorical cross-entropy loss function was selected, which penalizes misclassifications based on the differences in log-probabilities between predicted and actual outcomes.

The baseline model was fit by backpropagation was chosen by trial and error by testing different optimizers Adam and learning rates. The Keras ImageDataGenerator class was used to create python generators to supply batches of pre-processed images during the model fitting process. The pre-labeled training data was randomly split into training and validation datasets, comprised of 280,000 and 64,800 images, respectively. Each generator was given a batch size of 512 images. Given the large size of the dataset, data augmentation was not needed to prevent overfitting.

The baseline model was fit using the fit() method applied to the training and validation generators. The steps_per_epoch value was set to 545 and 126 for the training and validation sets to ensure the full set of image tiles would be used for training over each epoch (iteration over the full dataset). The number of epochs was set to 30 with early stopping enabled in the event the validation loss stabilized before reaching 30 epochs.

The accuracy and loss values for the training and validation datasets were recorded over the course of training. The training process achieved more than 99.6% accuracy on the validation data after only a few epochs. As shown below, additional epochs beyond 10 did not produce significant gains in performance, and early stopping was triggered after 13 epochs.

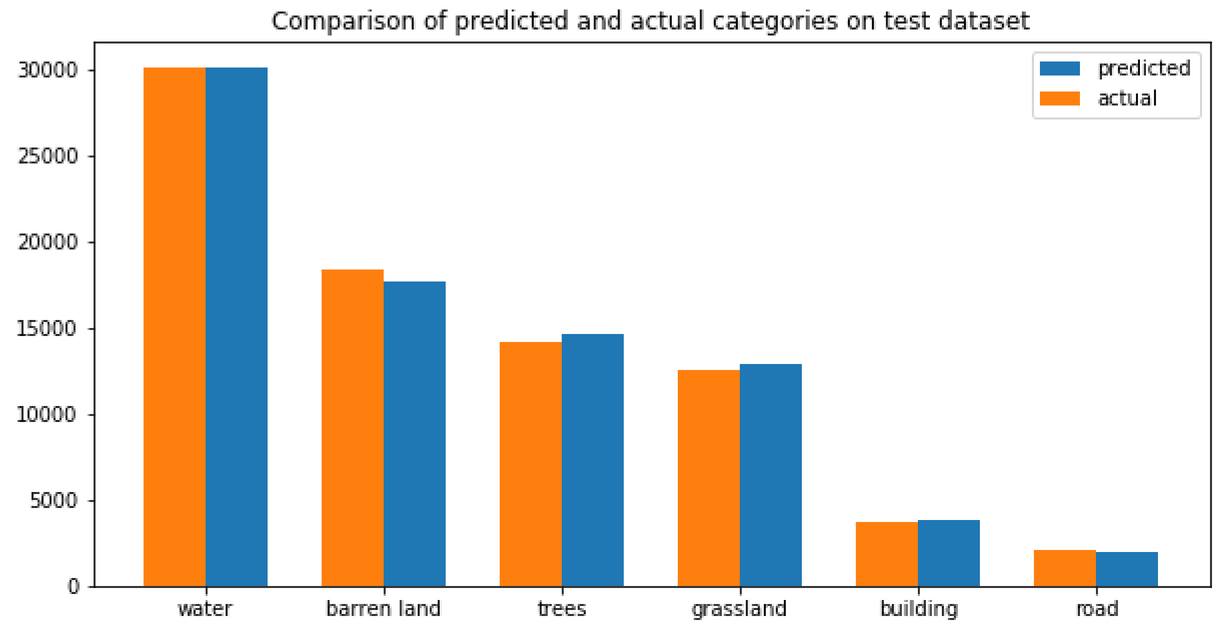


The training history plots suggested that training loss was slightly higher than the validation loss after each epoch. This counterintuitive result may be the result of applying regularization with a dropout layer. Regularization increases the ability of the model to generalize to unseen data at the expense of the accuracy of individual training predictions. Dropout is only applied during the training stage and not during validation, this can result in validation scores that exceed training scores. Also, training loss is continuously measured over an epoch, while validation loss is measured at the end of an epoch. This can produce higher training losses when the model fit improves over the epoch.

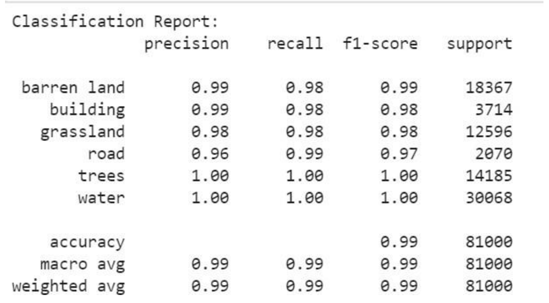
Baseline model evaluation on held-out data

The results for the validation set during training suggested that the model would generalize well to unseen data. This was tested by using the baseline model to predict the land use classification on a set of 81,000 labeled training images provided with the DeepSat-6 dataset.

The model predicted the image classes of the held-out set with 99.2% accuracy. The following bar chart compares the frequency of the labelled and predicted classes, and demonstrates that the classes are similarly distributed, with some misclassifications of barren land, trees, and grassland.



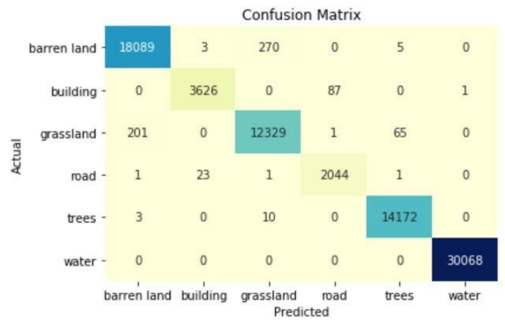
The classification report presents the precision (proportion of classifications into that class that were correct), recall (proportion of items in the class that were identified), and f1-score (harmonic mean of precision and recall) of each class. The results show slightly lower precision and recall for the grassland and road classes.



```
Classification Report:
              precision    recall  f1-score   support

 barren land       0.99      0.98      0.99     18367
    building       0.99      0.98      0.98      3714
   grassland       0.98      0.98      0.98     12596
        road       0.96      0.99      0.97      2070
       trees       1.00      1.00      1.00     14185
       water       1.00      1.00      1.00     30068

    accuracy                           0.99     81000
   macro avg       0.99      0.99      0.99     81000
weighted avg       0.99      0.99      0.99     81000
```



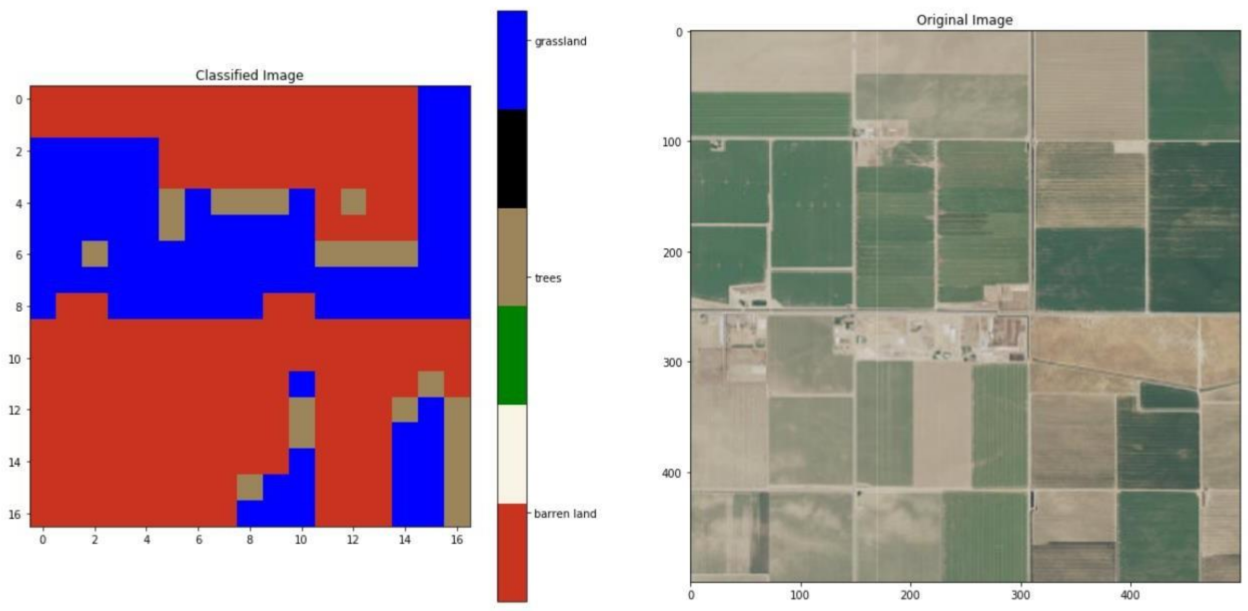
Confusion Matrix

Actual \ Predicted	barren land	building	grassland	road	trees	water
barren land	18089	3	270	0	5	0
building	0	3626	0	87	0	1
grassland	201	0	12329	1	65	0
road	1	23	1	2044	1	0
trees	3	0	10	0	14172	0
water	0	0	0	0	0	30068

A confusion matrix was also used to assess the baseline model's performance. The most common classification errors are barren land misclassified as grassland, and grassland misclassified as trees.

Applying the Baseline Model to New Imagery

Since the DeepSat-6 dataset does not provide location data for the image tiles, the resulting classifications could not be ground-truthed against the original imagery or displayed concurrently with other spatial datasets. To get around this limitation, randomly chosen the image from internet and applied my base line model.



References:

- https://www.kaggle.com/mrpinky/simple-cnn-accuracy-98-95-after-10-epochs