Capstone Project

Schuman Zhang
February 6th, 2018

I. Definition

Project Overview

This capstone project is an analysis of satellite imagery to identify areas of deforestation in the Amazon rainforest. The use of deep learning and artificial neural networks for image recognition problems are well established.[1] This project attempts to identify features in a large set of satellite images of the Amazon rainforest by training convolutional neural networks, the dataset used for this project can be found on the Kaggle website.[2]

Deforestation is a major problem contributing to climate change. A newly launched satellite called 'Planet' can help governments and organizations combat deforestation. From Planet's database of images, we can use computer vision technologies to automatically identify and detect areas of man-made deforestation on a large scale. The accuracy and speed of our solution will enable governments to respond effectively and in a timely manner.

Problem Statement

The problem in this project is to correctly and accurately identify relevant features within Planet's vast database of satellite images of the Amazon basin. This can be classified as a multi-class and multi-label image recognition problem, meaning each image may contain none or up to 17 relevant labels. The relevant labels that we are attempting to detect are essentially features relevant to understanding deforestation, they include detecting areas of primary rainforest, agriculture, bare ground, blooming, blow down, clear weather, partly cloudy, haze, habitation, cultivation, conventional mining, road, selective logging, slash burn, water, and artisinal mining.

Our solution is to train a number of convolutional neural networks (CNNs) on a training image set, then using the trained networks to predict the test image set. We will attempt to build a CNN from scratch, then we will use transfer learning on the ResNet50 and Xception architectures. The solution we will present will include a web application where the user can upload a test

[1] Use of deep learning to analyze satellite imagery - https://arxiv.org/pdf/1704.02965.pdf.
[2] Planet - Understanding the Amazon from space - https://www.kaggle.com/c/planet-understanding-the-amazon-from-space/data

image and get a prediction on the relevant features detected in that image, and also batch processing a bunch of testing images and get the predictions in a .csv file.

Metrics

The metrics that we will use to evaluate our models include the F-2 score (or F-beta score), as well as the recall and precision scores. In a multi-label classification context, these three metrics are particularly important, with recall measuring the 'completeness' or our result and precision measuring the 'relevance' or 'accuracy' of our result. The F-2 score is a balance between recall and precision with greater weight being placed on the recall score. Our formula for F-2 score is presented below.

$$(1+\beta^2)\frac{pr}{\beta^2 p + r}$$

Figure 1: F-2 score, where p is precision, r is recall and beta=2

In a multi-label classification context, measuring accuracy is an interesting challenge since any given result can be a completely correct, partially correct or not correct at all. A better measure is how complete and precise our predictions are. For any given prediction, recall measures how successful the deep learning model is at 'recalling' all the correct labels. While precision looks at the predictions given by the model measures how 'correct' they are. A balance of the two metrics is computed by the F-2 score, in this case recall is more important than precision since we want to make sure that all relevant features are detected in the satellite images, hence a beta value of 2 is assigned.

II. Analysis

Data Exploration

The dataset we are using for this project consists of a set of images in .jpg format. Each image is essentially a 'chip' taken from a 'scene' from Planet's from full-frame analytic scene product. The particular scene we are using for this project covers an area of 24,408 hectares and spans the Amazon basin. Each chip in our dataset measures 256 x 256 pixels and covers 221 hectares.

Kaggle has provided two major sets of images, the first set is a training set consisting of 40,479 images and the second set is a testing set of 40,689 images. The training set is labelled by a combination of satellite image analysts and crowdsourcing efforts, the labels are provided via a

.csv file. The testing set however, is unlabelled. The purpose of the testing set is for competition participants to upload the predicted labels for evaluation.

For the purposes of our project. We will focus on the labelled set of images (training set) provided by Kaggle. This means that we will further divide the labelled set of images into our own training, validation and testing sets. Our testing set will be 20% of the total amount of images while our validation set will be 20% of the training set. In summary we will have 25,906 training images, 6477 validation images and 8096 testing images. The unlabelled set of images from Kaggle will be used for the batch processing part of our solution.

Exploratory Visualization

Since our dataset consists of a collection of 256 x 256 pixel images, it is useful to visualize them along with their labels and intensity distribution histogram. This way we can visualize the meaning behind our target labels. We have used a jupyter notebook to visualize some of the images.

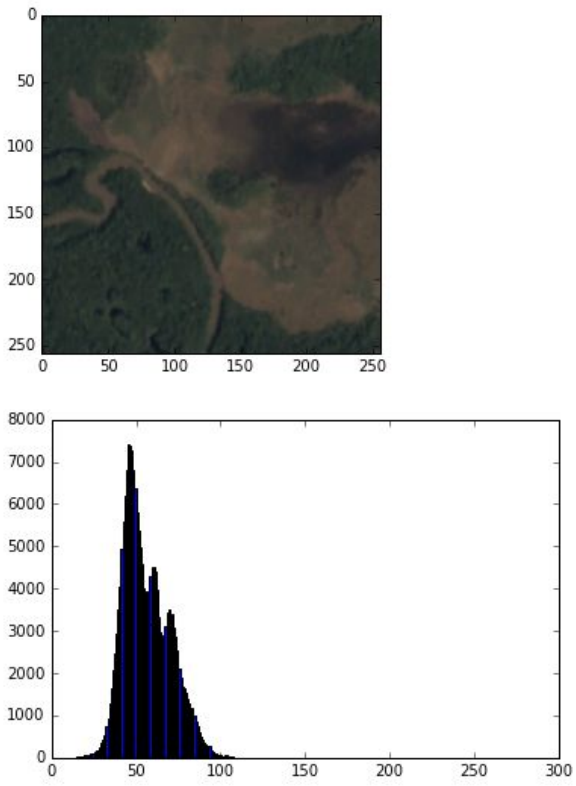

Figure 2: A visualization of the input images along with their labels. The histogram has pixel values on x-axis and corresponding number of pixels on the y-axis. The left region of histogram signifies darker pixels while the right region means brighter pixels

Algorithms and Techniques

There are two major techniques we will use to solve this problem. The first technique is to use a CNN built from scratch. This will consist of several convolutional 2D layers with max-pooling layers in between. The final output layer will use a ‘sigmoid’ activation function since we are solving a multi-label classification problem, this means that in the output vector a probability greater than 0.5 is needed in order to be identified as a positive label. This CNN will be relatively unsophisticated and will only be a handful of layers deep, so we expect a fairly quick training time. We will use 8 epochs and a SGD optimizer, then saving the best trained weights to be used for our evaluation and predictions.

In the process of building a CNN, it is important to note parameters such as number of filters, kernel size, stride and padding. These parameters will determine the shape of the following convolutional layer. As a general rule, filters increase as the CNN gets deeper, since we want to extract or convolve large general features from the image then proceed to extract finer details in the latter layers. Kernel size is the size of the filter moving over the image while stride dictates the number of steps the filter moves across the image. Lastly the padding determines if the image should be padded with zeros in order for the filter to completely cover the image. As we add more convolutional layers the number of parameters will drastically increase, hence we use a downsampling technique with max pooling layers to reduce the number of total trainable parameters.

The second major technique is to use transfer learning on state-of-the-art architectures trained on imagenet. There are a number of readily accessible CNN architectures in Keras. In particular we will use ResNet50 and Xception. The use of transfer learning will likely yield more accurate results since these state-of-the-art architectures were also applied to image recognition problems. For our purposes, we will replace the output layer of both ResNet50 and Xception with our own dense layer with a ‘sigmoid’ activation function.

One interesting concept regarding transfer learning is the similarity and volume of our dataset compared to the dataset of the pre-trained model. The more similar the images and classification categories, the less of the pre-trained model we have to retrain. If the image data is not similar to the pre-trained data, then we can still keep many of the initial layers in the pre-trained CNN since they learn very generic features such as edges and shapes. We can then choose to unfreeze more and more weights in the latter layers in order to tailor the network for our image classification task.

Benchmark

The benchmark model we will use is to randomly generate labels for our test images. Since this is a multi-label problem, for each image in the test set the random label generator needs to generate between 0 to 17 labels. The code block below from the jupyter notebook illustrates our

benchmark model. The result from such a model only achieves a F-2 score of 0.27, we expect our trained models to significantly outperform our benchmark.

```
: import random

def generateRandomLabels():
    num_labels = random.randint(1, 5)
    prediction_list = []
    for i in range(len(test_targets)):
        prediction = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        for j in range(num_labels):
            index = random.randint(0, 16)
            prediction[index] = 1

        prediction_list.append(prediction)

    return prediction_list

random_predictions = np.array(generateRandomLabels())

#print(random_predictions[:5])
random_recall_score = recall_score(test_targets, random_predictions, average='weighted')
random_precision_score = precision_score(test_targets, random_predictions, average='weighted')
random_fbeta_score = fbeta_score(test_targets, random_predictions, average='weighted', beta=2)

print('Random recall score:', random_recall_score)
print('Random precision score:', random_precision_score)
print('Random fbeta score:', random_fbeta_score)
```

```
Random recall score: 0.260122066535
Random precision score: 0.549366891787
Random fbeta score: 0.267310995709
```

Figure 3: The benchmark model, generating labels randomly

III. Methodology

Data Preprocessing

The data preprocessing stage of our project involves splitting the image data into training, validation and testing sets. The labelled data will need to be encoded and converted into a numpy array. Then finally the input images and their respective file paths will also need to be in a numpy array format.

The first step in our preprocessing is to turn the .csv data into a pandas dataframe that we can use. It is important in this step to ensure the ordering of the images match up with the tags column, since the ordering of the image file paths will be scrambled when using the 'load_files' function from sklearn. The image below shows the output of our dataframe after importing the relevant file paths and tags from the .csv file provided by Kaggle.

	filepath	image_name	tags
id			
0	../data/train/train-jpg/train_0.jpg	train_0	haze primary
1	../data/train/train-jpg/train_1.jpg	train_1	agriculture clear primary water
2	../data/train/train-jpg/train_2.jpg	train_2	clear primary
3	../data/train/train-jpg/train_3.jpg	train_3	clear primary
4	../data/train/train-jpg/train_4.jpg	train_4	agriculture clear habitation primary road

Figure 4: Pandas dataframe with the appropriate file paths lined up with their respective labels

The second step is to ensure the 'tags' or our labels are one-hot encoded in numpy array format (eg. [0 0 1 0 1 1 0]). We then assign the one-hot encoded numpy array to a variable called targets. Our training files and targets will then be split into the appropriate training, validation and testing sets.

```
: total_classes = ['agriculture', 'bare_ground', 'blooming', 'blow_down', 'clear',
      'conventional_mine', 'cultivation', 'habitation', 'haze', 'partly_cloudy',
      'primary', 'road', 'selective_logging', 'slash_burn', 'water', 'cloudy', 'artisinal_mine']

  def constructLabelVector(dataFrame):
      result_vector = []
      for index, row in dataFrame.iterrows():
          pos = findPositionsWithinClasses(row['tags'])
          vector = createVector(pos)
          result_vector.append(vector)

      return result_vector

  def findPositionsWithinClasses(string):
      positions = []
      list_strings = string.split()
      for word in list_strings:
          index = total_classes.index(word)
          positions.append(index)

      return positions

  def createVector(pos):
      vector = []
      for i in range(len(total_classes)):
          if i in pos:
              vector.append(1)
          else:
              vector.append(0)

      return vector

  targets = np.array(constructLabelVector(df))

  print('There are %d labels in total.' % len(targets))
```

Figure 5: Code to one-hot encode the labels from the previous pandas dataframe

The final step in our data preprocessing is to turn the train_files, valid_files and test_files into tensors as shown in the image below. Once that is done we will have the appropriate inputs and targets for our CNNs.

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|██████████| 25906/25906 [00:54<00:00, 473.59it/s]
100%|██████████| 6477/6477 [00:12<00:00, 537.33it/s]
100%|██████████| 8096/8096 [00:15<00:00, 536.57it/s]
```

Figure 6: Code to turn file paths into tensors for input into our CNNs

Implementation

In this section we will describe the implementation of our CNN built from scratch then our implementation for using transfer learning using ResNet50 and Xception.

The first CNN that we build will be known as the original model, it will consist of 3 convolutional layers with an increasing number of filters, a kernel size of 3 with relu activation functions. Wedged in between the convolutional layers will be max pooling layers. The output with be a Dense fully connected layer with a sigmoid activation function.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 254, 254, 20)      560
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 127, 127, 20)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 125, 125, 40)      7240
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 62, 62, 40)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 60, 60, 80)        28880
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 30, 30, 80)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 30, 30, 80)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 72000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 17)                1224017
=================================================================
Total params: 1,260,697.0
Trainable params: 1,260,697.0
Non-trainable params: 0.0
_________________________________________________________________
```

Figure 7: Summary of our CNN model built from scratch

We then compile the model with a SGD optimizer and a binary cross entropy loss function. When we train our model we use batch size of 20 and save our best weights in a .hdf5 file. The inputs that we use for training will be training and validation file and their respective targets. The training will be done over 8 epochs.

The transfer learning techniques using ResNet50 and Xception models take advantage of pre-trained weights from imagenet. These two models can be loaded directly from Keras.applications. For both our ResNet50 and Xception transfer learning techniques, we simply load the base model, remove the output layer, then add in a global average pooling layer as well as our own Dense layer with a sigmoid activation function. It is important to freeze the pre-trained layers.

```
def build_ResNet50_shallow():
    x = base_model_Resnet50.output
    x = GlobalAveragePooling2D()(x)
    predictions = Dense(nb_classes, activation='sigmoid')(x)

    model = Model(base_model_Resnet50.input, predictions)

    #freeze pretrained base layers
    for layer in base_model_Resnet50.layers:
        layer.trainable = False

    #print(model.summary())
    return model
```

Figure 8: Code for building our transfer learning model using ResNet50.

When we compile these two models we will use a slower learning rate (1e ^ -4), but we will keep the SGD optimizer and binary cross entropy loss function and we also need to train these models for a higher number of epochs. The best weights will also be saved in a .hdf5 file after training.

One interesting challenge while building these CNNs is the initialization of parameters such as filters, kernel size, number of epochs and batch size. The challenge is to find a combination of these parameters that will yield useful results but also keeping trainable parameters at a reasonable level to ensure manageable training times. The parameters used in this implementation achieves this goal. While they may not be the most optimal, but they certainly yield useful predictions within a reasonable level of training time. Some parameters such as batch size, epochs and learning rate were chosen after training the CNNs for a handful of epochs in order to estimate total training time and rate of improvement in training/validation loss.

Refinement

There are two methods of refining our implementation from above. The first is to fine tune the parameters in our original CNN. The second is to fine tune weights at different depths for the transfer learning models.

Fine tuning parameters in a CNN is not an exact science but depends upon the type and complexity of the images. The parameters chosen with kernel size of 3, padding of 'valid', stride of 1, learning rate of 0.01 and a doubling of filters as convolutional layers increase yielded useful predictions. However, I did not extensively perform refinement on these parameters due to limitations in cost and time. Also I believed transfer learning models will yield even better results, so I did not spend too much time optimizing the original CNN model.

In our transfer learning models, refinement can be done by either just retraining the top layer if the data and image recognition task is very similar to imagenet, or we can unfreeze weights from previous layers if the task is substantially different. For our purposes, the satellite images are rather different from imagenet in that they are not as complex, as such keeping the lower layers of pre-trained models should yield desirable results. Retraining higher layers will tailor the network to our specific classification problem.

```
def build_ResNet50_deep():
    x = base_model_Resnet50.output
    x = GlobalAveragePooling2D()(x)
    predictions = Dense(nb_classes, activation='sigmoid')(x)

    model = Model(base_model_Resnet50.input, predictions)

    for layer in model.layers[:26]:
        layer.trainable = False
    for layer in model.layers[26:]:
        layer.trainable = True

    #print(model.summary())
    return model
```

Figure 9: Unfreezing lower layers in our ResNet50 network in order to refine our model

While our previous model yielded some promising results, refining the transfer learning models could increase the performance of our evaluation metrics. For our ResNet 50 model, I chose to retrain 26 layers while for our Xception mode, I chose to retrain 19 layers.

IV. Results

Model Evaluation and Validation

In our implementation we trained a total of four models - an original CNN model built from scratch, a ResNet50 model with a retrained top layer, another ResNet50 model with half the layers retrained and finally a Xception model with 19 layers retrained. Their respective predictions on the test set are used to calculate the precision, recall and F-2 score, which are summarized in the table below.

Model	Recall	Precision	F-2 Score
Original Model	0.72	0.79	0.73
ResNet50 Shallow	0.79	0.85	0.79
ResNet50 Deep	0.83	0.89	0.84
Xception	0.68	0.82	0.68

Figure 10: The table above summarizes the metrics obtained from the trained models

The best trained model is the ResNet50 Deep model (optimal model), this is the transfer learning model where half the layers are retrained with our own training data. The training time for 25 epochs was close to 8 hours. Overall, this model is robust with a F-2 score of 0.84. The weights from this model was saved in a separate .hdf5 file which was used as part of our solution in a web application and batch processing to a .csv file.

In proving the robustness of the optimal model, I tested the model using unseen satellite images from Planet's test directory. The predictions produced were of high quality even when I flipped and translated the test images. This suggests the model generalizes well to unseen data. However, when I uploaded totally irrelevant images, the model produced some strange results by detecting labels when they clearly are not present. In summary, the model can be trusted if we test it with relevant Planet satellite imagery specific to our domain, but it cannot be trusted to process 'outliers' our irrelevant images. One way to prevent this from happening is to ensure that satellite imagery is detected before applying our specific classification task.

Justification

The optimal model vastly outperformed the benchmark model, while our benchmark only had a F-2 score of 0.27, our optimal model achieved 0.84. The performance metrics for the optimal

model may be improved further. However, in light of the computing resources and time available, the optimal model has achieved sufficient robustness and significance for us to build a complete solution based on this image recognition task.

The optimal model manages to recall 83% of labels while labels predicted are close to 90% accuracy. As a rough estimate, this level of performance may be comparable to an crowdsourced human analyst. At this level of robustness, it is possible to build a software application which automatically processes image chips from Planet satellite, with a human-in-the-loop to further validate and improve the model. Such a software application may alert governments and organizations to respond to deforestation events without the need for humans to slowly and painstakingly analyze a huge repository of images in a daily basis.

Finally, it is important to visualize the training history of the optimal model for further justification. We plotted training and validation accuracy and loss in the graphs below. We can clearly see that over the 25 epochs both training and validation loss are decreasing while accuracy is increasing. The lack of divergence between training and validation suggests that we are not at a point where the model is overfitting. However, improvements in both accuracy and loss slowing down as we increase the number of training epochs.

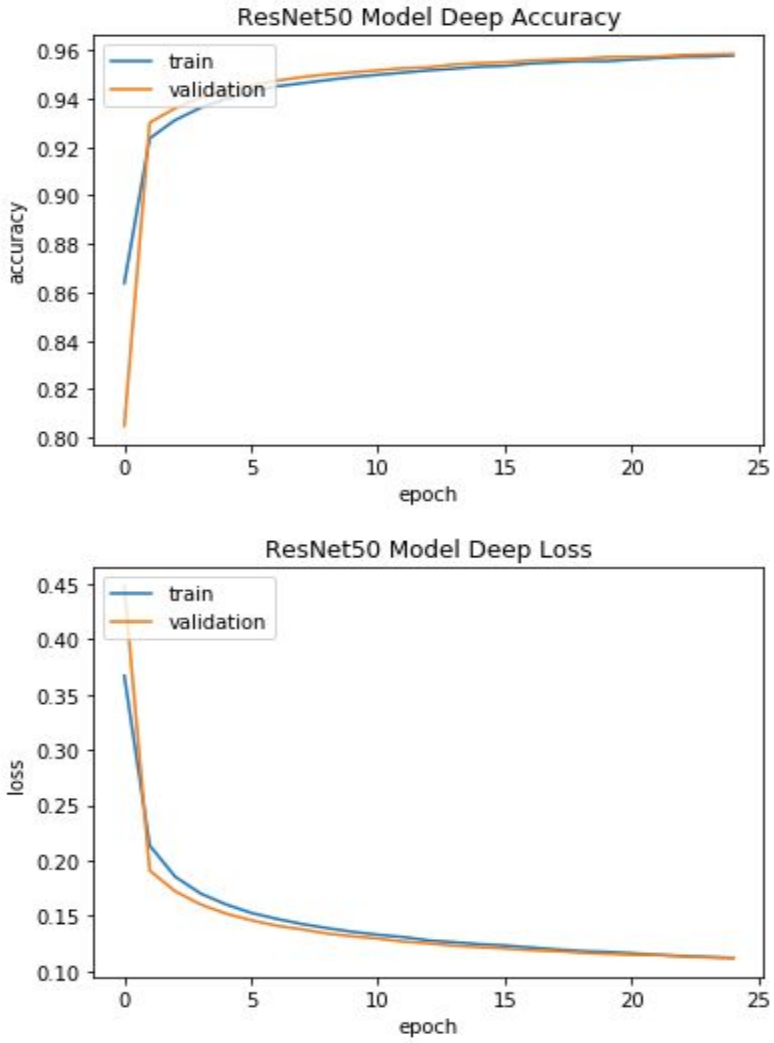

Figure 11: Graphs showing training and validation accuracy and loss for the optimal model

V. Conclusion

Free-Form Visualization

There are two ways of visualizing the solution obtained from the optimal model. The first is via a web application where the user can upload a satellite image. The user can also choose to use other trained models and view their respective metrics and predictions. Refer to the readme.md to run the web application.

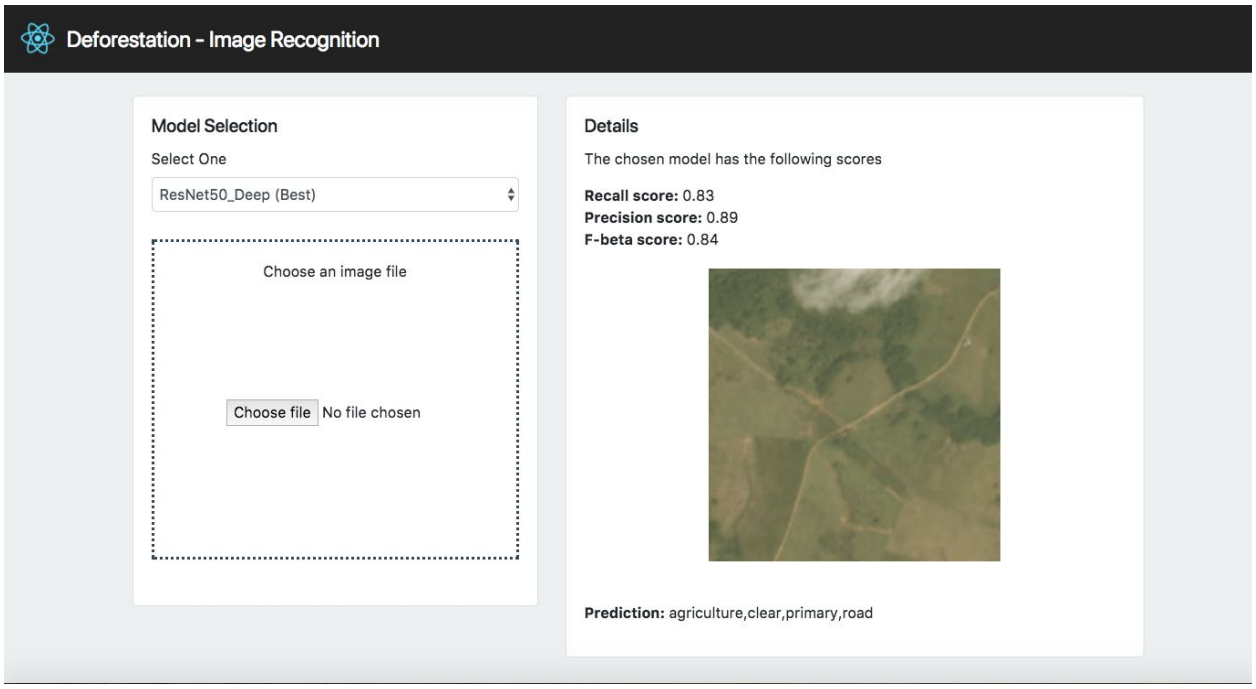

Figure 12: A web application written with Django and React.js to showcase the trained models

The second way the result can be visualized is via batch processing the unlabelled test images provided by Kaggle. The code for this is provided in the final code blocks of the jupyter notebook and the output is in a file named batch_processed_labels.csv.

Reflection

In this project we used deep learning techniques to detect relevant features in Planet's satellite images. Specifically four separate convolutional neural networks were used. The first one was built from scratch and three others were built using transfer learning techniques based on ResNet50 and Xception. The model training was completed on an AWS p2.xlarge instance with a GPU. In addition to training the models, a web application was built to showcase the metric and the predictions generated by each model.

The interesting/surprising aspect of this project is the relatively high performance of the model built from scratch. While the Xception model trained using transfer learning actually under-performed when compared to the relatively unsophisticated original model. This suggests that the complicated architecture in Xception is perhaps not extracting the relevant features for our task, as shown by a very low recall score compared to the other models. This also suggests that the features in our learning task are not particular complicated and a relatively high level of performance can be achieved with simple CNNs. While our transfer learning models are trained on imagenet to distinguish between 1000 everyday object categories, the complexity of these

models could be somewhat redundant for our task. It would be very interesting to see how far we can optimize an unsophisticated CNN to achieve better metrics than our optimal model.

The difficult aspect of this project is the computation resources and long training time of CNNs. The optimal model and the Xception model both took more than 8 hours to train on a GPU at $1.50 per hour. The lack of computational resources is a limitation which prevented further optimization and refinement, especially when it comes to experimenting with input parameters and other CNN architectures.

In light of the computational resources and time limitations, the results obtained by the optimal model exceeded my expectation for the solution to our task. I believe this model is already robust enough to build a software application which can achieve the stated goal of combating deforestation.

Improvement

After careful consideration, there are 3 major ways to improve the metrics obtained by our trained CNNs:

- Given more computational resources and time, we can experiment and refine our input parameters further. Particularly when it comes to using regularization, and maybe using a different optimizer, tweaking the number of training epochs as well as the learning rate. A model with even better metrics could be feasibly achieved

- Again given sufficient computational resources, we can further experiment with various CNN architectures. A higher F-2 score may be achieved by adding convolutional layers and pooling layers to our CNN model built from scratch. It would also be interesting to see the effects of a different filter size on these images

- Image augmentation can be done on our training set. This involves translating and flipping the existing images so our algorithms can be exposed to a wide range of images with different orientation of the features. This step could improve the metrics obtained by each of our models