Project - Final Paper

Project 1.7: Spatio-temporal Analysis of Satellite Imagery

Shaheer Akhtar

Introduction

Satellite imagery has the potential to transform urban development and planning. The onset of deep learning in recent years makes this a possibility more realistic than ever before. Analysing change in land use by doing spatio-temporal analysis of satellite imagery promises great possibilities in automating the process of urban analysis.

In this paper, we design and tune a Vision Transformers (ViT) in a Siamese network whilst utilizing the C2D2 dataset to understand and classify four different kinds of transitions in land use: construction, deconstruction, cultivation, and decultivation.

Note: Previously, a CNN was trained on the Asia-14 dataset to recognize 14 classes. Images taken from Google Maps from separate years were split into frames. Each frame was classified, and the classifications of the same frame from the different years were compared. This comparison showed the transition over the years. The model, however, did not perform great, and was not included in this paper as it takes a completely different approach to the problem (complete details of the model, its performance, and visualizations are in Project Deliverable 2).

Initial Model Architecture

For the model, a Siamese network architecture was used that incorporates Vision Transformers (ViT) to learn and predict transitions between images (which will be the satellite images over time).

We use the C2D2 dataset which consists of image stacks that contain 3 images each of a particular location, from 2011, 2013, and 2017. Each stack has 1 label, from construction, deconstruction, cultivation, decultivation. The label shows how the location changed overall through the three years. For our model, we initially used just the first and third image from each stack as the ‘before’ and ‘after’ image of a location, along with the stack’s label which showed the transition class (which acted as the label).

The architecture of the Siamese network consists of two identical subnetworks, each of which consists of a Vision Transformer base. This vision transformer base is vit_b16. The ‘b16’ configuration means the input images are split into patches of 16x16, which is beneficial for our model as it allows the model to capture subtle changes within the satellite images. It is pretrained on ImageNet, as this allows our model to learn a wide variety of visual features (such as textures and patterns) that can be useful in understanding our images, even with not-so-big quantities of data.

One of the two Siamese subnetworks is fed the ‘before’ image, while the other is fed the ‘after’ image. They are both processed independently, and individual embeddings are made for each. These are then concatenated, before they are passed through a series of dense layers. The last layer is a classification layer that predicts the output class using a softmax from the 4 possible classes.

Alternative attempted approach:

Earlier, we tried a different approach too, which was to make embeddings for the before and after images, place these embeddings in a vector space, calculate the Euclidean distance between them. We then let this direction vector between the two embeddings act as our point of information for the transition class. The hope was that when we give it two new images and place their embeddings in the vector space, the vector between them would indicate the transition type. However, this did not work out effectively, most likely due to the size of our data being small relative to its complexity. The approach would likely have had a better chance to work if the dataset was larger or less complex, which would have made it easier for the model to store the most useful information in some key dimensions in its embeddings, which could then primarily be used as the points of information.

Training and Tuning:

The model was trained over 50 samples, 200 samples, and then the entire dataset of 836 images of Lahore for 10 epochs. A test set was made of 20% of the data, and a validation set of 10%. The following were some key metrics of the training, validation, and test sets after training over the entire dataset (please note that more metrics such as precision and recall were also calculated and saved throughout the model tuning, but only loss and accuracy are mentioned in this document, for the sake of simplicity and to illustrate the direction the model improvements are going in):

	Training set	Validation set	Test set
Loss	0.0018	1.0605	1.4896
Accuracy	1.0000	0.6875	0.6499

With the model having perfect accuracy over the training set but performing not-so-great over the validation and test sets, it was clear it was overfitting. Several approaches were explored to fix this.

Increasing the 'pairs' of images

The first attempt to fix overfitting was made by attempting to increase the size of the dataset. Instead of using just the first and third image from each image stack, we decided to use the following pairs of images as the 'before' and 'after', respectively:

- First and second
- Second and third
- First and third

Each of these were considered a pair, and each had the same label as the stack originally did.

However, when these were initially used, the model got an accuracy of 1.0 over the training, validation, and test set. This happened to be because there were three pairs being made from each stack, but across these three pairs, there were the same images. So if, for example, 2 of these pairs had been a part of the training set, the model learnt about it. And then if the third image was part of the test set, the model predicted it successfully. In other words, the model had already seen the validation and test data.

To deal with this, instead of making these pairs first and then splitting the data, we simply first split the stacks into training, validation, and test stacks, and then split the stacks into pairs. This ensured that all three pairs from one stack stayed in the same data split, and the validation and test splits had entirely unseen data.

Having done this, the amount of data was now technically thrice in amount. Although there was redundancy in this, there still was new information (especially because of the second image of each stack) that could be learnt.

We ran the model again on the entire dataset, and these were now the results:

	Training set	Validation set	Test set
Loss	0.00004	1.8026	1.7764
Accuracy	1.0000	0.6865	0.7063

The test accuracy increased by about 4%, which was a positive sign. However, the training set still had perfect accuracy, which was a lot more than the validation and test set. The model was therefore still overfitting.

Hyperparameter tuning, batch normalization, and early stopping

In an attempt to further reduce the overfitting, we tried batch normalization as well. However, the loss and accuracy of the validation set fell significantly, and we decided to certainly not use it. Then we tried hyperparameter tuning using Keras Tuner. We primarily focused upon:

- Dropout rate: we introduced a dropout rate and varied its values between 0, 0.1, 0.25, and 0.5. This was in an attempt to help the model generalize better, by reducing dependency on specific feature paths.
- We also changed the number of neurons in the fully connected classification layer, changing them between 64, 128, and 256. This helped vary the model's complexity.

This hyperparameter tuning was very computationally intensive and was attempted over Google Colab over a paid GPU. Unfortunately, over an hour into the training, the program crashed, when running trial 15 out of 18. Still, the best values so far had been a dropout rate of 0.1 and number of neurons in the classification layer as 128. These numbers made sense and the training had almost been complete anyways. Thus, we decided to go ahead with these for now.

After the grid search, the following is how the heart of our model looked like (on top of vit_b16 on ImageNet):

```
combined_features = concatenate([processed_a, processed_b], axis=-1)
combined_features = Dropout(0.1)(combined_features)
classification_layer = Dense(128, activation='relu')(combined_features)
outputs = Dense(4, activation='softmax')(classification_layer)
```

To further prevent overfitting, we also introduced early stopping, to stop the model from learning if the validation loss started increasing again. This was done with a patience value of 5.

With this, training stopped after 6 epochs (out of 10) due to early stopping, and gave us the following values:

	Training set	Validation set	Test set
Loss	0.0693	1.4384	0.7682
Accuracy	0.9760	0.7063	0.6687

The loss for validation and test are both lower, while the accuracy over the validation set is higher, which are positive signs. The performance over the training set is also slightly less perfect. However, the test accuracy is lower than before. These changes might just have been due to the early stopping feature, and it is hard to say if the model actually improved due to the hyperparameter tuning.

To be sure, we tried running the model again without the early stopping, and had the following results:

	Training set	Validation set	Test set
Loss	0.0581	1.7348	1.4785
Accuracy	0.9834	0.6429	0.7123

As suspected, the previously seen changes had reversed. Excluding any potential affects of early dropout, the hyperparameter tuning gave us an improvement of roughly 1% in the test accuracy.

Using all 3 cities' data

Lastly, we decided to use the entire dataset of 1996 stacks of the three cities instead of just using the 836 stacks of Lahore.

We also decreased the learning rate from 0.0001 to 0.00001 to further prevent overfitting, and because we now had more data over which to generalize.

The 1996 stacks we were using resulted in 5988 pairs of 'before' and 'after' images (meaning 11,976 raw images). Unfortunately, Google Colab stopped working with that much data, likely due to system RAM limitations. We therefore attempted to use one pair of 'before' and 'after' images using the first and third image in each stack, similar to how we had done initially, but now with a much greater amount of raw data and hopefully improved parameters.

After training the model on the entire dataset, these were the results:

With early stopping:

	Training set	Validation set	Test set
Loss	0.1667	0.8219	0.7270
Accuracy	0.9620	0.7100	0.6800

Without early stopping:

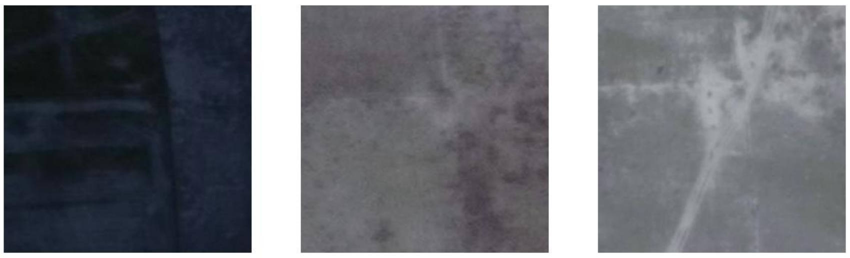
	Training set	Validation set	Test set
Loss	0.0943	0.8705	0.8183
Accuracy	0.9850	0.7050	0.7225

A test accuracy of 72% and a validation accuracy of 70.5% is the highest we have seen so far, and we have improved by about 7% in test accuracy since we started. Considering that we are doing multi-class classification with 4 classes, a randomly guessing model would have an accuracy of 25% (1 in 4 correct guesses). While this model's performance is still not ideal, it is clear that the model is learning, and the tuning is working.

Concerns with the dataset

But despite a great deal of tuning, our model's performance was not where we hoped it would be at. It is very important to take a moment to understand what might have been the biggest limiting factor from the get-go: the images in the dataset. Many of the images in the dataset are distorted and discoloured. Some images in the same stack are out of position (as in, they do not capture the exact same area, but are a bit off to the side). Many stacks are very hard for even a human to classify. Due to this, it was always going to be difficult for any model to successfully learn the transitions. The following page has some examples of all three images of some stacks placed together to illustrate the problems.

Transition Class: 0.0

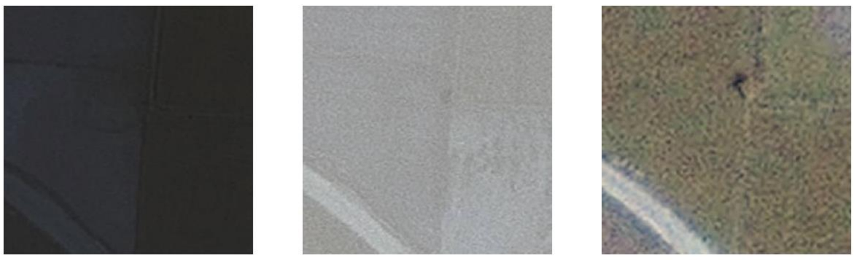


Transition Class: 0.0

Transition Class: 0.0

Transition Class: 0.0

Transition Class: 2.0



Future Improvements

While problems with the current model and data have been discussed throughout the paper, there are still future improvements that can be made. Some of these include:

- Using triplet loss
    - Triplet loss requires three images for every data point:
        - Anchor: This is the baseline image (such as a normal satellite image)
        - Positive: This image shows the positive change we are trying to detect (such as a new building having been constructed in the anchor image)
        - Negative: Some image that does NOT show the positive change, so the model learns what not to look for.
    - Using triplet loss can help our model understand better what features are important. However, currently when we used 'before' and 'after' images, these could very well serve as our anchor and positive respectively. But, we would need to manually identify negative images to feed into the model. The negative images will still be from the dataset, and there are techniques to semi-automate it.
- Improving the quality of the dataset
    - We can work on preprocessing the images, or finding clearer satellite images
    - Multispectral and hyperspectral images can also be included, such as thermal infrared or UV, as they can capture additional information that could be relevant.
- Changing the transition classes we are seeking to identify
    - Classifying cultivation and decultivation in particular can be very difficult due to colours. Barren land would look brown, and wheat grown on it would also look brown. It would be difficult to identify cultivation here, especially from low-resolution satellite imagery.