Terrain Classification with Deep Learning

Chao Fang How, Gabriel

Abstract

Terrain classification, specifically semantic segmentation of aerial images into classes such as roads, buildings, trees etc. is a computer vision problem particularly suited to a deep learning approach. This report describes the application of convolutional neural networks to this problem, focussing on two architectural aspects: the feature extractor and the decoder. Three models were implemented and trained on the ISPRS Potsdam dataset of RGB images: a ResNet-based FCN, a ResNet-based PSPNet, and a MobileNetV2-based FCN. Of these models, the MobileNetV2-based FCN has shown to be the fastest in both training and inference, allowing for more training iterations on limited computational resources and faster inference for time-sensitive applications.

1 Introduction

Terrain classification is a pertinent problem, especially in the context of guided Autonomous Vehicle (AV) navigation and route planning. For an AV to efficiently navigate from one point to another, it has to avoid obstacles such as dense forest or buildings along its path. Aerial imagery can provide advance knowledge of obstacle locations hundreds of metres away from the AV and thus greatly aid in macro-scale route planning, reducing travel time and keeping on-the-spot exploration and backtracking to a minimum.

However, manually identifying obstacles for AV route planning can be an arduous task, especially when controlling multiple AVs spread across a large area. A fully autonomous route planning approach, where terrain is automatically classified and the route planned accordingly, would greatly improve the efficiency of mass AV operations.

This report shows how convolutional neural networks (CNN) can be applied to perform automated terrain classification of aerial orthophotos using only RGB colour data, producing detailed semantic maps of equal resolution to the input images. An illustration of the desired product is provided in Figure 1.

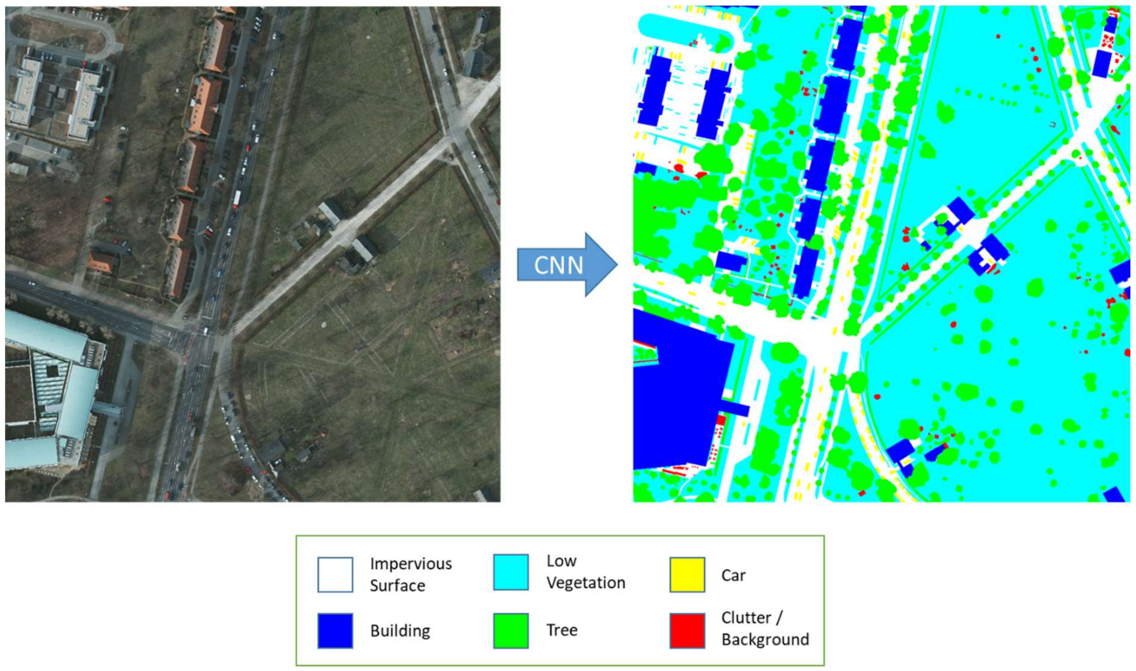


Figure 1: On the left is an RGB input image. The objective is to develop a CNN model to produce a terrain classification label map as shown on the right.

2 Literature Review

Machine learning has long been a mainstay approach to terrain classification. Neural network [1] and random forest/gradient boosted trees [2] techniques have been applied to RGB 3D point cloud data. Neural networks have also been used with colour and infrared (IR)-band orthophotos with a high degree of success [3]. Of interest are CNNs also trained on the ISPRS Potsdam dataset [4] [5], however these works also use the provided IR-band data in addition to colour, and in some cases the Digital Surface Maps (DSMs) as well.

Some works have also detailed full AV-command and control-UAV system implementations with hand-crafted classifiers [6] however those systems are outside the scope of this report.

3 Preliminaries

3.1 Deep Neural Networks

A neural network is a machine learning model that consists of nodes called *neurons*. Each neuron contains numbers called *parameters* which encode the knowledge learnt by the model. These parameters decide the output of the model given an input. Through a training process called *supervised learning*, the model is provided a set of inputs (a.k.a. *features*) and corresponding ground truth outputs (a.k.a. *labels* or *targets*), and the network's parameters are adjusted until the predicted outputs are as close as possible to the provided ground truth (a.k.a. *minimising loss*). A well-trained neural network should be able to generalise the knowledge learnt through training to never-seen-before input data, producing output predictions of good accuracy. Both accuracy and generality must be accomplished by designing the training process and model architecture such that the model does not *overfit* to the training data.

The core concept behind neural networks is that non-linear functions can be learnt by stacking *layers* of neurons. In a vanilla neural network, all the neurons in one layer are *fully connected* to all the other neurons in the previous layer. These layers are normally called *hidden layers*, where a neuron has a parameter for each connection with a previous neuron. This is illustrated in Figure 2.

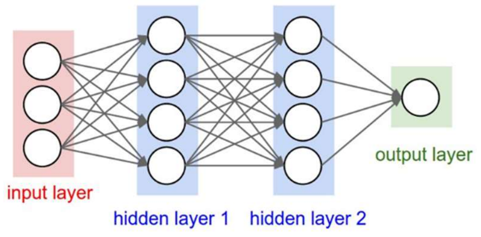


Figure 2: A neural network with two hidden layers [7].

The *depth* of a neural network refers to the number of layers it has. The more layers a neural network has, the *deeper* it is said to be. Generally speaking, each layer of neurons learns progressively higher level, more complex features than the previous layers. This ability of Deep Neural Networks (DNNs) is advantageous for computer vision applications, where visual recognition often involves learning complex combinations of shapes and colours.

Thanks to advances in computing hardware, such as Graphics Processing Units (GPUs) and Application-Specific Integrated Circuit (ASIC) artificial intelligence accelerators such as Tensor Processing Units (TPUs), deeper and more complex DNNs have been made feasible for practical applications.

3.2 Convolutional Neural Networks

Although vanilla DNNs provide a powerful framework for learning complex functions, there are several characteristics that make them inefficient for most computer vision applications. Firstly, the fully connected nature of the hidden layers causes the number of parameters to grow exponentially with each added layer. As modern computer vision neural networks often require dozens or even hundreds of layers, the number of parameters could quickly number in the trillions. Secondly, a naively designed DNN would be forced to learn how to recognise features in each part of the image, even if it has learnt to recognise that feature before in other locations. For example, if a DNN had learnt to recognise a horizontal line at the top-right corner of an image, it would be unable to reuse that knowledge to recognise a horizontal line at the bottom of the image, or even a horizontal line shifted a few pixels left or right. To apply neural networks to visual recognition, these deficiencies must be addressed.

The convolutional neural network was introduced to rectify these shortcomings. The intuition behind CNNs is that the parameters for recognising a feature in one part of an image are useful for recognising the feature in other parts as well. As such, the bulk of the fully connected layers in a neural network are replaced with *convolutional* layers. Each convolutional layer learns a set of *filters*[1]. Each filter can be thought of as a set of parameters for recognising one kind of feature or combination of features. In a convolutional layer, these filters are *convolved* to detect features across the entire image. This greatly reduces the number of parameters needed to accurately perform visual recognition, and makes neural networks practical for large-scale computer vision. Some examples of filters are shown in Figure 3.

[1] Filters are also often referred to as *kernels*.

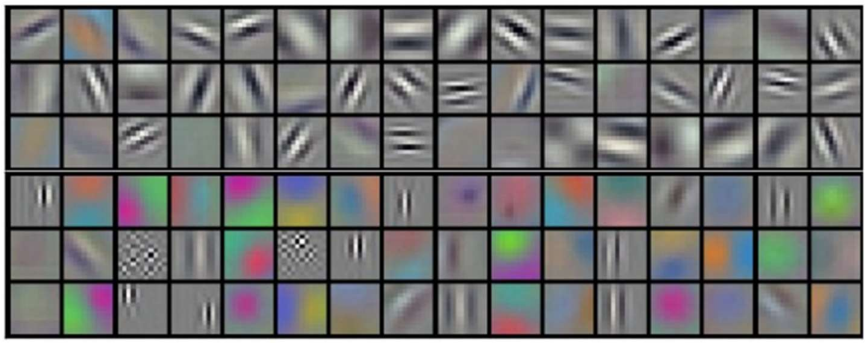


Figure 3: Example filters learnt by Krizhevsky et al [8].

3.2.1 Feature Extractor

The part of the CNN that extracts and recognises features from input images is called the *feature extractor*, and usually consists of multiple convolutional layers of various combinations of *hyperparameters*[2]. Through a *softmax* layer after the feature extractor, it is possible to receive as output a vector of probabilities that describes how likely the image matches the set of predefined label classes the model was trained to recognise, such as car, person, tree etc.

Much research has been performed on designing feature extractor architectures that improve a CNN's ability to understand real-world imagery. A sampling of milestone architectures are AlexNet [8], VGGNet [9], Microsoft's ResNet [10], and Google's Inception [11]. Most feature extractor designs are focussed on delivering the best accuracy, although some, such as MobileNet, aim to provide the best accuracy-performance trade-off for applications with fewer available computing resources [12].

3.2.2 Decoder

Regular CNNs with only a feature extractor can classify the entirety of each input image as one class. However, this is not adequate for our application, which requires not just *image classification*, but *semantic segmentation*. That is, we require each pixel of the image to be labelled with its appropriate class, not just the entire image. This is because there may be multiple objects and stuff and classes of objects and stuff in an aerial image, and the boundaries and contours between them may be highly irregular. The output would thus have to be a map of equal resolution and detail to the input image.

To achieve this, another part, called a *decoder*, is added to the CNN. The role of the decoder is to decode the feature maps extracted by the feature extractor back into a high resolution output image of class labels. Like the feature extractor, the design of the decoder is an architectural aspect that is being continually improved.

The pioneering work that enabled CNNs to efficiently perform semantic segmentation is the Fully Convolutional Network (FCN) [13] which makes use of a decoder that takes feature maps from different layers of the feature extractor and upsamples them before merging them together. This captures both finer grained but simple details from the earlier layers of the feature extractor and coarser but more complex activations from the later layers. This idea is illustrated in Figure 4.

[2] Hyperparameters are settings that describe the configuration of a model's architecture, as opposed to parameters which hold the knowledge learnt by the model.

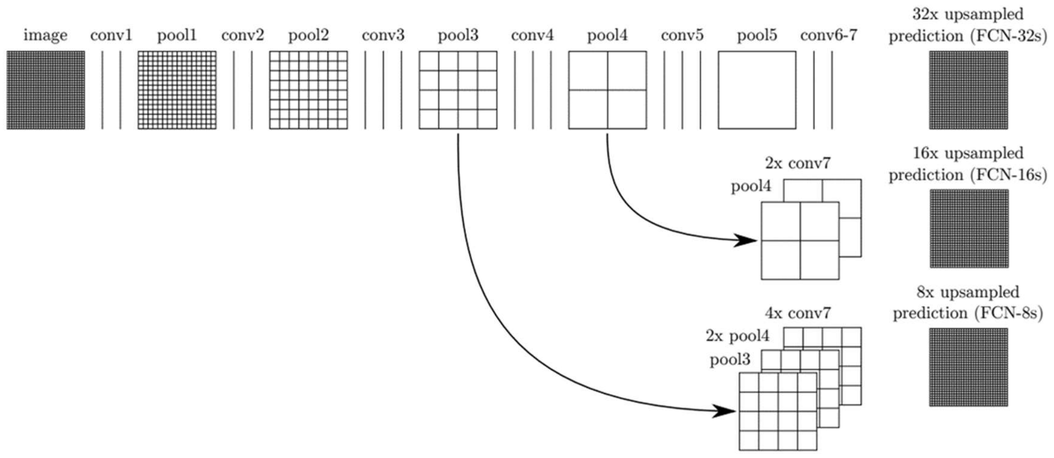


image
conv1
pool1
conv2
pool2
conv3
pool3
conv4
pool4
conv5
pool5
conv6-7
32x upsampled prediction (FCN-32s)
2x conv7
pool4
16x upsampled prediction (FCN-16s)
4x conv7
2x pool4
pool3
8x upsampled prediction (FCN-8s)

Figure 4: FCN skip architecture. The convolutional layers on the top row represent the feature extractor. The FCN decoder uses 'skip' connections to take the outputs at different layers of the feature extractor, and perform some pooling and convolution before upsampling and combining the extracted feature maps. [13]

4 IMPLEMENTATION

4.1 MODELS

Table 1: Summary of implemented models

	Fully Convolutional Network (FCN)	**Pyramid Scene Parsing Network (PSPNet)**	**MobileNetV2**
Year of authorship	2015	2016	2018
Key insight	Produce detailed output images of same dimensions as input via a skip architecture	Exploit global context information through pyramid pooling	Improve speed with linear bottlenecks between layers with shortcut connections
Feature extractor	Resnet50	Resnet50	MobileNetV2
Decoder	3-Skip	Pyramid Pooling Module (PPM)	3-Skip
Number of parameters	23,609,234	46,767,302	1,866,770
Size of model	270MB	535MB	22MB

Three different models were referenced and implemented for this project. The goal is to explore how the characteristics of the different architectures might make them more or less suitable for the task. The three models, the Fully Convolutional Network (FCN) [13], Pyramid Scene Parsing Network (PSPNet) [14], and MobileNetV2 [15] were chosen as they represent important contributions to the semantic segmentation literature.

All models were implemented in Keras using the TensorFlow backend. Keras is a high-level API that greatly simplifies the prototyping of DNN models. It also provides support for GPU computation on NVIDIA GPUs through tensorflow-gpu, although this functionality has not been used due to the lack of a suitable GPU.

Following is a brief overview of the implementation of each model. Note that these are prototype implementations drawn from publicly available open-source code. As such they may not fully represent the most optimised versions of each design. For full details of each design please refer to the referenced papers.

4.1.1 Fully Convolutional Network

As opposed to the original FCN published in 2015, this FCN uses ResNet as a feature extractor, specifically the Resnet50 implementation provided as part of the `keras.applications` library. However, a small modification is made to the `ResNet50` code to provide custom names for the last activation layer of the 3rd, 4th, and 5th stages so as to allow for the easy creation of the FCN skip connections through the `.get_layer()` function.

The FCN '3-Skip' decoder is then implemented using the process described by Azavea [16] in developing their 'Raster Vision' application. Firstly, the outputs of the `act3d`, `act4f`, and `act5c` layers are extracted and are each fed into separate Conv2D layers with kernel size (1, 1) to reduce the number of feature maps to the number of classification labels. Then, they are each upsampled using bilinear interpolation through a `Lambda` layer before being merged by an Add layer. Finally, a softmax layer is applied to obtain probabilities. The output of the model is a 3D `NumPy` array of dimensions (`nb_rows`, `nb_cols`, `nb_labels`) that provides a probability vector for each pixel in the input image that describes how likely the pixel is of each terrain class.

The FCN was built with an input size of (224, 224).

4.1.2 Pyramid Scene Parsing Network

The PSPNet as described by the authors consists of two main insights: the Pyramid Pooling Module (PPM) and a system for training with auxiliary loss for optimising the learning process. Notably, the implementation presented here does not include the auxiliary loss functionality. The PSPNet authors showed that a well-tuned auxiliary loss can grant an improvement of 0.94% percentage points of accuracy [14].

Similarly to the original PSPNet, this implementation uses ResNet50 as a feature extractor. The code used does not utilise the `keras.applications` implementation, and instead defines the ResNet from scratch [17]. Modifying the code to use the optimised library implementation may be one possible avenue for improving performance.

Instead of using skip connections from earlier layers like the FCN, the PPM decoder works solely on the last feature map of the feature extractor. Using convolutions on various pooling sizes, it aims to aggregate features of varying scales so as to exploit global context information in the image. However, the PPM adds a considerable number of parameters to the model, doubling its size compared to the FCN.

The referenced code is not yet truly fully convolutional as the PPM hyperparameters were hard coded to only accept input sizes of (473, 473) and (713, 713). The PSPNet was thus built to accept an input size of (473, 473), which is more than twice the size of the other two models. This may affect performance comparisons. However, the code was modified to also include PPM hyperparameters for input size (224, 224), so a PSPNet for this input size can also be built for further study.

4.1.3 MobileNetV2

The MobileNetV2 is a novel feature extractor that uses modern ideas such as depthwise separable convolutions, inverted residuals, and linear bottlenecks to reduce the number of parameters and increase performance with a tolerable decrease in accuracy. As seen in Table 1, this causes it to be significantly smaller than both FCN and PSPNet.

Similarly to the other two architectures, this implementation defines MobileNetV2 using open-source code [18].

The MobileNetV2 feature extractor in this implementation has been coupled with the FCN '3-Skip' decoder. The decoder is structured the same way as in the FCN, extracting the final 28x28, 14x14, and 7x7 layers and merging them.

The MobileNetV2 was built with an input size of (224, 224).

4.2 Prediction Aggregation

As mentioned before, the models accept inputs of rather small image sizes such as (224, 224) or (473, 473). However, predictions may have to be made on aerial images of much larger sizes. One way of achieving this is to split the entire input image into smaller patches of suitable sizes, use the model to predict each patch, and then combine the patch predictions back into a large image. To improve accuracy, some overlap can be included between each patch. This is referred to as a *sliding window* approach to prediction. The input size is yet another possible hyperparameter for tuning.

This implementation achieves this through a `ProbabilityTile` class that aggregates the probabilities of each predicted patch. To get the class of highest probability for each pixel, a simple `argmax` function is used. If the raw probabilities for each class are desired, the class first divides the summed probabilities by the number of predictions to obtain an averaged prediction probability.

4.3 Dataset

Training data is of crucial importance when creating machine learning models. For this task, a dataset is required that has high-resolution RGB aerial images and corresponding high-quality ground truth label maps. The ground truth in particular would take much work to create as it would involve manually annotating maps on the pixel level by hand.

Fortunately, the International Society for Photogrammetry and Remote Sensing (ISPRS) has released such a dataset to the public for the ISPRS Semantic Labeling Contest. The dataset consists of 38 6000x6000 true orthophotos[3] (TOPs) of the city of Potsdam, Germany at 5cm resolution, with 24 of them accompanied by corresponding ground truth. The colour bands provided include RGB and infrared (IR) data, as well as Digital Surface Models (DSMs) which describe the elevation of the image areas. The ground truth segments the imagery into six terrain classes: impervious surfaces, buildings, low vegetation, trees, cars, and clutter/background. Each pixel is labelled as one and only one terrain class. For an example of an RGB TOP-ground truth pair, refer to Figure 1.

As this project is focussed on classifying imagery taken with readily available consumer drones, only the RGB bands together with the ground truth were used to train the model.

[3] A true orthophoto is a top-down aerial image rectified so that all points in the image appear to be oriented at a right angle to the viewer. This is usually accomplished by combining multiple photographs of the subject area taken at different angles and correcting for perspective distortions.

4.3.1 One-hot Encoding/Decoding

The ground truth labels are provided as TIFF images where classifications are segmented by colour. However, for learning, the model would require a binary vector that describes which class a pixel belongs to. Each value in the vector corresponds to one terrain class, and so the value corresponding to the labelled class would be 1 while all other values would be 0. This is called a *one-hot encoding*.

For example, if we defined the ordering for a one hot encoding to be `[impervious_surface, clutter, car, tree, low_vegetation, building]`, the class of a tree pixel could be described by a vector `[0, 0, 0, 1, 0, 0]`.

Similarly, to convert the predictions of the model back into a colour-coded image, we would have to perform one-hot *decoding*, mapping the binary vectors back to RGB colours.

One-hot encoding and decoding has been implemented with the `NumPy` library, which provides APIs to access C-compiled native code that can run significantly faster than naïve Python code if used properly. However, the code has not yet been fully vectorised to take advantage of `NumPy`. Further optimisations to the `one_hot_labels` module may result in significant performance gains.

4.4 TRAINING METHODOLOGY

Due to time and resource constraints, the training time for each model was limited and was thus held constant. The three models were consequently trained for approximately one week each on an Intel Xeon CPU. As slower models require longer times for each training iteration, a crucial consequence of this training methodology is that slower models had effectively less training than the faster ones.

4.4.1 Train-test Split

To avoid overfitting, machine learning datasets are often split into at least two sets: one for training, and one for validation of the trained model. The model does not train on the validation set; instead, the trained model is used to predict inferences using the input data from the validation set, and the generated predictions are compared to the ground truth to assess the accuracy of the model. Here, 17 of the 24 image tiles in the dataset are used for training, while 7 are withheld for validation.

4.4.2 Patch Selection

The FCN and MobileNetV2 accept (224, 224) images while the PSPNet accepts (473, 473) images. Thus, a method is needed to obtain appropriately sized patches from the 6000x6000 image tiles in the training dataset. This is accomplished by randomly cropping RGB-ground truth pairs from randomly selected tiles.

4.4.3 Data Augmentation

Data augmentation is a procedure for producing more training data by transforming the training dataset, producing variations of the original images. Training a model on augmented data is advantageous as it would learn to be more robust, recognising features and producing good predictions even on data that has been slightly altered.

The following transformations were performed for on-the-fly data augmentation during training:

- Random brightness scaling
- Random 90° rotation
- Random horizontal flips
- Random vertical flips

4.4.4 Batch Size

Neural networks are often trained on *batches* of training data instead of singular data points. This smoothens the learning process as variations between training data updates are averaged out. Larger batch sizes generally provide a greater smoothening effect and are usually preferred, however they use up more memory during training. Thus this is yet another hyperparameter in the training process that must be fine-tuned and tweaked experimentally.

Due to limited RAM on the training machine, the larger models were trained with smaller batch sizes than the smaller ones. This is shown in Table 2.

Table 2: Batch sizes used for training

Model	Batch size
FCN	8
PSPNet	2
MobileNetV2	16

4.4.5 Steps per Epoch

An *epoch* is a unit describing the number of training iterations, generally defined as a single training pass over the entire training dataset. So, training for 100 epochs generally means that the model has trained on the training dataset 100 times.

The number of training samples in such an epoch can be estimated with the following formula:

$$samples\ per\ epoch = \frac{pixels\ per\ tile \times number\ of\ tiles}{pixels\ per\ patch}$$

With 6000x6000 image tiles and 17 tiles, this gives approximately 12,000 samples per epoch for a patch size of (224, 224), or around 1,500 batches (a.k.a. *steps*) with a batch size of 8. However, for convenient scheduling of training, the steps per epoch was set significantly lower at 30 batches. Practically speaking, the number of training steps per epoch does not have any effect on the training process itself, but is rather an arbitrarily defined number for measuring progress during training.

Importantly, however, validation is performed after each epoch, which gives the validation accuracy, used as a gauge for how well the model has learnt at each epoch. The number of validation steps is also configurable, and naturally a larger number of steps would give more accurate estimates of accuracy.

Ideally, each epoch would involve training over the entire training set and validating with the entire validation set. This way, an epoch would be a meaningful unit for measuring training performance. The trainer could then decide to automatically save the best model after each epoch, or decide whether to continue or stop training.

The number of epochs to train for is not set in stone, and, informally speaking, boils down to how long the trainer is willing to wait. Modern CNNs can take days or weeks to train even on high-performance GPU-accelerated hardware. 100 full epochs is suggested as a guideline.

4.4.6 Optimizer and Learning Rate

Various algorithms, called *optimizers*, have been developed for training neural networks. Each algorithm has a different way of updating the model's parameters in response to the gradients derived from each training example. Classic optimizers include Stochastic Gradient Descent (SGD), RMSProp, and Adaptive Moment Estimation (Adam).

Optimizers also come with their own hyperparameters that must be fine-tuned. The most important of these is the *learning rate* which is the main variable that dictates how much the model parameters are adjusted at each training step.

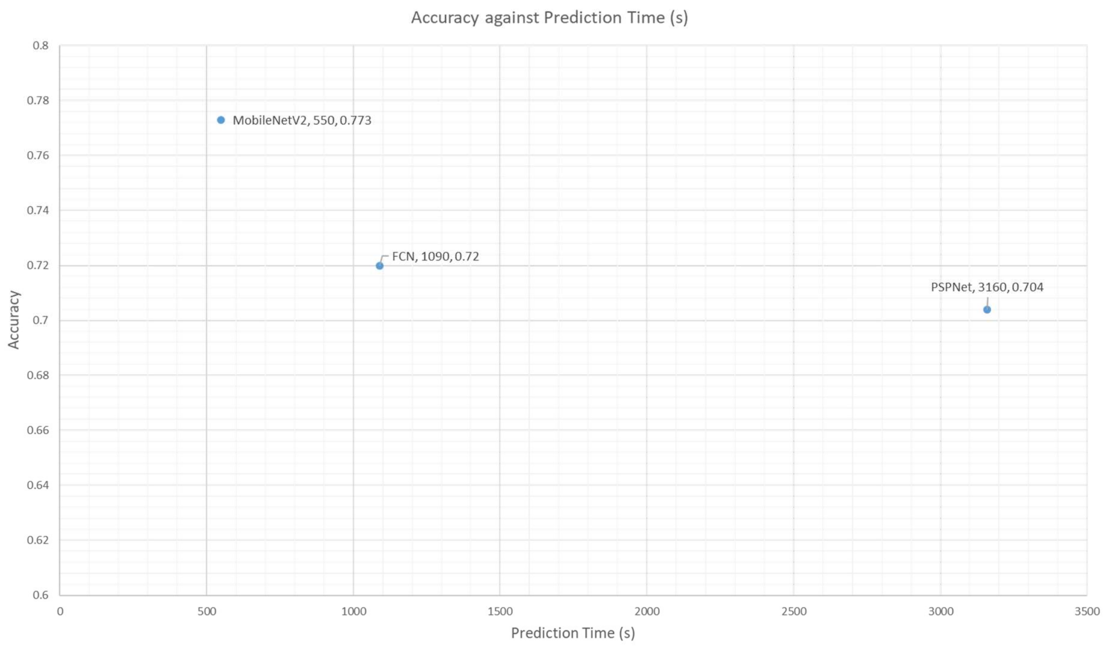
In this implementation, all models were trained with the Adam optimizer with the canonical learning rate of 1e-5. This is another area of opportunity for further optimization as better results may be found with other combinations of optimizers and learning rates.

4.5 USE REQUIREMENTS

As a consequence of the training process, the model is only able to accurately classify aerial images that are true orthophotos at 5cm resolution. While not an issue when working with the Postdam dataset, for practical use, inputs of different scales should be rescaled such that the resolution is 5cm per pixel.

A possible method of teaching the model scale invariance would be to add random rescaling to the list of data augmentation procedures. However, care should be taken when rescaling the ground truth labels to ensure that the resampled pixels are still of the correct colours of their terrain class.

5 RESULTS



	Accuracy (%)	Prediction Time (s)
FCN	72.0	1090
PSPNet	70.4	3160
MobileNetV2	77.3	550

Figure 5: Graph of accuracy against prediction time for the three implemented models.

The trained models were evaluated using tiles 2_12 and 3_12 of the validation set. Accuracy[4] is simply defined as the following:

$$accuracy = \frac{number\ of\ correctly\ classified\ pixels}{number\ of\ pixels}$$

In addition to accuracy, prediction time on a Xeon CPU for a single 6000x6000 tile is also recorded as a measure of speed. The results are summarised in Figure 5. Predictions on an example tile are also shown in Figure 6.

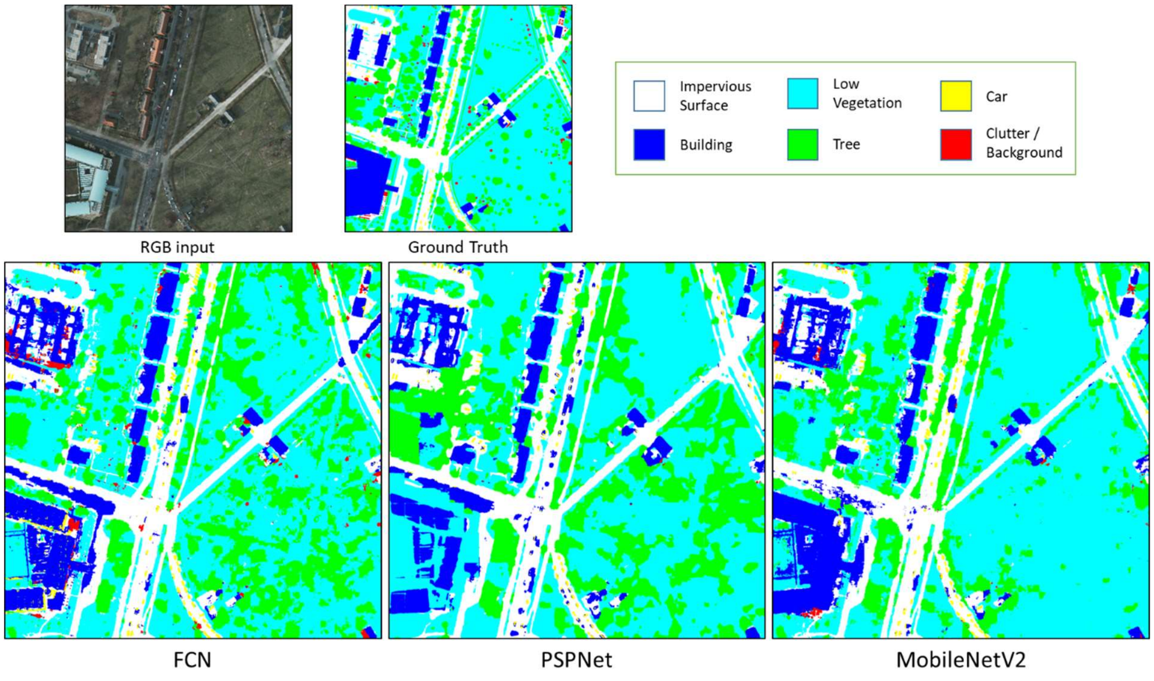


Figure 6: Example predictions for each model.

5.1 Analysis

MobileNetV2 has shown to have the best speed and accuracy, beating the FCN and PSPNet by a wide margin. This is most likely due to the training methodology, which allowed it to train on more samples than the other two models. Indeed, the accuracy of the models seem to be positively correlated with their speed. It is probable that with more training, FCN and PSPNet could meet or even surpass the accuracy of MobileNetV2.

However, although not representative of the true accuracy of fully-trained models, this result is still a testament to MobileNetV2's performance-optimised architecture. This is because the prediction time would still be a significant factor in time-sensitive applications such as AV route planning, as prediction time remains roughly constant and does not change with training.

Another consideration that supports the importance of speedy models is the possibility of on-the-spot training of classifiers, where a human controller provides example classifications of unknown environments during a mission, allowing the classifier to more effectively adapt to environments

[4] This refers to the pixel accuracy. Other statistics that may be of interest are the Jaccard Index (Intersection over Union), or the similar Dice Coefficient (F1 score).

unseen in the training data [19]. In such a system, fast training would be of paramount importance, similar to the methodology used here.

As a caveat, it is important to note that the models tested here are prototype implementations and may not represent the most optimised versions of each architecture. Prediction time may also vary with hardware.

6 Conclusion

This report has shown that deep learning is a promising approach to terrain classification of aerial imagery. The MobileNetV2, in particular, has shown to be a fast and accurate feature extractor, with potential to be used in time-sensitive applications such as AV route planning.

However, further research could be done on improving the accuracy and performance of the classifier. Two possible directions are improving on the design of the model architectures and obtaining more high-quality data for training. The latter is especially important if the model is required to perform on varying kinds of terrain compared to the training dataset. Indeed, it is often said that a machine learning model can only be as good as the data used to train it.

Research in the field of Artificial Intelligence is advancing rapidly, and better solutions to each problem continue to emerge. AI, and deep learning in particular, is definitely an area to watch.

6.1 Possible Extensions

The models were implemented in the Keras high-level API. It is possible that by working with the base TensorFlow code instead, the models could be optimised to run faster. Furthermore, the preprocessing code was written in Python, and could be optimised further by making better use of `NumPy`. Alternatively, processing could even be done in a more performant language such as C++.

Hyperparameter tuning is a crucial part of machine learning development, and is an experimental process mainly accomplished through trial and error. Due to time and resource constraints, the models were to a large extent used without tuning. Further accuracy improvements could result from a good choice of hyperparameters.

As a consequence of the training methodology, the models were not trained as much as they could have been. With better computing hardware, such as GPUs, a large performance boost could be achieved over CPU training, enabling the models to be trained faster. Further training is likely to produce models of better quality. Prediction time would also enjoy a corresponding speedup. The tensorflow-gpu and keras-gpu packages provide facilities for easily enabling GPU computation on NVIDIA GPUs with CUDA Compute Capability >= 3.5.

Training data was of the city of Potsdam, Germany, a temperate country, during winter time. As such the model may have difficulty classifying imagery from other types of biomes such as jungle or desert. To improve the models' accuracy in their target environment, suitable labelled training data that depicts similar areas should be used as well.

Terrain classes that the models were trained to recognise include impervious surfaces, buildings, low vegetation, trees, cars, and clutter/background. If more terrain classes are needed, labelled training data is required to train the model to recognise those classes.

Many CNN architectures for semantic segmentation have been proposed. It would be a worthwhile venture to experiment with different architectures in search of the best performing one for this task. Some architectures that may be of interest are the Xception feature extractor [20] and the DeepLabv3+ decoder module [21].

Another possible method of classifying terrain would be to work with 3D point clouds instead of 2D images. By encoding the 3D shapes of objects in addition to colours, point clouds can provide much more information than still images. OpenDroneMap is an open-source project for processing UAV imagery, and is able to generate point clouds from multiple photos. Possible approaches to point cloud classification can involve other machine learning techniques such as gradient boosted trees [2], however another direction is to use CNNs with 3D convolutions [22].

Finally, once development on the classifier is completed, it would have to be integrated into the AV route planning software. However, as the predictions from the model are unlikely to be 100% accurate, a probability-based approach could be taken using the softmax output together with an index of traversability for each terrain class.

APPENDIX

TRAIN-TEST SPLIT

The following image tiles from the Potsdam dataset were used for training:

2_10, 3_10, 3_11, 4_10, 4_11, 5_10, 6_7, 6_8, 6_9, 6_10, 6_11, 7_7, 7_8, 7_9, 7_10, 7_11, 7_12

The following were used for validation:

2_11, 2_12, 3_12, 4_12, 5_11, 5_12, 6_12

ENVIRONMENT SETUP

The project environment can be setup on an offline system. The following files are required:

1. Anaconda distribution with Python 3.6 installer, which can be obtained from https://www.anaconda.com/download/.
2. Python package installers for dependencies: `absl-py`, `protobuf`, `opencv`, `tensorflow`, and `keras`. If GPU functionality is desired, `tensorflow-gpu` and `keras-gpu` are also required. These Python .whl files can be obtained from the Python Package Index https://pypi.org/.
3. The ISPRS Potsdam dataset (using RGB TOPs) which can be downloaded from http://www2.isprs.org/commissions/comm3/wg4/data-request-form2.html.
4. The `terrainclassifer` project files.

To install, follow the following procedure:

1. Install the Anaconda distribution with Python 3.6.
2. Open the Anaconda Prompt and install each dependency with the command `pip install --no-dependencies <filename>`
3. Copy the Potsdam dataset into .datasets/Potsdam/Training/RGB, .datasets/Potsdam/Training/Labels, .datasets/Potsdam/Validation/RGB, and .datasets/Potsdam/Validation/Labels in the project directory as appropriate.

Project Directory Structure

terrainclassifier

- artifacts
- datasets
    - Potsdam
        - Training
            - RGB
            - Labels
        - Validation
            - RGB
            - Labels
- nets
    - fcn
        - fcn.py
        - resnet50.py
    - mobilenetv2
        - mobilenetv2.py
    - pspnet
        - pspnet.py
- models_config.py
- one_hot_labels.py
- predict_tile.py
- preprocess.py
- ProbabilityTile.py
- train.py

Script Usage

With a trained model, prediction can be performed with the predict_tile.py script. To display help and usage information, use the `python predict_tile.py --help` command from the Anaconda Prompt. Training is done with train.py, and command line usage for this script can be enabled with slight modifications. Detailed instructions are provided as comments in the file. By default, models are saved and loaded from the artifacts folder, although this can be changed via arguments.

REFERENCES

[1] B. Sofman, J. A. Bagnell, A. Stentz and N. Vandapel, “Terrain Classification from Aerial Data to Support Ground Vehicle Navigation,” Robotics Institute Carnegie Mellon University, Pittsburgh, 2006.

[2] C. Becker, N. Hani, E. Rosinskaya, E. d'Angelo and C. Strecha, “Classification of Aerial Photogrammetric 3D Point Clouds,” Pix4D SA, 2017.

[3] R. Hudjakov and M. Tamre, “Aerial imagery terrain classification for long-range autonomous navigation,” Optomechatronic Technologies, 2009. ISOT 2009., 2009.

[4] J. Sherrah, “Fully Convolutional Networks for Dense Semantic Labelling of High-Resolution Aerial Imagery,” 2016.

[5] M. Volpi and D. Tuia, “Dense semantic labeling of sub-decimeter resolution images with convolutional neural networks,” 2016.

[6] J. Peterson, H. Chaudhry, K. Abdelatty, J. Bird and K. Kochersberger, “Online Aerial Terrain Mapping for Ground Robot Navigation,” MDPI, 2018.

[7] “CS231n Convolutional Neural Networks for Visual Recognition,” Stanford University, [Online]. Available: http://cs231n.github.io/. [Accessed 21 May 2018].

[8] I. S. G. H. A. Krizhevsky, “ImageNet Classification with Deep Convolutional Neural Networks,” 2012.

[9] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” ICLR 2015, 2015.

[10] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” CVPR 2016, 2016.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going Deeper with Convolutions,” CVPR 2015, 2015.

[12] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” 2017.

[13] J. Long, E. Shelhamer and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” 2015.

[14] H. Zhao, J. Shi, X. Qi, X. Wang and J. Jia, “Pyramid Scene Parsing Network,” 2016.

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” 2018.

[16] R. E. Lewis Fishgold, “Deep Learning for Semantic Segmentation of Aerial Imagery,” Azavea, 30 May 2017. [Online]. Available: https://www.azavea.com/blog/2017/05/30/deep-learning-on-aerial-imagery/. [Accessed 21 May 2018].

[17] VladKry, “Keras implementation of PSPNet(caffe),” GitHub, [Online]. Available: https://github.com/Vladkryvoruchko/PSPNet-Keras-tensorflow. [Accessed 21 May 2018].

[18] Larry, “MobileNet v2,” GitHub, [Online]. Available: https://github.com/xiaochus/MobileNetV2. [Accessed 21 May 2018].

[19] J. Delmerico, A. Giusti, E. Mueggler, L. M. Gambardella and D. Scaramuzza, ““On-the-spot Training” for Terrain Classification in Autonomous Air-Ground Collaborative Teams,” 2016.

[20] F. Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” 2016.

[21] L. Chen, Y. P. G. Zhu, F. Schroff and H. Adam, “Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation,” 2018.

[22] Y. Ben-Shabat, M. Lindenbaum and A. Fischer, “3D Point Cloud Classification and Segmentation using 3D Modified Fisher Vector Representation for Convolutional Neural Networks,” 2017.