
Classifying Deepsat-6 Satellite Images Using Convolution Neural Networks

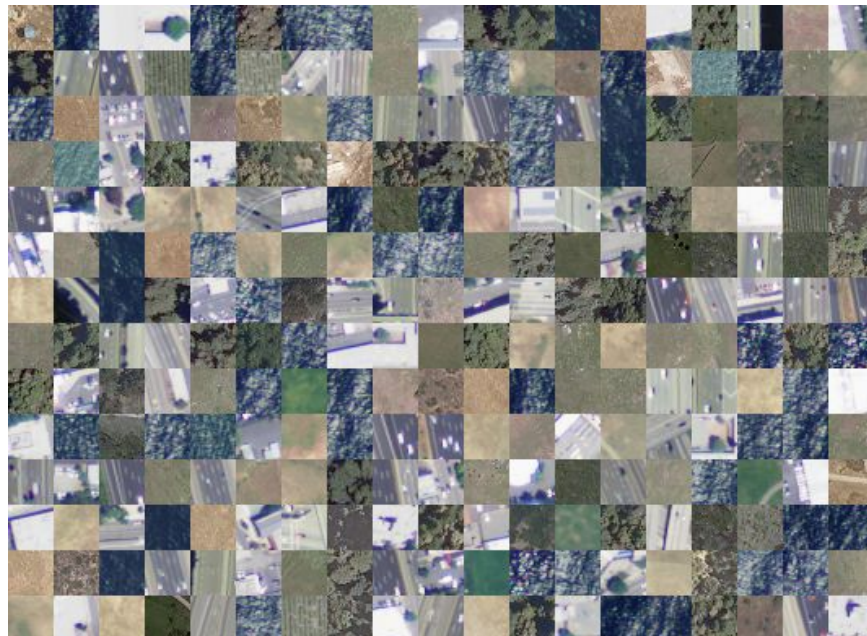


Table of Contents

Table of Contents	1
Introduction	2
Problem statement	2
Deep-Sat6 Dataset	3
Machine Learning for Image Classification	7
Deep Learning for Image Classification	9
Different CNN models	12
Evaluating Model Performance	14
Feature Visualizations	16
Making predictions on new images	18
Conclusions	22
Appendix	24

Introduction

Mapping land cover types is one of the primary methods to track the actual land use activity across the earth's surface. Understanding land cover and land use patterns also helps document how people are using land. Together, Land Use and Land Cover (LULC) maps help us track the trends and changes in land use patterns and this is particularly helpful for environmental conservation, resource management and land use planning.

Satellite images and GIS are some of the most commonly used data sources to understand and map land cover types. Satellite images with moderate to high resolution are now widely available across different parts of the earth's surface at different temporal resolutions. Modern satellite images are multi-million pixel arrays, represented by multispectral bands usually Red, Green, Blue and near infrared. The list of satellite imagery providers is extensive, and it includes both public and privately available datasets provided by such as Landsat, Sentinel missions, and Planet to name a few.

Most of the satellite imagery needs to be pre-processed by performing object detection, land use classification, boundary-detection, or semantic segmentation to gain actionable insights.

The fields of Deep Learning and Computer Vision have recently gained a lot of attention, and application of these fields to satellite imagery have led to the automation of these processing tasks resulting in cost reduction and improvement in accuracy of interpretation.

Problem statement

In this project, I'm using the Deep-Sat6 satellite dataset which contains images extracted from the National Agriculture Imagery Program (NAIP) dataset to build a multi-class classification algorithm using convolution neural networks to assign a label for each image among several labels.

The most important use of this project would be for environmental conservationists to use the model developed here to identify areas previously classified as trees or forests that have been converted to other land types such as barren land, agriculture or urban areas. This model can also be used to identify barren lands and grasslands that can be reforested. Another useful application of this work is for construction agencies and real estate management companies to identify barren land next to existing urban areas for future development.

Deep-Sat6 Dataset

The dataset used in this project was obtained from Kaggle. The dataset itself comes from the National Agriculture Imagery Program (NAIP) dataset. The NAIP dataset consists of a total of 330,000 scenes spanning the whole of the Continental United States (CONUS).

The SAT-6 dataset consists of 405,000 image patches each of size 28x28 and covering 6 land cover classes - barren land, trees, grassland, roads, buildings and water bodies in California. There are 324,000 training images with labels and 81,000 test images with labels consisting of one-hot encoded vectors. The training and test sets were selected from disjoint NAIP tiles. Each image is a 28 by 28 pixel image with 4 channels - Red, Green, Blue and Near InfraRed (NIR).

The entire dataset in Kaggle consisted of csv files for the training and test images, csv files training and test labels, and the original MAT files for the training and test images and labels.

The `DeepSat6-LoadData.ipynb` notebook contains the code to load the dataset.

Deep Dive into Deep-Sat6

Structure of Data:

Next, let's take a closer look at the DeepSat6 dataset.

The Deep-sat6 dataset are stored as dictionaries containing key-value pairs for the training and test images and labels, as well as annotations. The dataset is already split into testing and training data and the shape of each dataset is provided below.

Dataset type	Shape
train_X	(28, 28, 4, 324000) -> 28 by 28 images with 4 channels
test_X	(28, 28, 4, 81000) -> 28 by 28 images with 4 channels
train_y	(6, 324000) -> 6 by 1 vector of labels
test_y	(6, 81000) -> 6 by 1 vector of labels

annotations	(6,1) -> label annotations
-------------	----------------------------

First, individual nparrays were created for the training and test images and labels from the .mat file. Next, the training and test arrays were reworked to move the position of the last axis to the first which converted the arrays from (# samples, image width, image height, #channels) format to (image width, image height, #channels, #samples). The training and test labels were transposed to be of type (#samples, one-hot encoded vectors). A dictionary with values as land use categories (in text) were then created from the annotations, and the training and test labels were mapped to these land categories.

Occurrence of Labels:

Figure 1 contains some sample images from each land use type. **Figure 2** contains a distribution of the label instances in the training and test set. 'Water' is the most commonly occurring image in the dataset while roads and buildings are the least common images. The unequal distribution of images by land use type will likely result in a class imbalance issue while building the Convolution Neural Network (CNN) [Ref]. We will address this issue in the next sections.

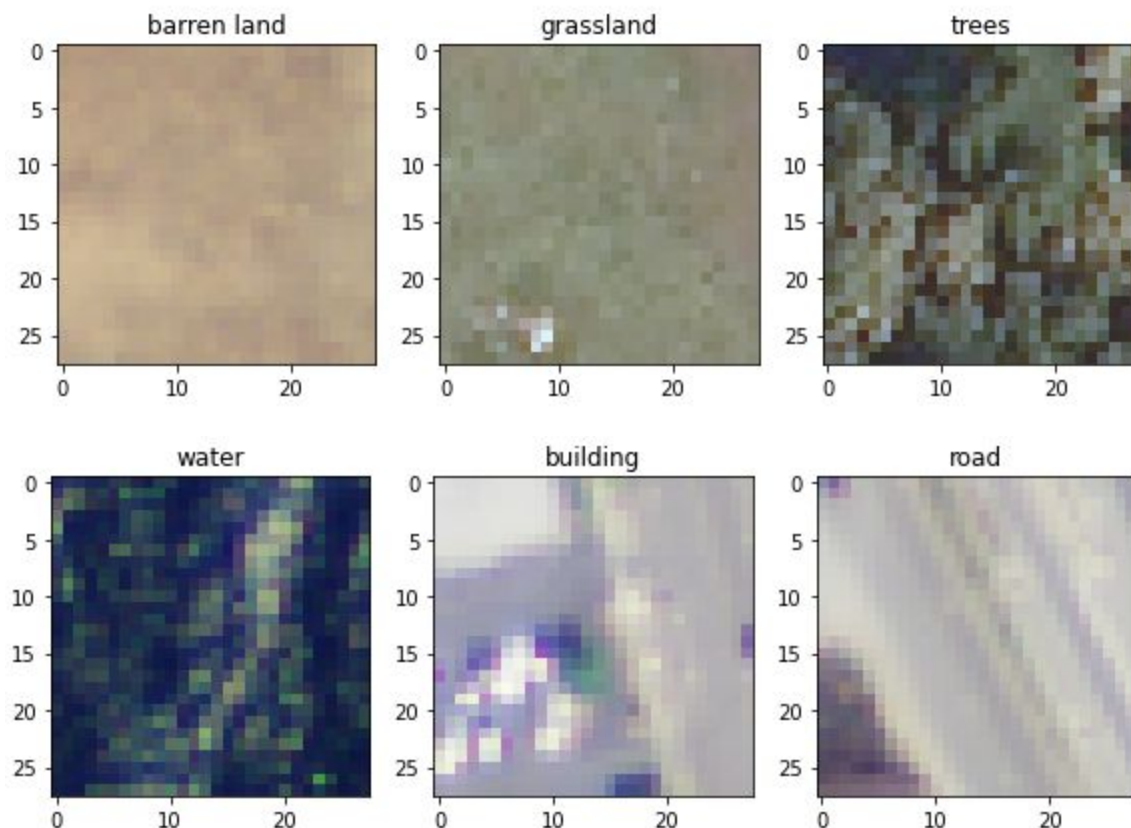


Figure 1: Sample images from each land use type

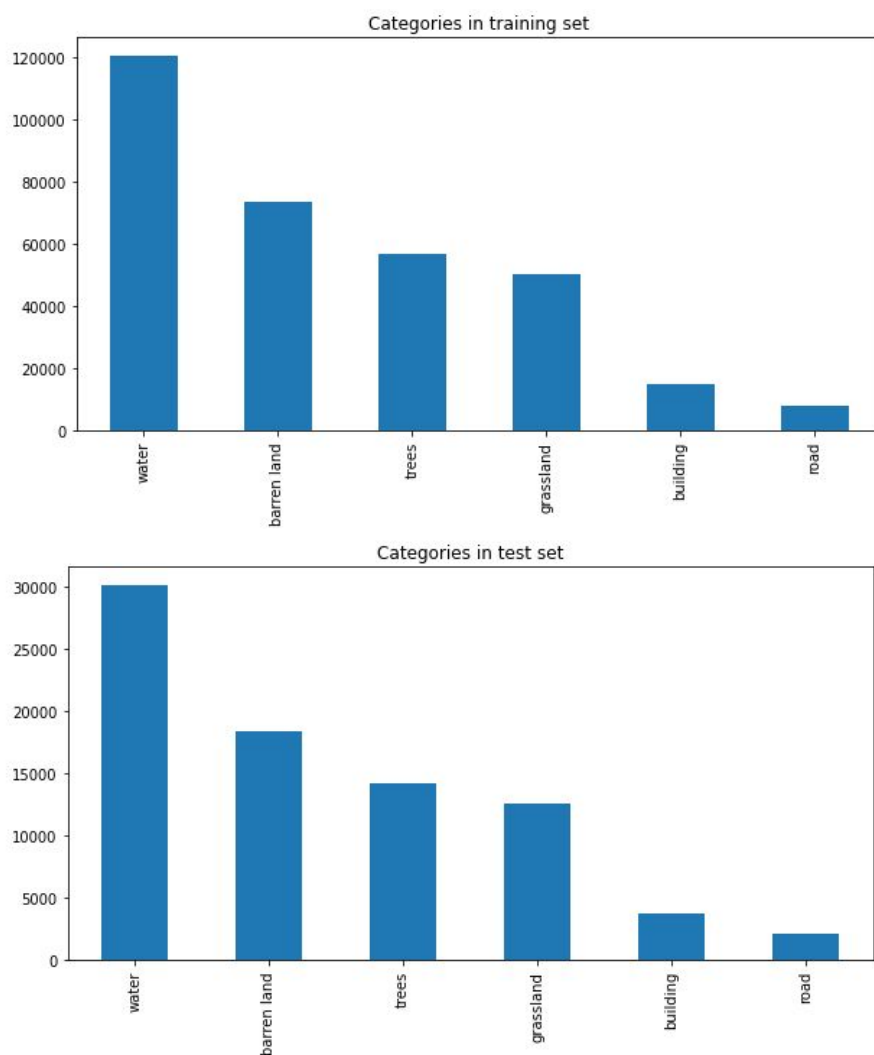


Figure 2: Number of images by land use type in training and test set

Decomposing Images:

Figure 3 contains the average RGB values of all images belonging to a class type. As seen in the Figure, red is more dominant in barren lands, while is slightly more dominant in vegetation classes such as grassland and trees. Blue is more dominant in the water land class.

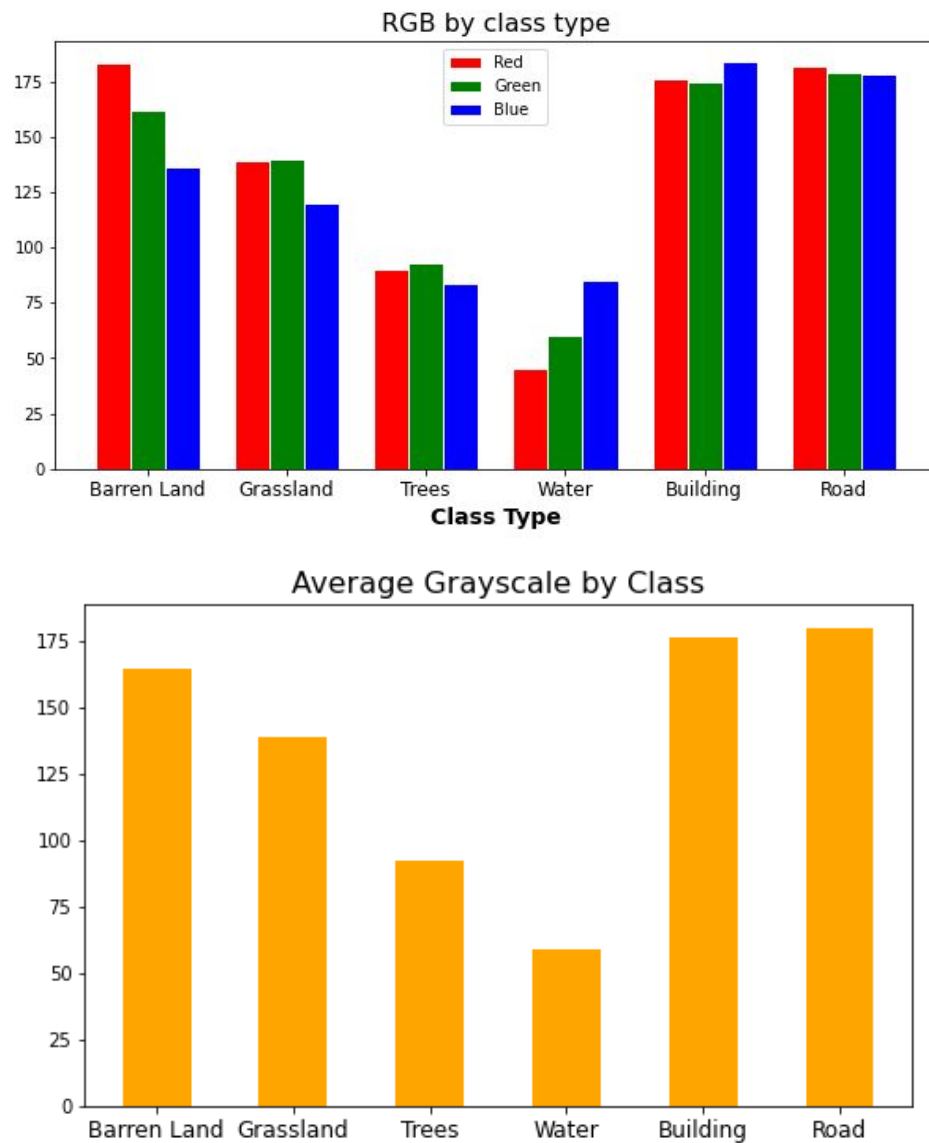


Figure 3: Average RGB and Average Grayscale by Class Type

Figure 4 contains images representing the average RGB channels calculated on a pixel by pixel basis for each label in the training dataset. The average RGB by pixel somewhat reflects what we observed

in Figure 3, the water land class had more blue shade, while buildings and roads had equal shades of RGB appearing as a light color.

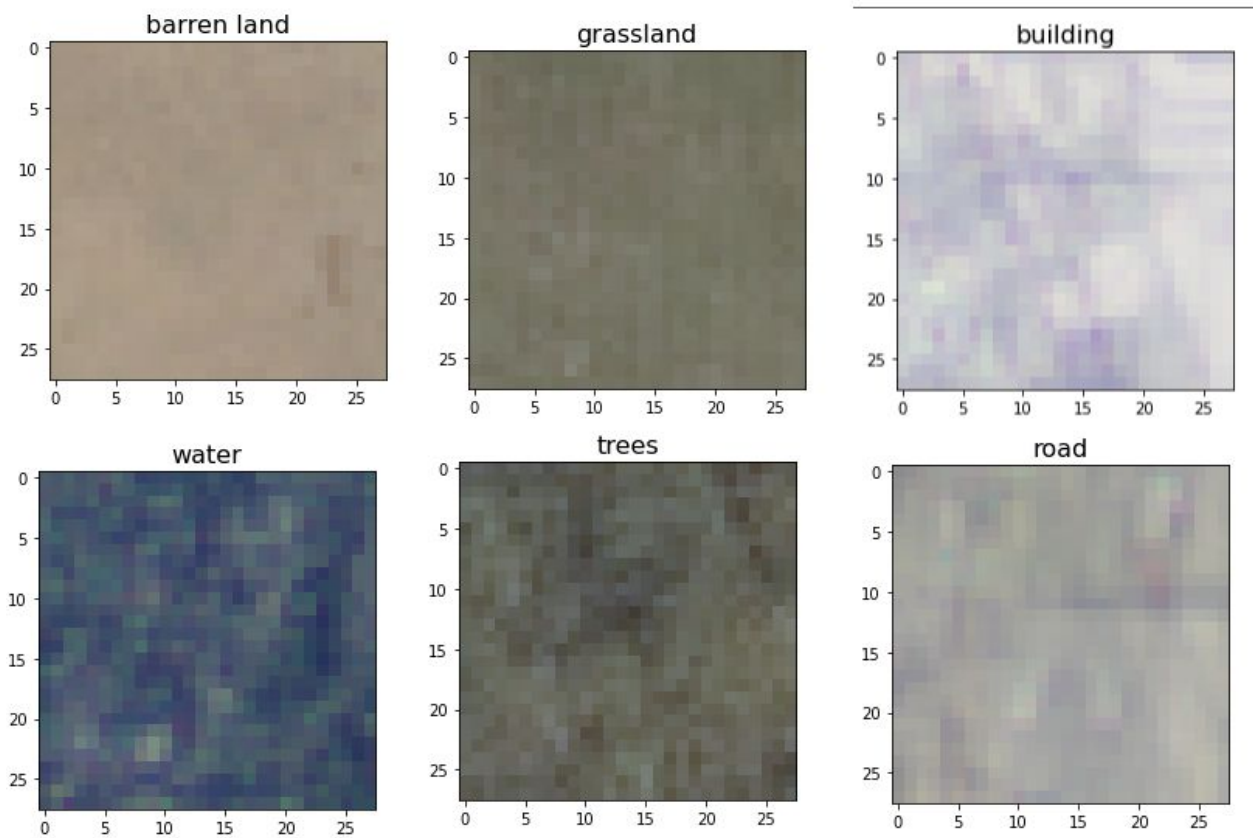


Figure 4: Average image by Class Type

Machine Learning for Image Classification

As a first step, a simple RandomForest approach was used to build a baseline model that can classify images into the different class types. The performance of the baseline random forest model will be compared against the Convolution Neural Network (CNN) models.

The Random Forest model was first trained on a subset of the training data instead of all the 324,000 images. First, a function to read a subset of the training and test csv files was created. 20,000 training images were used to train the random forest model, with 100 estimators and a max depth of 10.

```
RandomForestClassifier(n_estimators=100, max_depth = 10)
```

Class Type	f1-score
------------	----------

Barren Land	0.96
Building	0.93
Grassland	0.87
Road	0.75
Trees	0.94
Water	0.99

Table 1: F1 Scores for Baseline Random Forest

Table 1 contains the F-1 scores for the random forest model. The model does a good job of predicting water, and barren land but does not do a great job at predicting grasslands, since it is misclassifying grasslands or trees as shown in **Figure 5**.

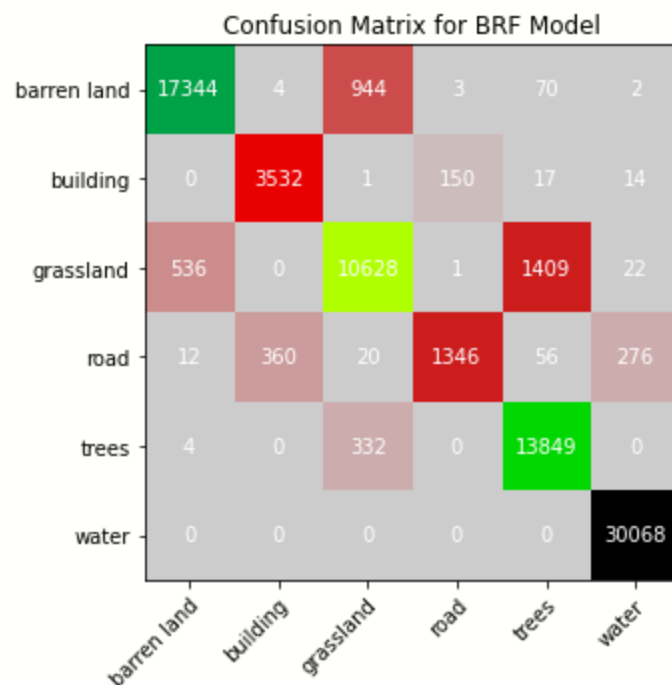


Figure 5: Confusion Matrix for Baseline Random Forest

The metric chosen for model evaluation was the balanced accuracy instead of the overall accuracy. The balanced accuracy was chosen as a metric because the training dataset contains an imbalance of classes, and balanced accuracy is a better judge of performance across classes in the imbalanced class setting.

Balanced accuracy is based on two more commonly used metrics: **sensitivity** (also known as **true positive rate** or **recall**) and **specificity** (also known as **false positive rate**). **Balanced accuracy is also the average of recall across all classes.**

Sensitivity is calculated as: $\frac{TP}{(TP+FN)}$

Specificity is calculated as: $\frac{TN}{(FP+TN)}$

Balanced accuracy: $\frac{Sensitivity + Specificity}{2}$

Balanced accuracy score = 89.69%

Number of mis-matched predictions = 4,208

The code for the Random Forest approach is in the 'DeepSat6-RandomForest.ipynb'.

Deep Learning for Image Classification

Convolution Neural Networks (ConvNet/CNNs) is a Deep Learning algorithm which takes an image as input, simplifies it through various "convolutions", and assigns weights and biases to various aspects of the image to classify them. CNNs have been successfully applied in satellite image classification problems [\[Ref\]](#).

Outline of Approach:

This section talks about the CNNs that were built for classifying the Deep-Sat6 images. These CNNs were built using Keras with TensorFlow backend in Google Colab. The different CNNs that were built and tested are as follows:

- 1) Simple CNN built from scratch - **Baseline CNN model (B-CNN)**
- 2) Transfer Learning using VGG16 by padding input image **(TL-1)**
- 3) Transfer Learning using VGG16 by upsampling input image **(TL-2)**
- 4) Transfer Learning using VGG16 with fine tuning and padding **(TL-3)**

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford [\[Ref\]](#). The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. The VGG-16 is a commonly used CNN architecture for satellite image classification. The VGG-16 architecture is shown below.

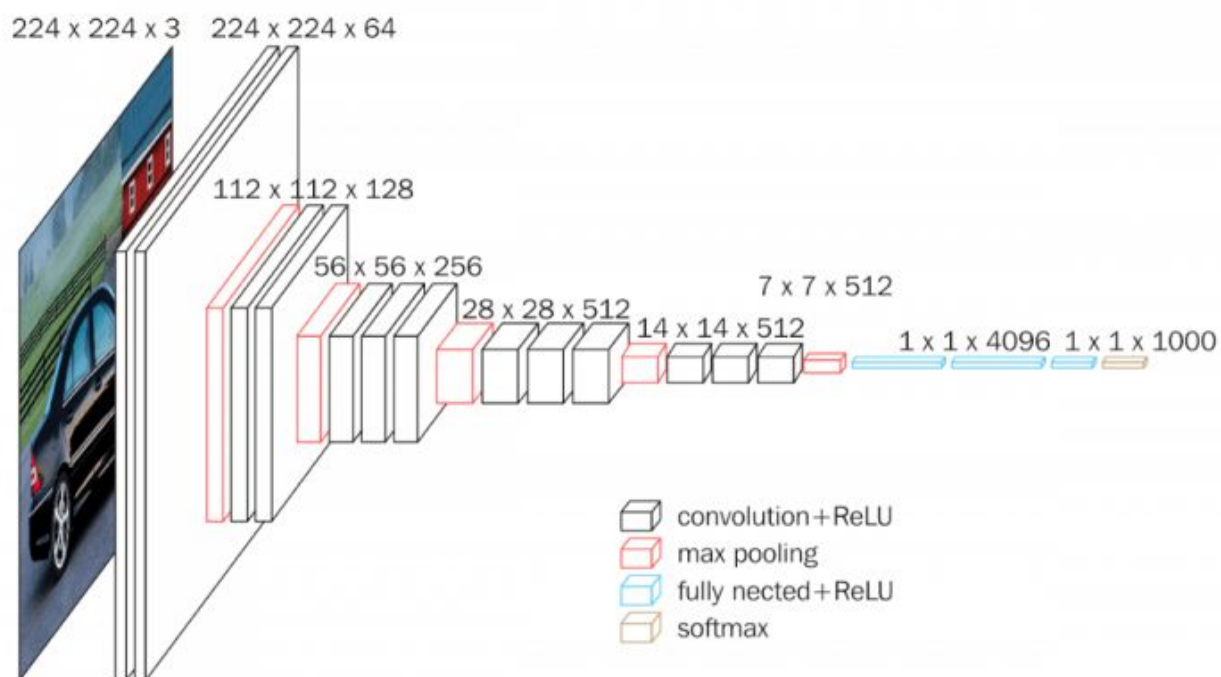


Figure 6: VGG-16 architecture

Before getting into the details of each model, let's first discuss the deep-learning data pipeline, metric for model evaluation, feature visualization method, and how each CNN model was used to perform a multi-label classification even though the models were built to predict only a single label for each image.

Deep-Learning Data Pipeline and Hyperparameters:

Split training data into training and validation set

For the CNNs, the entire training dataset was used for training the CNN models. TensorFlow's image preprocessing function 'ImageDataGenerator' was used to generate batches of tensor image data with real-time data augmentation and to split the training data into training and validation sets. The images were augmented by setting 'horizontal_flip' to true and the 'validation_split' was set to 0.2. Each image was also rescaled by dividing the RGB values by 255. The original images consist of RGB coefficients in the 0-255, but such values would be too high for the CNN models to process (given a typical learning rate), so we target values between 0 and 1 instead by scaling with a 1/255. factor.

Model compilation and Model checkpoint:

For all the CNN models, the **Adam optimizer** with a **categorical_crossentropy** as the loss function was used. The models were optimized on the loss function and evaluated on metrics including 'accuracy', the custom built 'f1 score', 'prediction', 'recall' and 'balanced accuracy'.

An early stopping method was applied to remove the need to manually set the number of training epochs. This is a type of regularization method in that it can stop the network from overfitting. At the end of every epoch, the network performance is evaluated on the validation set and if the network outperforms the previous model, a copy of the network along with weights is saved. The idea of adding simple stopping is to prevent overfitting on training data and determine the model with best test accuracy.

In this case, we monitor the *validation balanced accuracy score* in each epoch. Some other parameters for fitting the model are set below:

Steps_per_epoch = 150

Epochs = 20

Evaluating Model Performance:

Similar to the Random Forest approach, the model performance for each model is evaluated based on the **overall balanced accuracy score**.

For further context, a few other metrics are also evaluated in order to further understand the performance of each model, including - **Number of predicted vs actual images** in each category in the test dataset, **Classification report and confusion matrix** and **Prediction error**. Each of these metrics are discussed separately for the different CNN models.

Feature Visualization:

In order to visualize features from the different CNN models, the following approach was used - **Visualizing Filters by maximizing the activation of a specific filter in a target layer**. In this method, we follow the approach in the code provided [here](#). The approach is to maximize the activation of a specific filter in a target layer and represent a visualization of the pattern that filter corresponds to. We do this by feeding in a random image to the network, and the image is updated progressively to maximize the filter's activation function. The resulting image gives us an insight into what the network's filter is "looking

for” or identifying in each image. Results of this method for each CNN model will be discussed further in detail.

Different CNN models

This section discusses the model architecture for the different CNN models, their performance on the test set based on the metrics defined above as well as their performance to classify new images.

1) Baseline CNN model (B-CNN)

The detailed model architecture for the baseline CNN model is shown in the Appendix. The input image to the baseline CNN model is a **28 by 28** image with only 3 bands. The NIR band is not included in this model.

Model architecture:

- Input image = (28,28,3)
- Convolution layer = 3 layers with (3,3) kernel size
- Maxpooling2D = 2 layers
- Dropout = 2 layers with dropout size of 0.2 and 0.5
- Flatten layer = (256x1)
- Dense layers = 2 layers, final layer = (6x1) output with ‘softmax’ activation
- Trainable parameters = 57,524
- Activation function = Rectified Linear Unit (ReLU) activation

2) Transfer Learning using VGG16 by padding input image (TL-1)

The detailed model architecture for TL-1 is shown in the Appendix. Input image is a **32 by 32** image with only 3 bands.

Model architecture:

- Original image = (28,28,3)
- Padding layer = ZeroPadding layer to convert (28,28,3) -> (32,32,3) smallest image size for a VGG-16 architecture [\[Ref\]](#)
- Input image = (32,32,3)
- Next layer = VGG 16 layers, all parameters are non-trainable
- Flatten layer = (512)
- Dropout layer = 0.5 drop out rate
- Dense layers = 2 layers, final layer = (6x1) output with ‘softmax’ activation
- Trainable parameters = 265,734

3) Transfer Learning using VGG16 by Upsampling input image (TL-2)

The detailed model architecture for TL-2 is shown in the Appendix. This transfer learning approach is similar to the TL-1 model, except that the input images are converted from 28x28x3 dimension to 56x56x3 by using the 'Upsampling3D' approach instead of 'ZeroPadding'.

Model architecture:

- Original image = (28,28,3)
- Upsampling 3D layer = Convert (28,28,3) -> (56,56,3) image as input to VGG-16 architecture [\[Ref\]](#)
- Input image = (56,56,3)
- Next layer = VGG 16 layers, all parameters are non-trainable
- Flatten layer = (512)
- Dropout layer = 0.5 drop out rate
- Dense layers = 2 layers, final layer = (6x1) output with 'softmax' activation
- Trainable parameters = 66,438

4) Transfer Learning using VGG16 with fine tuning and padding (TL-3)

In this approach, we follow the same architecture as the TL-1 model, but in addition to changing the last layer to match the number of classes in our dataset, we also retrain the layers of the network that we want. The detailed model architecture for TL-3 is shown in the Appendix.

Model architecture:

- Original image = (28,28,3)
- Padding layer = ZeroPadding layer to convert (28,28,3) -> (32,32,3) smallest image size for a VGG-16 architecture [\[Ref\]](#)
- Input image = (32,32,3)
- Next layer = VGG 16 layers, all layers except the last convolution layer (block 5) and classification stages are retained (set as non-trainable). Only block 5 is set to trainable.
- Flatten layer = (512)
- Dropout layer = 0.5 drop out rate
- Dense layers = 2 layers, final layer = (6x1) output with 'softmax' activation
- Trainable parameters = 7,079,424

In this case, we will allow the last convolution layer (block 5) and classification stages to

Evaluating Model Performance

The model performance for different models were evaluated based on the balanced accuracy scores and F1 scores for the different models.

Metric	B-CNN	TL-1	TL-2	TL-3
Balanced Accuracy	93.11%	91.83%	90.69%	95.47%

Comparison of F1 scores for the different models are shown below:

Class Type	B-CNN	TL-1	TL-2	TL-3
Barren Land	0.96	0.94	0.89	0.96
Building	0.93	0.92	0.90	0.94
Grassland	0.92	0.88	0.83	0.93
Road	0.83	0.84	0.88	0.91
Trees	0.97	0.95	0.94	0.98
Water	1.00	0.99	0.99	1.00

Table 2: Balanced Accuracy Scores and F1 Scores for CNN Models

Table 2 compares the balanced accuracy score and F1 scores for the different CNN models. The model that yielded the highest balance accuracy score is the Transfer Learning approach with VGG16 and fine tuning, followed by the Baseline CNN model built from scratch.

The F1 scores for all land classes except 'Buildings' are higher for the transfer learning with fine tuning model (TL-2) than the baseline CNN model (B-CNN). The F1 scores for grasslands and roads are lower than the rest of the land classes indicating that these labels are being predicted incorrectly.

The confusion matrices for the different models are shown in **Figure 7**.

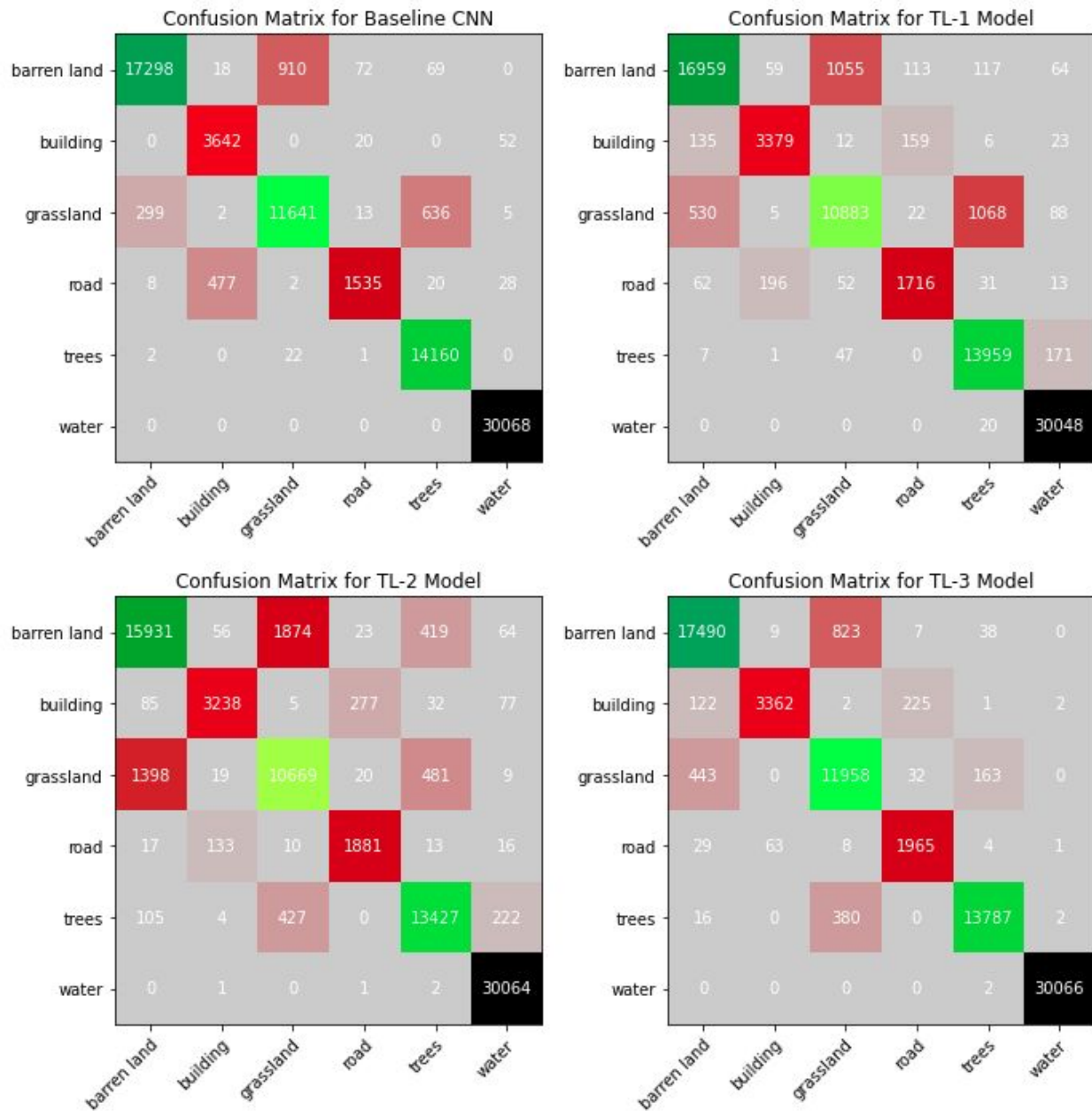


Figure 7: Confusion Matrices for different models

Figure 7 indicates that 'grasslands' are being misclassified as barren lands or trees, and in some cases, roads are being misclassified as buildings.

The models were also evaluated on their prediction errors on the test set i.e. number of mis-classified images on the test data. The table below shows the number of mis-classified images and a breakdown of the mis-classifications by land use category for each model.

Feature Visualizations

Feature visualizations for the different models were performed by selecting a target layer (convolution layers) in each model and maximizing the activation for a specific filter in that layer.

Figure 8 shows a comparison of the filter visualizations for the Baseline CNN, TL-1, TL-2 and TL-3 models. The filters shown below for the different models are from different convolution layers in the model. The initial layers of a CNN extract higher level features from the image and use fewer filters. The deeper layers in the network use much more filters (two or thrice the previous layer) and learn more complex features. But the deeper layers are computationally more intensive than the initial layers.

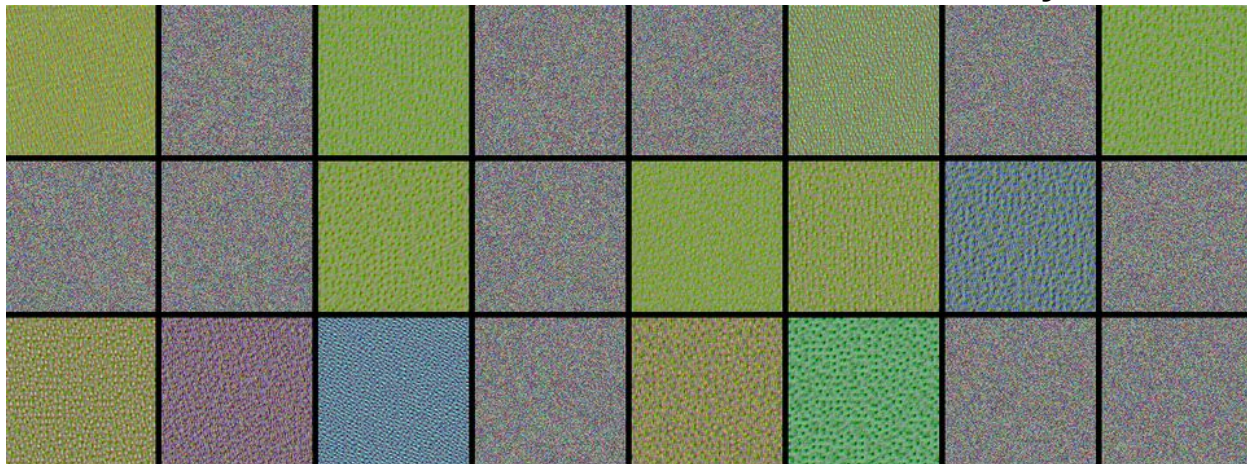
The filter shown below for the *Baseline CNN model* is from the second convolution layer and these filters could represent colors in the input images such as green for trees and grasslands, blue for water and the brown shades for barren land, or grasslands. The first few layers basically encode direction and color in images. These direction and color filters then get combined into basic grid and spot textures. These textures gradually get combined into increasingly complex patterns.

The filters shown below for the *TL-1 model* look very different from the baseline CNN convolution layer and are now detecting directions as well as colors. The filters could represent patterns in the input images such as water or the vertical lines that represent roads. The layers presented here are the first few convolution layers which encode just colors and directions.

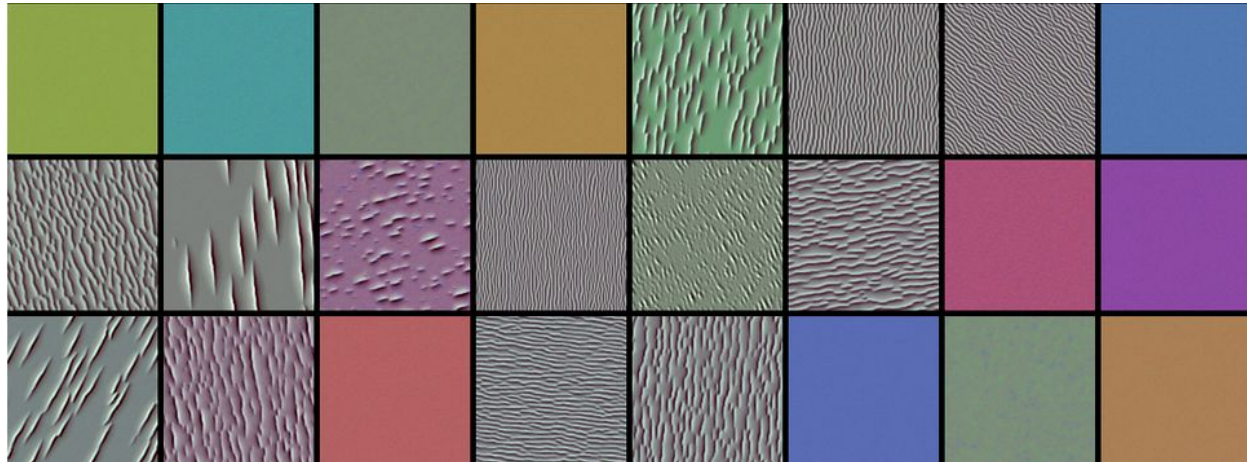
The filters shown below for the TL-2 model are from the third convolution layer, and similar to TL-1, these filters also detect both direction as well as colors of the vgg16 model.

The last visualization shown in **Figure 8** is for the TL-3 model which is for the 13th convolution layer. This represents a much deeper layer in the architecture and the filter is now detecting more complex features and patterns. Since the filter shown in this model was originally trained on the ImageNet dataset, we can start to see patterns such as faces, or animals that this filter is detecting.

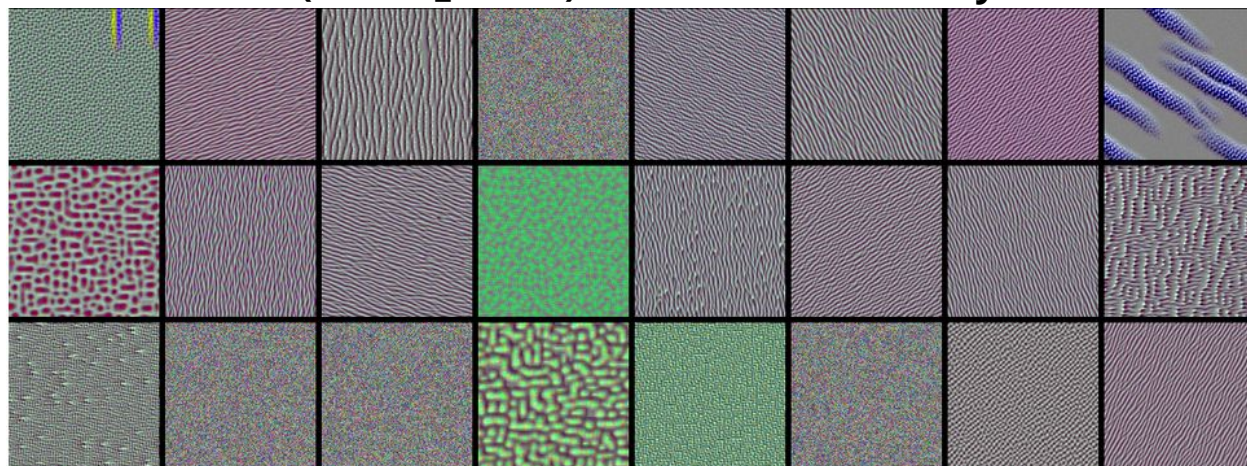
Baseline CNN (Conv2d_2) Second Convolution Layer



TL-1 (Block1_Conv2) Second Convolution Layer



TL-2 (Block2_Conv1) Third Convolution Layer



TL-3 (Block5_Conv3) Thirteenth Convolution Layer

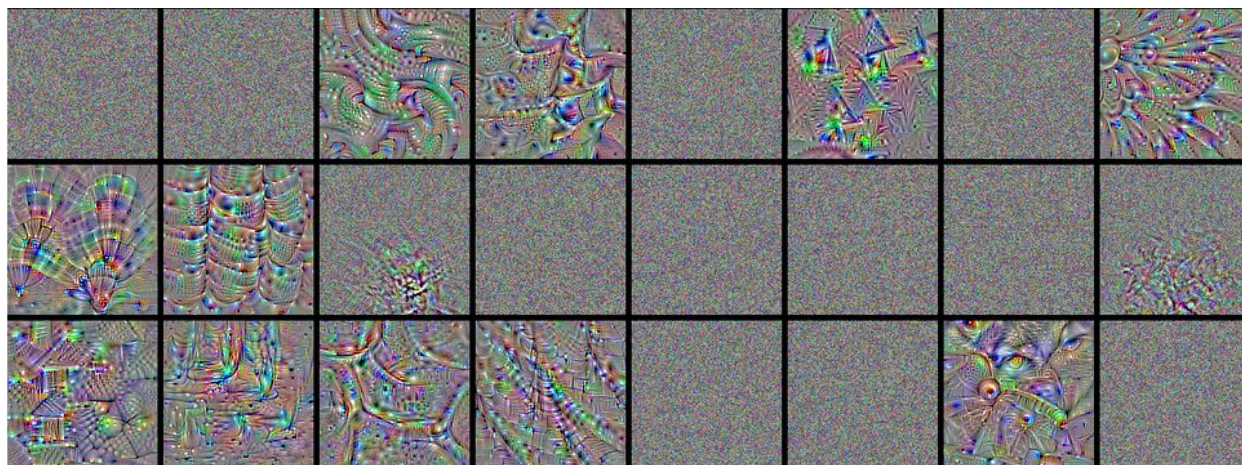


Figure 8: Filter Visualizations - Baseline CNN, TL-1, TL-2 and TL-3 models

Making predictions on new images

Each CNN's model performance was also tested by making predictions on a completely new NAIP image. First, an NAIP image in Northern California was obtained from the [USGS National Map Downloader](#) in JPEG2000 (JP2) format. An image processing pipeline was set up to convert this NAIP image to a format that can be input into the CNN models.

Step 1: Image in JP2 format was converted to a PNG using `gdal_translate` command line function

Step 2: The original image of size (11606, 15729) was resized to a (5600, 5600) image using the `Pillow (PIL)` library and setting the filter to `ANTIALIAS` to get the highest image quality.

Step 3: To perform a multi-label classification using a model built for predicting single labels, the resized multiband image was split into image tiles once again using the `gdal_translate` function. The resized (5600,5600) was split into 40,000 (28 by 28) images which is the input shape of images to the CNN models. The output (28,28) image was stored as a .jpg file on Google Drive.

Step 4: Next, a function was defined to loop through each .jpg file, and make predictions on each image based on the trained CNN models. For each image, the predictions and tags (tile numbers) were returned.

Step 5: Next a dataframe containing each prediction and tile number was created, with 5600 columns and 5600 rows. Each image class type was assigned a color to be able to plot the predicted values on an image

Step 6: Next, the predicted image class for the resized image was compared against the original image to visualize the quality of predictions.

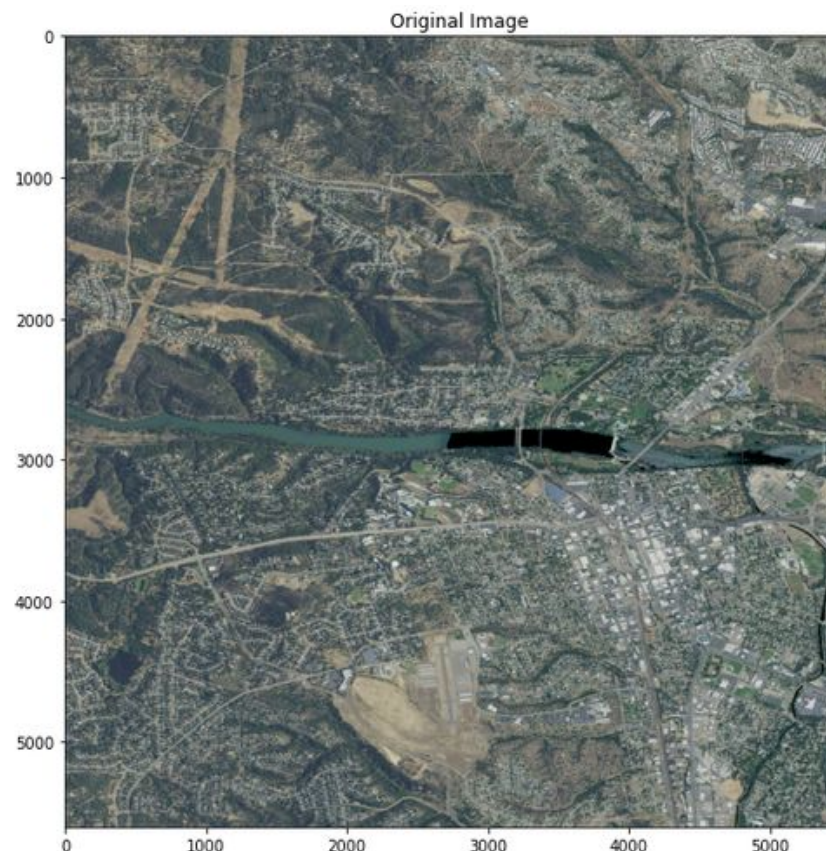


Figure 12: Original Image

The image prediction for the Baseline CNN model, TL-1 and TL-3 models are shown below. Overall, the Baseline CNN model is doing a better job of predicting new and unseen images at different resolutions compared to the TL-1 and TL-2 models.

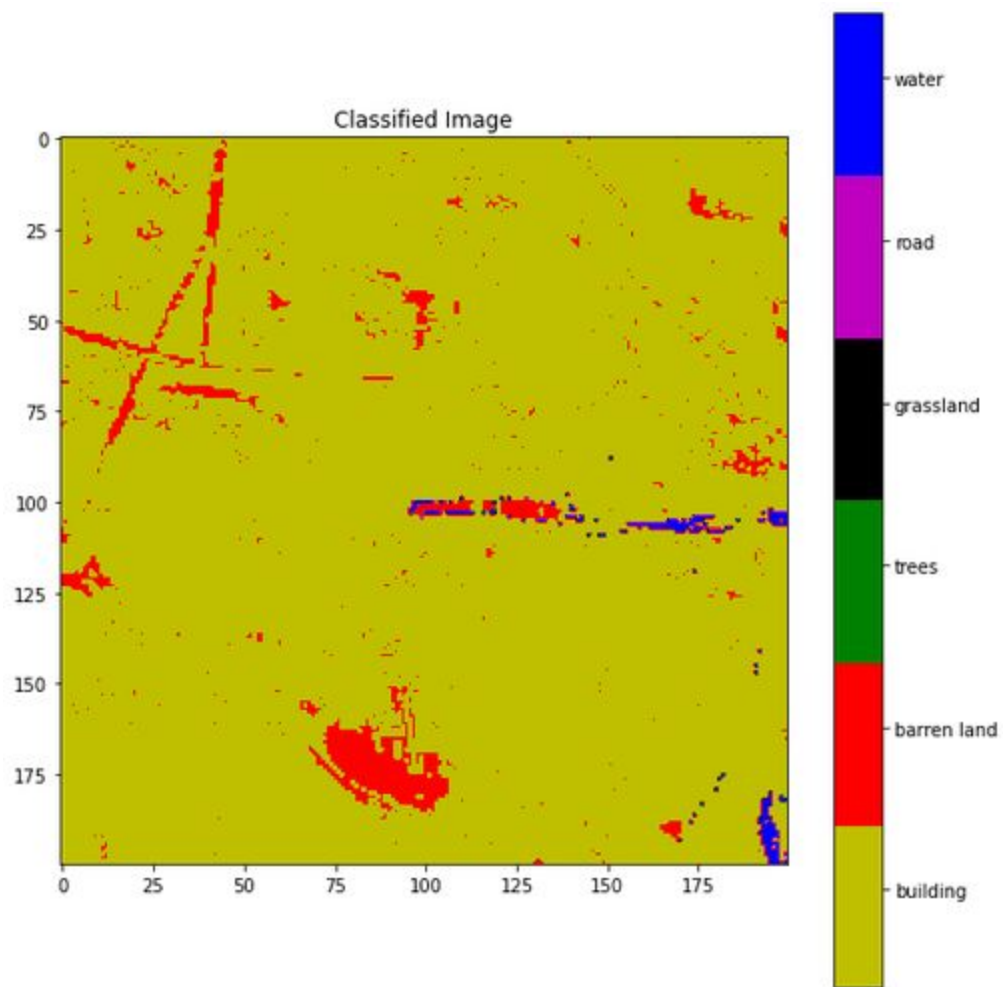


Figure 13: Image predictions using Baseline CNN Model

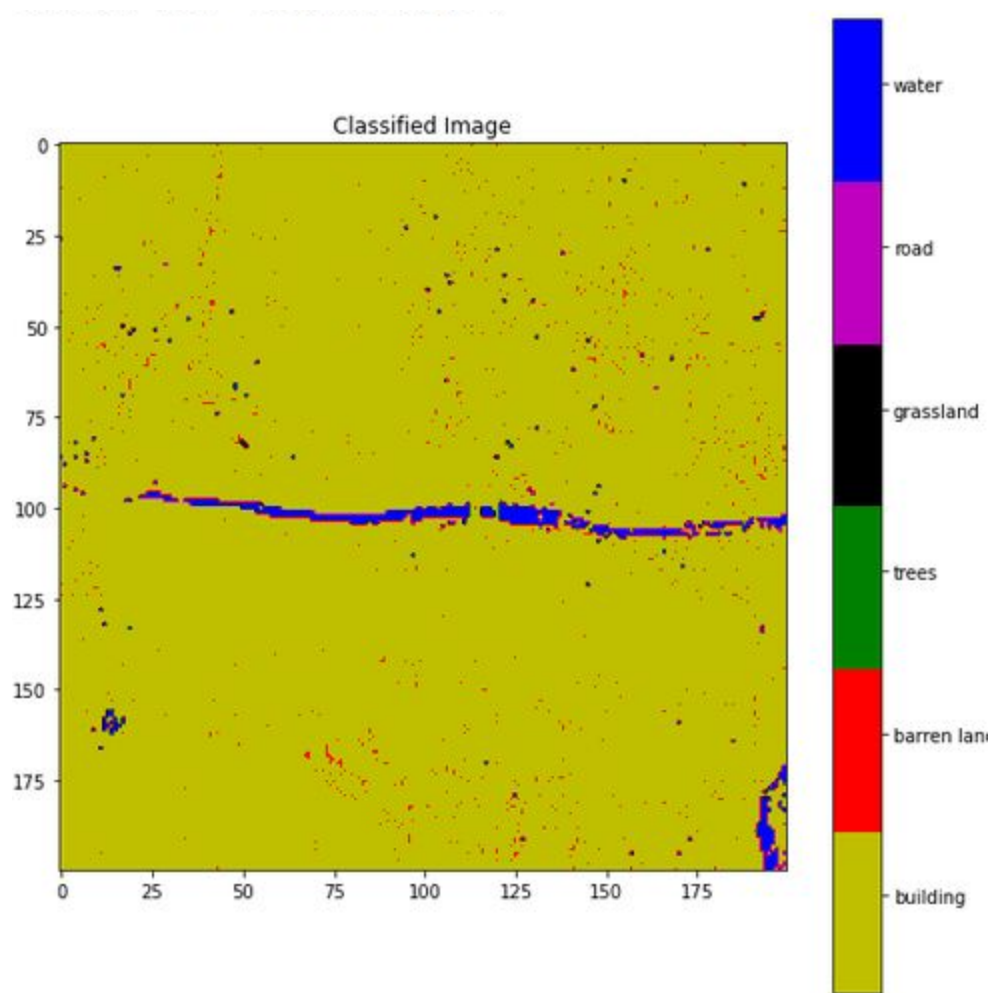


Figure 14: Image predictions using TL-1 Model

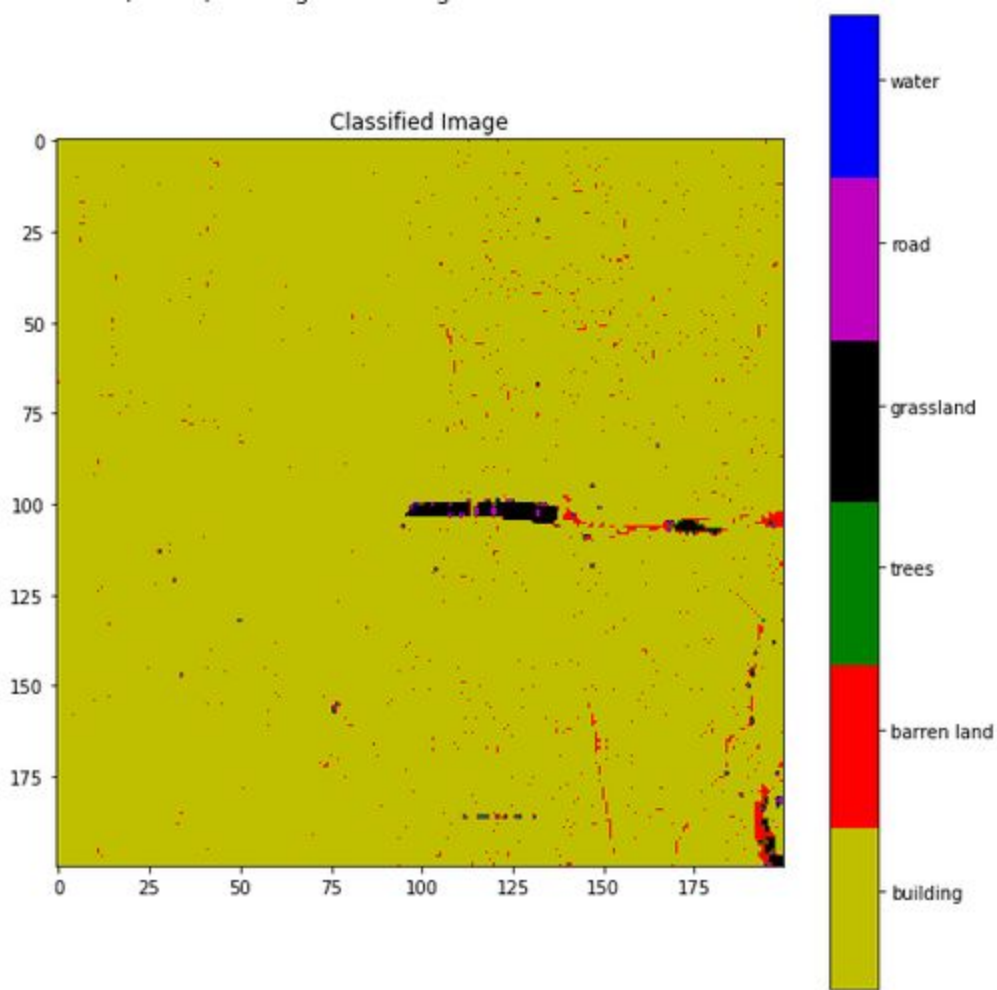



Figure 15: Image predictions using TL-3 Model

Conclusions

This work explored different CNN models for classifying the DeepSat-6 satellite image dataset. The original dataset contained 324,000 training images, 81,000 test images comprising 6 different land classes. Several different models were built and tested for performing image classification tasks including a simple random forest model, a baseline CNN model built from scratch, and three different transfer learning models using the VGG-16 architecture.

Since the dataset contained an uneven number of images for each class, a balanced accuracy score was used as the metric to evaluate the performance of each model. Overall,



the transfer learning model with a fine tuning of one of the layers resulted in the highest balanced accuracy of 95.47%. The transfer learning model with fine tuning had a deeper and a more complex network compared to the baseline CNN model. However, the transfer learning model did not do a great job of classifying image tiles obtained from the new NAIP image dataset. The TL-3 model misclassified water and grasslands in the new image dataset and it over-predicted on the building land use type. The baseline CNN model however, did a better job of classifying the image tiles from the new NAIP image, which shows that the Baseline CNN model is better at making predictions on images that the model has not seen before and at different resolutions.

The classification algorithm developed in this work can be used by environmental conservationists and the general public looking to classify satellite images into the different land use types. This project can be used to identify areas that have undergone land use change such as areas that were previously classified as trees or forests, and now converted to barren land, crops or urban areas.

The classification algorithms can also be used to identify areas that can be reforested. Finally, construction agencies and real estate management companies can identify areas for future development next to existing urban areas.

The next steps on this project would be to try and improve the F1 scores on the land use classes with low scores by trying our image augmentation and identifying optimal hyperparameters that can improve model performance. Finally, we could also use the model developed here to build an app that can classify land use types in satellite images.

Appendix

1) Baseline CNN Model Architecture

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 16)	448
conv2d_1 (Conv2D)	(None, 26, 26, 32)	4640
max_pooling2d (MaxPooling2D)	(None, 8, 8, 32)	0
conv2d_2 (Conv2D)	(None, 6, 6, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 2, 2, 64)	0
dropout (Dropout)	(None, 2, 2, 64)	0
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 6)	774
Total params: 57,254		
Trainable params: 57,254		
Non-trainable params: 0		

2) Transfer Learning using VGG16 by padding input image (TL-1)

Model: "sequential_4"

Layer (type)	Output Shape	Param #
=====		
zero_padding2d_4 (ZeroPaddin	(None, 32, 32, 3)	0
<hr/>		
vgg16 (Functional)	(None, 512)	14714688
<hr/>		
flatten_2 (Flatten)	(None, 512)	0
<hr/>		
dense_4 (Dense)	(None, 512)	262656
<hr/>		
dropout_2 (Dropout)	(None, 512)	0
<hr/>		
dense_5 (Dense)	(None, 6)	3078
=====		

Total params: 14,980,422

Trainable params: 265,734

Non-trainable params: 14,714,688

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 32, 32, 3)]	0
block1_conv1 (Conv2D)	(None, 32, 32, 64)	1792
block1_conv2 (Conv2D)	(None, 32, 32, 64)	36928
block1_pool (MaxPooling2D)	(None, 16, 16, 64)	0
block2_conv1 (Conv2D)	(None, 16, 16, 128)	73856
block2_conv2 (Conv2D)	(None, 16, 16, 128)	147584
block2_pool (MaxPooling2D)	(None, 8, 8, 128)	0
block3_conv1 (Conv2D)	(None, 8, 8, 256)	295168
block3_conv2 (Conv2D)	(None, 8, 8, 256)	590080
block3_conv3 (Conv2D)	(None, 8, 8, 256)	590080
block3_pool (MaxPooling2D)	(None, 4, 4, 256)	0
block4_conv1 (Conv2D)	(None, 4, 4, 512)	1180160
block4_conv2 (Conv2D)	(None, 4, 4, 512)	2359808
block4_conv3 (Conv2D)	(None, 4, 4, 512)	2359808
block4_pool (MaxPooling2D)	(None, 2, 2, 512)	0
block5_conv1 (Conv2D)	(None, 2, 2, 512)	2359808
block5_conv2 (Conv2D)	(None, 2, 2, 512)	2359808
block5_conv3 (Conv2D)	(None, 2, 2, 512)	2359808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0
global_max_pooling2d_4 (Glob	(None, 512)	0
Total params: 14,714,688		
Trainable params: 0		
Non-trainable params: 14,714,688		

3) Transfer Learning using VGG16 by Upsampling input image (TL-2)

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
up_sampling3d (UpSampling3D)	(None, 56, 56, 3)	0
vgg16 (Functional)	(None, 1, 1, 512)	14714688
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 128)	65664
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 6)	774
=====		

Total params: 14,781,126

Trainable params: 66,438

Non-trainable params: 14,714,688

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 56, 56, 3)]	0
block1_conv1 (Conv2D)	(None, 56, 56, 64)	1792
block1_conv2 (Conv2D)	(None, 56, 56, 64)	36928
block1_pool (MaxPooling2D)	(None, 28, 28, 64)	0
block2_conv1 (Conv2D)	(None, 28, 28, 128)	73856
block2_conv2 (Conv2D)	(None, 28, 28, 128)	147584
block2_pool (MaxPooling2D)	(None, 14, 14, 128)	0
block3_conv1 (Conv2D)	(None, 14, 14, 256)	295168
block3_conv2 (Conv2D)	(None, 14, 14, 256)	590080
block3_conv3 (Conv2D)	(None, 14, 14, 256)	590080
block3_pool (MaxPooling2D)	(None, 7, 7, 256)	0
block4_conv1 (Conv2D)	(None, 7, 7, 512)	1180160
block4_conv2 (Conv2D)	(None, 7, 7, 512)	2359808
block4_conv3 (Conv2D)	(None, 7, 7, 512)	2359808
block4_pool (MaxPooling2D)	(None, 3, 3, 512)	0
block5_conv1 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv2 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv3 (Conv2D)	(None, 3, 3, 512)	2359808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0

Total params: 14,714,688

Trainable params: 0

Non-trainable params: 14,714,688

4) Transfer Learning using VGG16 with fine tuning and padding (TL-3)

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
zero_padding2d (ZeroPadding2D)	(None, 32, 32, 3)	0
vgg16 (Functional)	(None, 512)	14714688
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 1000)	513000
dropout (Dropout)	(None, 1000)	0
dense_1 (Dense)	(None, 6)	6006
=====		

Total params: 15,233,694

Trainable params: 7,598,430

Non-trainable params: 7,635,264

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 32, 32, 3)]	0
block1_conv1 (Conv2D)	(None, 32, 32, 64)	1792
block1_conv2 (Conv2D)	(None, 32, 32, 64)	36928
block1_pool (MaxPooling2D)	(None, 16, 16, 64)	0
block2_conv1 (Conv2D)	(None, 16, 16, 128)	73856
block2_conv2 (Conv2D)	(None, 16, 16, 128)	147584
block2_pool (MaxPooling2D)	(None, 8, 8, 128)	0
block3_conv1 (Conv2D)	(None, 8, 8, 256)	295168
block3_conv2 (Conv2D)	(None, 8, 8, 256)	590080
block3_conv3 (Conv2D)	(None, 8, 8, 256)	590080
block3_pool (MaxPooling2D)	(None, 4, 4, 256)	0
block4_conv1 (Conv2D)	(None, 4, 4, 512)	1180160
block4_conv2 (Conv2D)	(None, 4, 4, 512)	2359808
block4_conv3 (Conv2D)	(None, 4, 4, 512)	2359808
block4_pool (MaxPooling2D)	(None, 2, 2, 512)	0
block5_conv1 (Conv2D)	(None, 2, 2, 512)	2359808
block5_conv2 (Conv2D)	(None, 2, 2, 512)	2359808
block5_conv3 (Conv2D)	(None, 2, 2, 512)	2359808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0
global_max_pooling2d (Global	(None, 512)	0
Total params: 14,714,688		
Trainable params: 7,079,424		
Non-trainable params: 7,635,264		