Machine Learning Ⅱ
Final Project Report-Airborne Dataset

04/24/2018

Qianwen Liu

Mark Barna

Thanh Nguyen

Introduction

This project is an exploration of image classification. Our team implemented a Convolution Network in Pytorch to explore DeepSat - 6 AirBorne Dataset.

Deep Learning has gained more popularity recently. Its ability to perform complex training makes it powerful tool. Deeping Learning is state-of-the-art for images and can scale up to big data more naturally than alternative approach.

Satellite image classification is a challenging problem that involves three fields: remote sensing, computer vision, and machine learning. There are limited researches in the field of satellite image classification. Our group found the research of this dataset could be meaningful in many industries, such as aviation and agriculture, and it's also useful in developing and updating maps, monitoring the environment, exploring and protecting natural resources, etc.

We started our project with cleaning and formatting the dataset. Pytorch was used as a framework to build our convolutional neural network. We introduce pooling layers to decrease the size of the input as well as dropout layers to reduce overfitting. At the end, we plot our training loss, kernels, feature map as visual results as part the conclusion of our research.

The main contribution of our work is twofold:

Firstly, we present a great practice of using cloud. We set up our dataset on cloud. In this project, our dataset comes from Kaggle. It's a 2.42 GB dataset, which is too large to run in local machine, so we use Kaggle API to load the dataset into our cloud platform and run the program on cloud all along.

Secondly, we present an implementation of Convolution Network in Pytorch and through vary network configurations and parameters, we aim at finding a balance between increasing the accuracy and decreasing the training time. Our main network consist of convolutional layer, max pooling layer and fully connected layer, upon that structure, we update one parameter a time to compare them and find the best parameters to train the network. We introduce pooling layers to decrease the size of the input as well as dropout layers to reduce overfitting. At the end, our model produces an accuracy of 98.18% with a training time of 2753.4s.

Description of the dataset

The dataset that called DeepSat - 6 Airborne Dataset. Originally, images were extracted 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 authors used the uncompressed digital Ortho quarter quad tiles (DOQQs) which are GeoTIFF images and the area corresponds to the United States Geological Survey (USGS) topographic quadrangles. The average image tiles are ~6000 pixels in width and ~7000 pixels in height, measuring around 200 megabytes each.

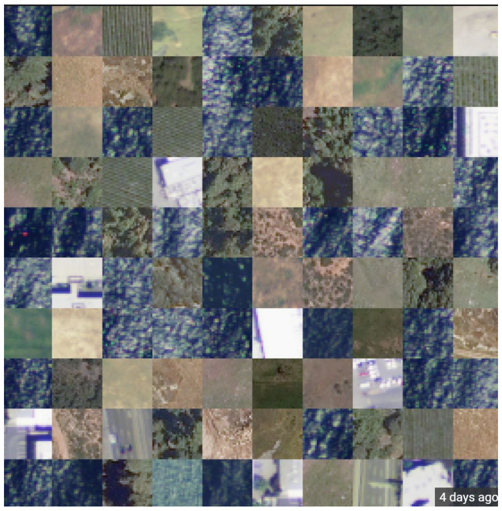


Figure 1. DeepSat-6 sample images.

Our dataset, which contains a total of 405,000 images, is large enough to train a deep network. Each image in the dataset is 28 x 28 pixels and contains 4 bands - red, green, blue, near-infrared. In order to maintain the high variance inherent in the entire NAIP dataset, the author sample image patches from a multitude of scenes (a total of 1500 image tiles) covering different landscapes like rural areas, urban areas, densely forested, mountainous terrain, small to large water bodies, agricultural areas, etc. covering the whole state of California. An image labeling tool developed as part of this study was used to manually label uniform image patches belonging to a particular land cover class.

Once labeled, 28x28 non-overlapping sliding window blocks were extracted from the uniform image patch and saved to the dataset with the corresponding label.

There are 6 different categories, representing the six broad land covers, including barren land, trees, grassland, roads, buildings and water bodies, throughout the whole state of California. Our training and test sets have 324,000 and 81,000 entries, respectively.

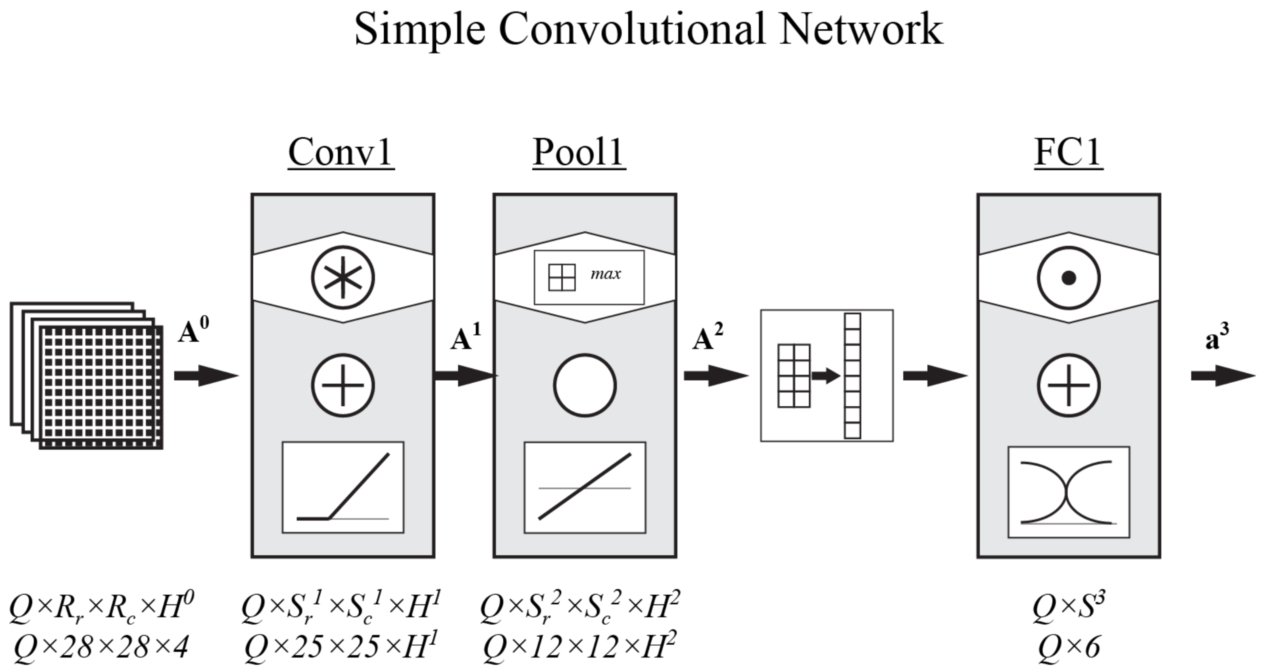
Description of the network and training algorithm

Background

We implement a Convolution Neural Network (CNN) in this research because they are ideal for working with image inputs. Whereas traditional perceptron networks require inputs to be vectors, CNNs can accept inputs in higher dimensions. They operate by passing a filter or kernel over the image. The kernel is a matrix, each element of which is a parameter of the network. Most CNNs contain a mix of convolution, pooling (to reduce the image size), and linear layers, as we will show here.

Network Architecture

In this CNN, we vary the number of convolution and *max pooling* layers. We decided to start with the simplest possible CNN as a benchmark: one convolutional layer, incorporating a *ReLu* transfer function, feeding into a *max pooling* layer, and finishing in a linear layer (see network diagram).



Simple Convolutional Network
Conv1
Pool1
FC1
max
$\mathbf{A}^0$
$\mathbf{A}^1$
$\mathbf{A}^2$
$\mathbf{a}^3$
$Q\times R_r\times R_c\times H^0$
$Q\times 28\times 28\times 4$
$Q\times S_r^1\times S_c^1\times H^1$
$Q\times 25\times 25\times H^1$
$Q\times S_r^2\times S_c^2\times H^2$
$Q\times 12\times 12\times H^2$
$Q\times S^3$
$Q\times 6$

Figure 2. Simple Convolutional Network Diagram.

In the convolutional layer, we used a kernel size of $S_r^1 = S_c^1 = 4$, an even factor of 28 (the input image's height and width), with stride of 1. We experimented with a range of kernel quantities in this layer ($H^1 = 4,\ 8,\ 16,\ 32$). In the *max pooling* layer, we used a kernel size of $S_r^2 = S_c^2 = 3$

with a stride of 2 to evenly reduce the image to $Q \times 12 \times 12 \times H^2$ (where Q = *batch size* and $H^1 = H^2$ since pooling layers maintain the prior layer's channel dimension). Finally, the fully-connected linear layer outputs a tensor of $Q \times 6$, corresponding to the six classes of terrain. The linear layer contains a *softmax* transfer function because we are performing classification. The *softmax* outputs a probability distribution corresponding to the likelihood of the sample belonging to a particular class.[1] Before the output of the pooling layer can be processed by the linear layer, it must be transformed from a matrix into a vector (per batch) so that it can pass through the dot product function.

A pass through any specific layer takes the following form:

$$a^{m+1} = f^{m+1}\left(o^{m+1}\left(h^{m+1}\left(W^{m+1}, a^m\right), b^{m+1}\right)\right) \text{ for layers } m = 0, 1, ..., M-1,$$

where $h(W^{m+1}, a^m)$ is the weight function, $o(h^{m+1}, b^{m+1})$ is the net input function, $f(n^{m+1})$ is the transfer function and M represents the last layer of the network.

Each of the three types of layer employed in this network calculate according to the following formulas:

Convolutional layer with ReLu transfer function:

$$n_{i,j} = \sum_{k=1}^{c} \sum_{l=1}^{r} w_{k,l} v_{i+k-1, j+l-1} + b \text{ or in matrix form: } N = W \circledast P + b$$

$$A = \{0,\ N < 0;\ N,\ N \geq 0$$

Max pooling layer with linear transfer function:

$$n_{i,j} = max(v_{r(i-1)+k, c(j-1)+l} | k = 1, ..., r; l = 1, ..., c) \text{ or } N = MAT_{r,c}^{max} V$$

Fully-connected layer with softmax transfer function:

$$n = WP + b$$

$$a = \frac{e^n}{\sum_{i=1}^{6} e_i^n}$$

After several training runs with varying hyperparameters using the simple CNN network, we implemented a variant of the AlexNet architecture (we call it "QuasiAlexNet") to see if we could improve upon our testing accuracy. The original AlexNet was trained on images of size

[1] The *softmax* transfer function is shown as part of the linear layer since it resides here from the network architecture's point-of-view. As a practical matter, PyTorch incorporates the *softmax* function into its *cross-entropy* loss function class so in the our code, the *softmax* is not included in the network class.

227×227. Since our images are roughly one-eighth the size (28×28), we scaled down the number of kernels and their sizes roughly proportionally. We felt this would be an appropriate decision given that a smaller image contains less complexity. We also removed two of the *max pooling* layers so as not to reduce the image resolution too much in the first layers of the network. Further, the original dataset had 1,000 classes, whereas ours only has six.

The AlexNet architecture also employs dropout in the first and second fully-connected layers. Dropout randomly inhibits the output from individual neurons with a specified probability (p=0.5 in this case) as a way to reduce overfitting. We incorporated this same dropout structure in our QuasiAlexNet setup. Finally, the AlexNet authors implemented their architecture on two GPUs, running in parallel because of memory limitations at the time (Krizhevsky et al. 2012).. We have a single GPU (see diagram below for comparison of architures).

Performance Function

Because this is a classification problem with more than two classes, the *cross-entropy* loss function is the ideal choice for the performance index especially when paired with a *softmax* output layer. *Cross-entropy* is calculated as follows (Dahal):

$$\widehat{F}(x) = -\sum_{i=1}^{6} t_i log(a_i),$$

where *a* is the neuron output, the probability of the image being in class *t*, calculated for each of our six classes, which is given by the output of the *softmax* function.

Parameter Updates

As with a traditional multilayer perceptron network (MLP), after completing a forward pass and calculating the error as given by the *cross-entropy* ($\widehat{F}(x)$), we need to update the network parameters. We employed gradient descent with mini-batches as our optimization method, testing batch sizes of $Q = 10,\ 100,\ 1000$. Because the training set contains 324,000 images, it would be inefficient to attempt true stochastic gradient descent with no batching. The weight and bias update rules (using mini-batches of size *Q*) after completing forward pass *k* are (Demuth et al. 2014):

$$W^m(k+1) = W^m(k) - \tfrac{1}{Q}\sum_{q=1}^{Q} \alpha \frac{\partial \widehat{F}}{\partial W_q^m}$$

$$b^m(k+1) = b^m(k) - \tfrac{1}{Q}\sum_{q=1}^{Q} \alpha \frac{\partial \widehat{F}}{\partial b_q^m}$$

We initialized the network with a learning rate of $\alpha = 0.001$ and later increased to $\alpha = 0.005$ to try to speed the network training. After noting an increase in oscillations of the loss curve over time, we incorporated momentum into the update rule. Momentum adds a low-pass filter into the calculation, which serves to reduce the frequency of the oscillation in the loss. The weight update rule for with momentum of ρ as implemented in PyTorch is calculated by:

$$W^m(k+1) = W^m(k) - v(k+1) \text{, where}$$
$$v(k+1) = \rho v(k) + \alpha \frac{\partial \widehat{F}}{\partial W^m}$$

(We give the rule in SGD form without the summation over batches for simplicity, but the gradient term is averaged over each batch as above.)

The bias update rule is performed analogously, but we omit the formula for space. We tested the network at $\alpha = 0.005$ with $\rho = 0.8,\ 0.9$.

Because the original AlexNet incorporates a weight decay coefficient as well, we also added this term when implementing our QuasiAlexNet. The weight update rule (for SGD) with weight decay of ω is (Krizhevsky et al. 2012):

$$W^m(k+1) = W^m(k) - v(k+1) \text{, where}$$
$$v(k+1) = \rho v(k) - \omega\alpha W^m(k) + \alpha \frac{\partial \widehat{F}}{\partial W^m}$$

Krizhevsky et al. determined that a small amount of weight decay helped the network converge so they set $\omega = 0.0005$, which we mimicked in our QuasiAlexNet training runs. In general, weight decay provides a form of regularization in networks.

Backpropagation

As with the MLP network architecture, we must implement backpropagation in order to calculate the performance function derivatives with respect to the weights and biases ($\frac{\partial \widehat{F}}{\partial W^m}$ and $\frac{\partial \widehat{F}}{\partial b^m}$). Starting with the last layer, we apply the chain rule to replace $\frac{\partial \widehat{F}}{\partial W^M}$ with $s^M A^{m-1}$, because

$$\frac{\partial \widehat{F}}{\partial W^M} = \left(\frac{\partial \widehat{F}}{\partial n^M}\right)^T \frac{\partial n^M}{\partial W^M}$$
$$A^{m-1} = \frac{\partial n^M}{\partial W^M}$$
$$s^M \equiv \frac{\partial \widehat{F}}{\partial n^M} \text{ by definition}$$

Calculating the derivative of the *cross-entropy* function w.r.t. to the net input of the last layer (where the summation runs across our six classes) (Dahal):

$$s^M = \frac{\partial \widehat{F}}{\partial n^M} =- \sum_{i=1}^{6} \frac{\partial(t_i\, ln(a_i^M))}{\partial n_i^M} =- \sum_{i=1}^{6} \frac{t_i}{a_i^M} \times \frac{\partial(a_i^M)}{\partial n_i^M} =- \sum_{i=1}^{6} \frac{t_i}{a_i^M} \times \dot{\mathrm{F}}^m(n^M)$$

Since $\dot{\mathrm{F}}^m(n^M)$ is the derivative of the *softmax* transfer function, the last layer sensitivity reduces nicely to:

$$s^M = \sum_{i=1}^{6} a_i^M - t_i$$

We then backpropagate the sensitivities through the layers with the chain rule Demuth et al. 2014):

$$s^m = \frac{\partial \widehat{F}}{\partial N^m} = \left(\frac{\partial N^{m+1}}{\partial N^m}\right)^T \frac{\partial \widehat{F}}{\partial N^{m+1}}\text{ , which simplifies to:}$$

$$s^m = \dot{\mathrm{F}}^m(N^m)(W^{m+1})^T s^{m+1}\text{ ,}$$

where $\dot{\mathrm{F}}^m(N^m)$ is the derivative of the transfer function of layer *m*.

The pooling layer has no parameters to adjust, an advantage of the *max pooling* operation over *average pooling*. The convolutional layer takes the following backpropagation order (Jafari 2018):

$$dA^m \rightarrow dN^m \rightarrow dZ^m \rightarrow dA^{m-1}$$

where **dA** is the derivative of the performance function $(\widehat{F})$ w.r.t. the layer output; **dN** is the derivative of $\widehat{F}$ w.r.t. the net input; **dZ** is the derivative of $\widehat{F}$ w.r.t. the net input function, for layers $m = 0, 1, ..., M-1$.

Once we complete the calculation of gradients back through the network, we can update the parameters and proceed with training.

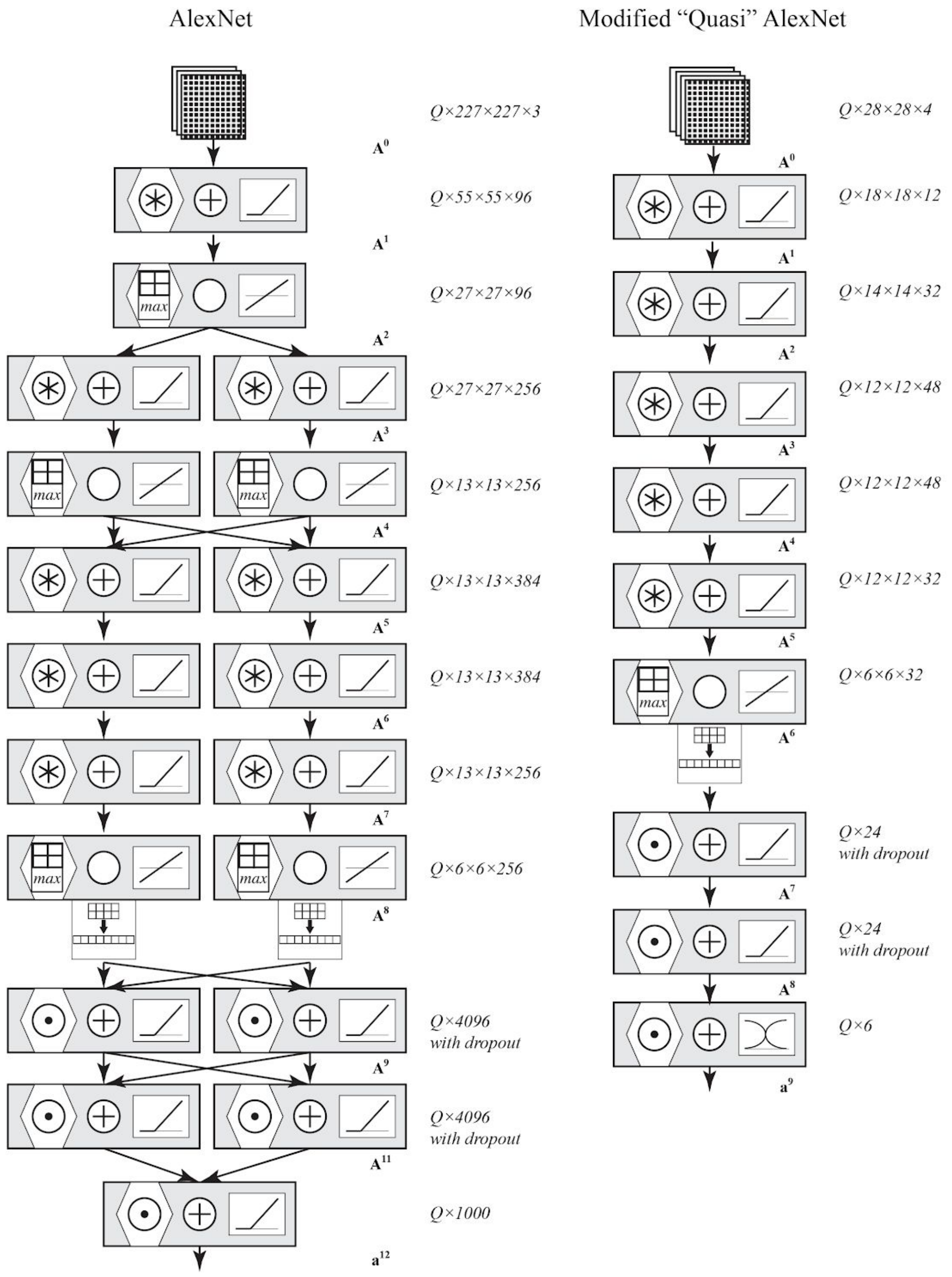


Figure 3. AlexNet and Modified “Quasi” AlexNet Diagram.

Experimental Setup

Data Collection

The dataset size is over 2GB and is splitted into X_train, Y_train, X_test and Y_test sets. Since github repository size is restricted to under 1GB, we utilized Kaggle APIs to download the dataset to our individual cloud instances without committing it.

Data Integrity

We did integrity checks on the dataset and the dataset looked good. The checks were to make sure that the train and test sets have correct number of entries and layers with no null values and the values must be between 0 and 255 for X sets and between 0 and 1 for Y sets. The train set was shuffle each time to prevent mini batch only contains one single class that could happen with a sorted dataset.

Network Implementation

We decided to use Pytorch for our experiment. The reason we chose Pytorch over Caffe and TensorFlow is because it has an extensive documentation as well as the flexibility that the other frameworks lack. Pytorch Module, providing us the flexibility to customize our network with different layers and parameters, and cuda, allowing us to use GPUs, were used to implement our convolutional network. As mentioned in our training algorithm, mini-batches were also used in our optimization method because of the big dataset size.

Parameter Updates

In order to decide on our network parameters, the network was tested with a small subset where only one parameter was updated at a time and training time and accuracy rates were compared to get the most optimal value.

Network Evaluation

Performance of the network was evaluated by a combination of different factors, including training time, losses and accuracy rates. The most fitted network should yield a significant increase in network accuracy together with low losses within a reasonable training time.

Overfitting

An overfitting network could be detected by looking at the train and test accuracy rates. If the training accuracy is high and test accuracy is low, the network is overfitting. This can be prevented by introducing dropout layers into the network.

Result

The training results for various network configurations and parameters were compared in order to select the best optimal network. Out of all the networks that we tested, six were recorded altogether with their configurations and results in the table below.

		QuasiAlex 1	QuasiAlex 2	Simple CNN 4	Simple CNN 8	Simple CNN Momentum 1	Simple CNN Momentum 2
Network Parameters	**Learning rate**	**0.01**	**0.005**	**0.001**	**0.005**	**0.005**	**0.005**
	Momentum	**0.9**	**0.95**			**0.8**	**0.9**
	Weight Decay	**0.0005**	**0.0005**				
	Epochs	**10**	**10**	**10**	**10**	**10**	**10**
Network Configuration	**Convolutional Layers**	**5**	**5**	**1**	**1**	**1**	**1**
	Pooling Layers	**1**	**1**	**1**	**1**	**1**	**1**
	Linear Layers	**3**	**3**	**1**	**1**	**1**	**1**
Result	**Training time(s)**	**3243.2**	**3350.1**	**2744.9**	**2878.8**	**2764.9**	**2753.4**
	Accuracy rate(%)	**97.18**	**96.76**	**96.71**	**97.30**	**98.06**	**98.18**

Table 1. Network architectures and their results.

QuasiAlex 2 had the longest training time while Simple CNN 4 had the shortest training time and Simple CNN Momentum 2 and Simple CNN 8 had the highest accuracy rate while Simple CNN 4 had the lowest accuracy rate (**Table 1**). Based on the criterias that we decided for evaluating our network, we chose Simple CNN Momentum 2 as our final network. As discussed in the experiment setup section, we wanted to pick a network could yield a high accuracy rate within a reasonable timeframe and Simple CNN Momentum 2 met both criterias. It had the highest accuracy rate, 98.18% as well as the second shortest training time, 2753.4s.

Network Discussion

Layer (Type)	Input shape	Kernel size	Stride	Output shape
Conv2d	28x28x4	4x4	1	25x25x16
MaxPool2d	25x25x16	3x3	2	12x12x16
Linear	12x12x16			1x6

Table 2. Simple CNN Architecture

As discussed in the description of the network section, our network started with a simple CNN with 1 convolutional, 1 pooling and 1 linear layer then expanded to a more complexed QuasiAlex CNN. With the training time increased significantly (~500s) while only a small hike in accuracy rate, we decided to use the simple CNN architecture. Learning rate of $\alpha = 0.005$ was used in order to speed up the training process and momentum was also implemented to reduce the frequency of oscillation in the loss. With momentum of 0.9, the network was trained ~ 100s faster while still yielded a similar result. Therefore, we settled on 0.9 for momentum. Different batch size (Q = 10, 100, 1000) were also tested. The batch size of 10 took approximately 8.5 minutes longer to train than the batch size of 100, with the latter not losing accuracy. On the other hand, the batch size of 1,000 did suffer a loss in testing accuracy over the same number of epochs while not gaining an advantage in time. Since we would expect a larger batch size to require more epochs to reach the same level of accuracy (if it ever did), there was no advantage in using a size of 1,000 (since it would take more time) so we settled on a size of 100.

Result Discussion

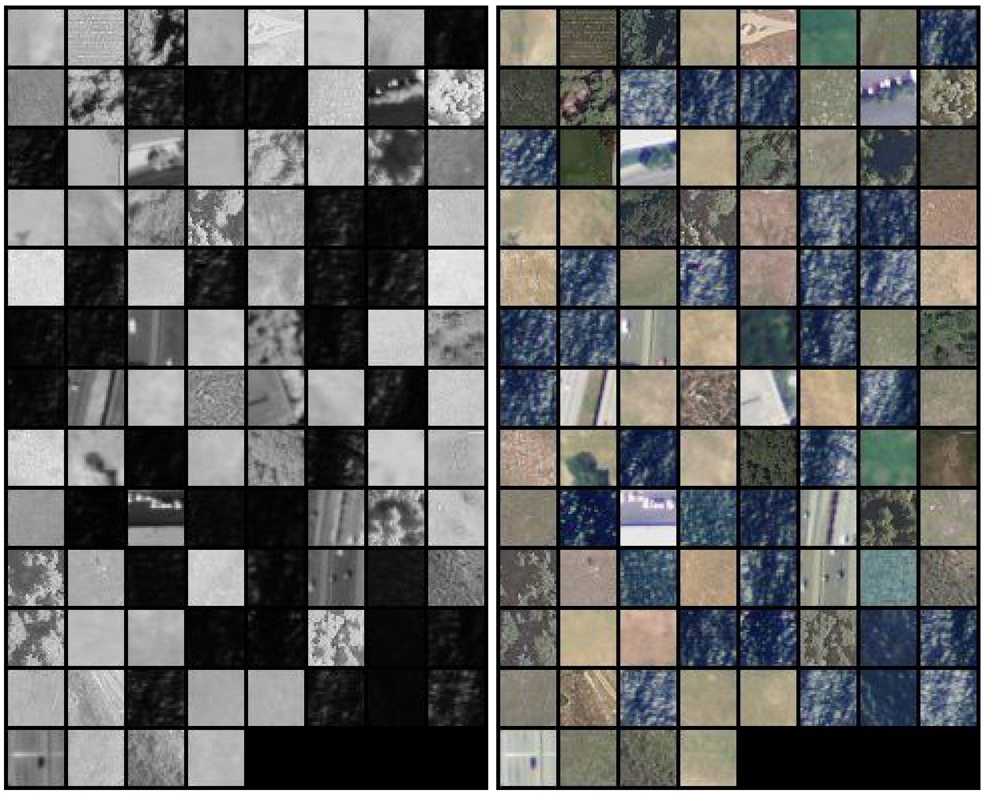


Figure 4. Sample NIR and RGB

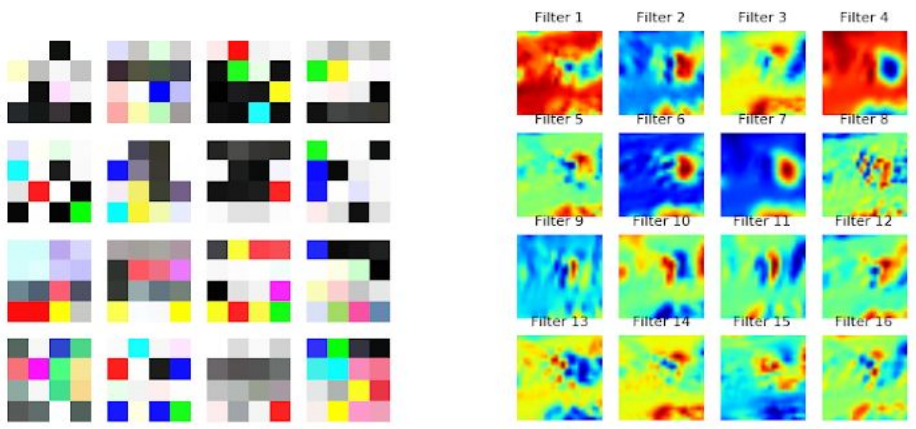


Figure 5. All kernels for convolutional layer and sample feature maps

Figure 4 shows samples of splitted RGB and NIR layers for one of our mini batches. There are a total of 16 4x4 kernels for the convolutional layer which produces 16 different feature maps. All kernels and sample feature maps of an image after one run are shown in **Figure 5**.

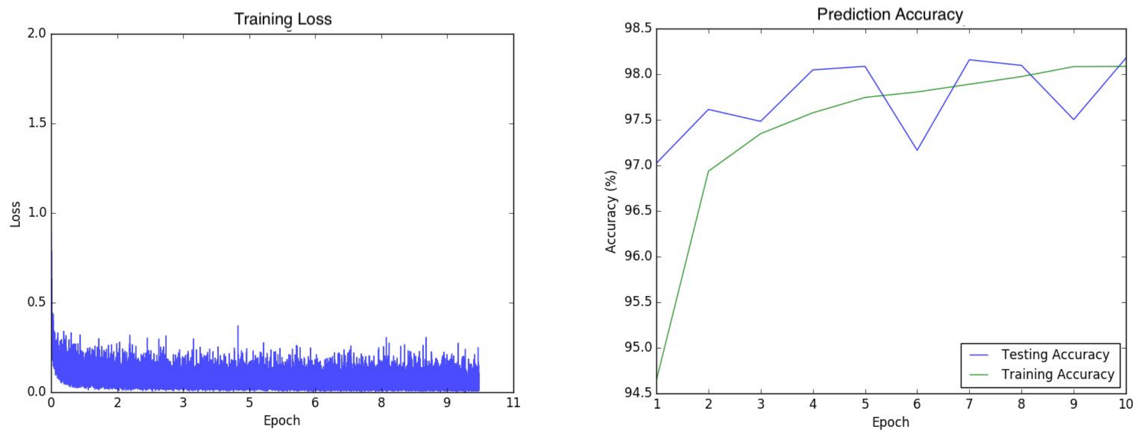


Figure 6. Training Loss and Accuracy Plot

The first graph in **Figure 6** shows the training loss trend. Training loss gradually decreased after each epoch with the average training loss was ~0.06 in the last epoch.

The second graph shows the training and testing accuracy. Training accuracy gradually increased after each epoch despite a big hike between the first and second and stabilized after epoch 9. On the other hand, even though testing accuracy did not show a consistent trend, the differences between testing and training accuracy were small (< 1%) suggesting that the network is not overfitted.

Even though we couldn't get the training loss close to 0, we still think this is a good network overall with a high accuracy rate 98.18% and an acceptable loss value.

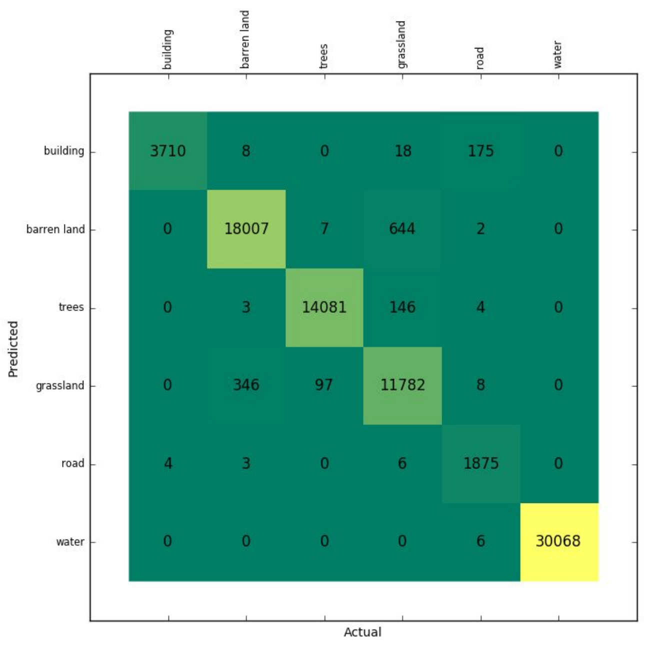


Figure 7. Confusion Matrix

Our network correctly classified 79523 out of 81000 test images (98.18%). Out of 6 land cover classes, water cover land had the highest successful rate (99.98%) while building cover land had the lowest successful rate (94.86%). We could see that there were some mix-ups between barren land, trees and grassland and between building and road. Out of the misclassifications, more than 97% barren land and trees were misclassified as grassland, ~78% grassland were misclassified as barren land and ~87% building were misclassified as road (**Figure 7**).

Summary

Review

The purpose for this project is to practice what we've learnt about Deeping Learning. In this project, we produce a convolutional network in Pytorch. Our network produces an accuracy of 98.18% , which we consider is a great result for the training of the model.

During the process of building the network, we start with picking up the dataset. The Airborne Dataset is a meaningful dataset, it's an application of remote sensing and Deep Learning. We're interested in land cover classification. The problem of detecting various land cover classes in general is a difficult problem considering significantly higher intra-class variability in land cover like trees, water bodies, grassland and roads, etc. There has been limited research in the field of satellite image, so what we achieved could be a good reference for further study.

Pytorch has gained a lot popularity. Compared to Keras, it's flexible and user-friendly and other advantages are it's multi-GPU support, custom data loaders and simplified preprocessors. We find it faster to develop a network with Pytorch and great experience to run on GPU.

When developing the network, the hard part is not building architecture, it's how adjust our configurations and parameters to get higher accuracy and shorter training time. In this process, through experimenting and comparing, we gained a better understanding of the differences between training functions, batch size, momentums and so on. Knowing the calculations behind them can be solid foundation for us to use them.

Recommendations

1. As for the dataset, data is clean when we fetched them since they have been organized by the owner. Next time we can try a messy dataset which is most commonly seen in real world data. we also believe it more data points for other classes since there is quite a big gap in sample sizes between water and other classes, which is the reason why we got the highest accuracy in classifying water bodies.

2. To reduce the bias in the confusion matrix, we can try tuning the decision boundaries in a proper way in certain classes.

3. We can split the data into RGB and NIR layers and train them separately.

4. The author originally used a window size of 64 x 64 to derive contextual information. For general classification problem, a 64 x 64 window is too big a context covering a total area of 64m x 64m. We are also interested in trying images that are larger than 28x28, which would increase the complexity.

Bibliography

Dataset

Basu, Saikat, Sangram Ganguly, Supratik Mukhopadhyay, Robert Dibiano, Manohar Karki and Ramakrishna Nemani. "DeepSat - A Learning framework for Satellite Imagery" ACM SIGSPATIAL 2015. Dataset hosted on Kaggle. https://www.kaggle.com/crawford/deepsat-sat6 (accessed April 2018).

Other References

Dahal, Paras. "Classification and Loss Evaluation - Softmax and Cross Entropy Loss." Web article on DeepNotes. https://deepnotes.io/softmax-crossentropy (accessed April 2018).

Demuth, Howard B. and Beale, Mark H. and De Jess, Orlando and Hagan, Martin T. 2014. *Neural Network Design (2nd ed.)*. United States: Martin Hagan.

Jafari, Amir. 2018. "Convolution Networks". Slides from lectures presented at the George Washington University: Washington, DC.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2013. "ImageNet Classification with Deep Convolutional Neural Networks." Advances in Neural Information Processing Systems. 2012, 25: 1097-1105.

Li, Fei-Fei, Justin Johnson, and Serena Yeung. 2017-18. "Convolutional Neural Networks for Visual Recognition." Lecture slides, videos, and notes from course given at Stanford University: Palo Alto, CA. http://cs231n.stanford.edu/syllabus.html

PyTorch (ver 2.0.1). Modules, tutorials, documentation. pytorch.org. (accessed April 2018).