Udacity Machine Learning Engineer Nanodegree Project 5 - Deep Learning for Satellite Image Recognition

Jason Osajima

1 Introduction

In the past few years, development of affordable satellites has increased the availability of satellite images. Several startups and companies are leveraging image recognition powered by machine learning algorithms and the availability of satellite images to produce actionable insights about a variety of subjects.

For example, Skybox Imaging was able to use satellite images to count all the cars in every Walmart parking lot in America on Black Friday [1]. Others are using satellite images to predict seasonal yields and monitor crop health, estimate the world's supply of oil, and map human rights infractions in developing countries.

One type of machine learning algorithm that has gained popularity for image classification in recent years are convolutional neural networks (CNNs). CNNs are a type of machine learning algorithm that were inspired by the organization of the neurons in an animal visual cortex. For image classification, the algorithm takes several images, makes a prediction about what class they belong to, and adjusts the way it makes its prediction based on how close its prediction was compared to the classes the images actually belong to. This process is repeated several times until the CNN arrives at a decent model which can make accurate predictions on images it hasn't seen yet.

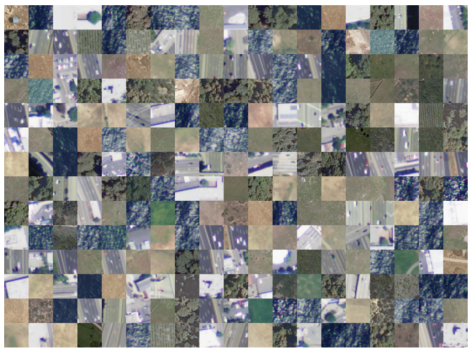
In this study, we use a CNN to classify satellite images into 6 different land cover types ('buildings', 'barren_ land', 'trees', 'grassland', 'roads', 'water_ bodies'). In Section 2 we explore the dataset and preprocess the images. In Section 3 we describe the architecture of the CNN, and in Section 4 we share the results of training and testing our CNN.

[1] http://spectrum.ieee.org/aerospace/satellites/9-earthimaging-startups-to-watch

2 Dataset

The dataset was created in part of a study first authored by Saikat Basu of Louisiana State University [2]. The images were taken from the National Agriculture Imagery Program (NAIP) dataset, and consists of 405,000 images. Each image is a picture of land cover taken at a 1-m ground sample distance and is 28 x 28 pixels, with 4 bands (red, green, blue, and near-infrared). Each image is labeled with its corresponding land cover type. Figure 1 shows a sample of the images.

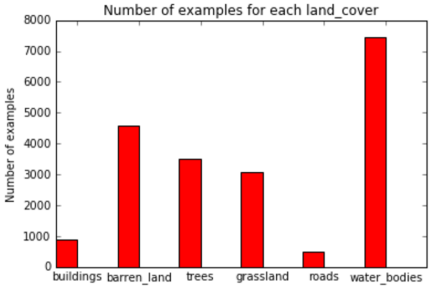
Figure 1: A sample of images in the SAT6 dataset



2.1 Data Exploration

Next, we check to see if the data is balanced across classes.

Figure 2: Distribution of examples for the Training Set



[2] http://csc.lsu.edu/ saikat/deepsat/

Figure 3: Distribution of examples for the Valid Set

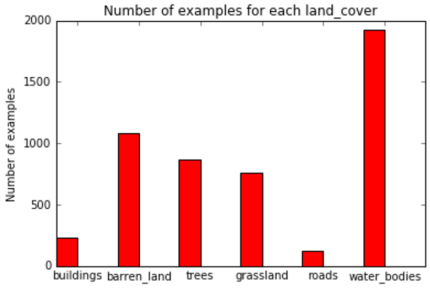
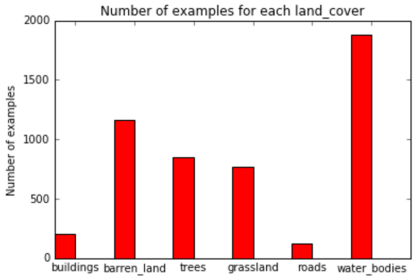


Figure 4: Distribution of examples for the Test Set



By visual inspection, the data looks balanced across sets. While the data is balanced across sets (training, validation, and testing) the data is not balanced across classes within each set. This is not necessarily bad, since the distribution of the classes reflects the distribution of land cover types that a classifier trained on this data will see.

Using all 405,000 images to train and test our CNN is beyond the scope of this project. Instead, we will cut our dataset to 20,000 training images and 5,000 test images.

2.2 Preprocessing

Before we started feeding our images through our CNN, we need to preprocess the images. When executing numerical computations, we have to be concerned about calculating values that are too large or too small, which can introduce error into our calculations. In order to avoid error, we want our mean to be equal to zero and our variances to be equal across variables.

Our dataset includes the pixel values of four layers (Red, Green, Blue, and Near-Infared) with values between 0 and 255. We can normalize the values, which will convert each pixel value to a value between -1 and 1.

3 CNN Architecture

Our CNN consists of two convolution layers followed by a fully connected layer. The architecture is inspired form the CNN tensoflow tutorial [3]. Let's investigate what happens at each level of our CNN.

The 28x28 pixel image with 4 feature maps is fed through the first convolutional layer. We take a small patch of the image (in the case of the first convolutional layer, the patch size is 5x5) and run it through a tiny neural network and receive 32 outputs. If you can imagine, the first patch that we run through the tiny neural network is in the bottom left corner. We then shift to the right by 1 (since our stride size is 1) and run it through the tiny neural network again. We slide across and then vertically until the entire image has been processed. Since we are using same padding, the new image that is formed from the compilation of the outputs of our tiny neural networks is now 28x28 with a depth of 32.

When we run it through the neural network, the patch is multiplied by a matrix of weights and then a bias is added to it. Next, the resulting matrix is run through a rectified linear unit (RELU) function, which produces 0 if x is less than 0 and x if x is greater than x. Next, we implement max pooling, where we look at every point on the feature map, look at a small neighborhood around that point, and compute the maximum of all the responses around it. Using max pooling often makes our model more accurate [4].

The data flows next into the second convolutional layer. The second layer is similar to the first, except our output now has a depth of 64 as opposed to 32.

After going through the second convolutional layer, the data goes into a fully connected layer. We reshape the output to fit the fully connected layer input first and then multiply by a set of weights and add a set of biases, which gives us an output of 1024 units. We then pass the output through a

[3] https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html
[4] http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/

RELU function and apply dropout.

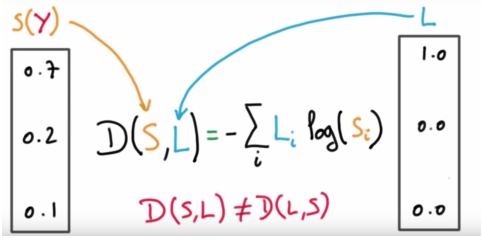
Dropout is another technique to prevent overfitting. It works in the following way: randomly select half (or for our purposes, one out of four) of the activations that go from one layer to the next and drop them, or randomly set them to 0. When we initiate dropout, the network can never rely on any given activation to be present because it might be squashed. So it is forced to learn a redundant representation for everything to make sure that at least some of the information remains. During training we zero out the activations that we drop and scale the remaining activations by a factor of two. This way, when it comes time to average them, we just remove the dropout and scaling and we get something that is properly scaled. It is only used during training and not evaluation [5].

The data is now a 6x1 matrix, with each element representing a land cover class. Our model then takes the element with the maximum value and makes a prediction about what land cover class our example belongs to.

3.1 Training the CNN

In the training phase of the CNN we use stochastic gradient descent to minimize our cost function. Our cost function is used to assess how accurate the predictions of the model are. For our purposes, we measure the cross entropy between the predictions the model makes and the actual label [6]. Cross entropy is essentially the distance between the vector of the prediction and the vector of the label.

Figure 5: A visual representation of the Cross Entropy Function



[5] https://www.udacity.com/course/deep-learning–ud730
[6] https://www.udacity.com/course/deep-learning–ud730

In the figure shown above, D represents the cross entropy function, S represents a vector of all the prediction vectors, L a vector of all the label vectors, and L_i and S_i represent the prediction and label vector for the ith example.

We want to try and minimize the entropy function as best as we can. In order to find the minimum value of the entropy function, we can take the derivative of it, set it equal to 0, and solve for the set of weights and biases values.

The method we use to minimize the entropy function is gradient descent. We can use gradient descent algorithms to find the optimal weights and biases by taking the derivative of our cost function and iterate several times to minimize our cost (or loss).

It would be difficult to iterate through our dataset, since we have several thousand pictures. We can use stochastic gradient descent to speed up this process. Stochastic gradient descent takes a random sample of a fraction of the total training set (for our purposes we are using a batch of 128 examples) and iterates through the model and finds the optimal weights and biases to minimize the cost. Stochastic gradient descent will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yields a result that is close enough.

There are two ways that we can help stochastic gradient descent reach an optimal minimum value for our cost function. The first way is using momentum. Momentum uses a running average of the weights and biases of the gradients as opposed to using the weights and biases generated from the latest batch to update the weights and biases for our model. The second way is using learning rate decay. As we get closer to our (hopefully) global minimum, we want to take smaller and smaller steps. Learning rate decay enables this, and allows the weights and biases generated by the batch to affect the weights and biases less and less as we iterate through more and more batches. We will implement both by using the AdamOptimizer, available through tensorflow.

3.2 Metrics

In order to test the accuracy of our model, we will use two metrics. To begin with, we will use a simple accuracy metric, which is the ratio between the

number of correctly identified examples and the total number of examples. Secondly, we will use a F_1 metric and confusion matrix. Since our dataset is relatively unbalanced between classes, a general accuracy metric may not capture the accuracy of our model for particular classes that show up less often. An F_1 score is the harmonic mean between precision and recall. Precision is the ratio of true positives to all positives, while recall is the ratio of true positives to all that were classified correctly. A confusion matrix will illustrate the performance of our classifier for each land class, and therefore give us a better sense of the land cover classes where the classifier is not performing as well. We will use a normalized confusion matrix since our classes are unbalanced.

As a benchmark for our model, we will use the overall accuracy of 79.06% attained from the CNN model used by Basu et al. in 2015. Since we are using 20,000 training images and 5,000 test images, our model will not be directly comparable to their CNN model, since they used the entire dataset of 500,000 images when training and testing their models.

4 Results, Refinement and Conclusion

4.1 Results

Using the CNN architecture described above, we constructed a model using the following parameters:

- learning_rate = 0.001
- training_iters = 50,000
- batch_size = 128
- display _step = 10 [7]
- n_input = 3136 [8]
- n_classes = 6

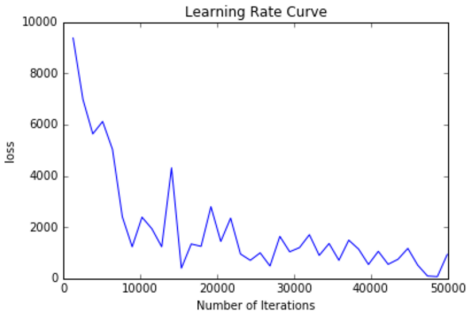
[7] The display_step determines how often we report the accuracy and cost of the model. With a display step of 10, we display the accuracy and cost after every $128 * 10 = 1280$ iterations.

[8] There are 28*28 pixels in each picture, and 4 map features (red, green, blue, near-infared). Therefore, we have $28 * 28 * 4 = 3136$ inputs

- dropout = 0.75
- training _set = 20,000
- validation _set = 5,000

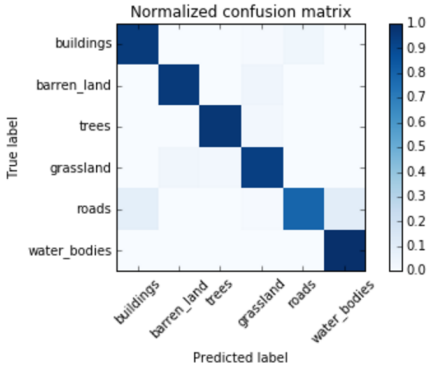
We have achieved an accuracy of 96.90% and F_1 score of 0.9689 on the test set, which is an improvement from our benchmark accuracy. The training time for 50,000 iterations of stochastic gradient descent was 244 seconds and prediction time on the validation dataset was 8.89 seconds. Figure 6 shows the value of our loss for each iteration at each display step.

Figure 6: Learning rate curve for cnn_1



Since our accuracy and F_1 scores are comparable, we can roughly conclude that our prediction is uniformly accurate for different land cover classes. We can verify this by looking at our normalized confusion matrix:

Figure 7: Normalized Confusion Matrix for cnn_1



The matrix shows that the algorithm performed the worst for the roads land cover class which makes sense, since we have the lowest number of roads in our training set.

4.2 Refinement

The first way we can refine our classifier is by increasing the number of iterations. Looking at our learning curve, it seems that the algorithm has not yet reached a local minimum and could benefit by more iterations. Let's increase our number to 100,000. The second parameter that we can change to refine our model is dropout. I want to try increasing dropout to 0.5 in order to make the nodes in our model less dependent on other nodes and hopefully allow the model to generalize better, increase accuracy, and prevent overfitting since we are doing more iterations of our algorithm.
Using the CNN architecture described above, we constructed a model using the following hyperparameters:

- learning_rate = 0.001
- training_iters = 100,000
- batch_size = 128
- display _step = 10 [9]
- n_input = 3136 [10]
- n_classes = 6
- dropout = 0.5
- training _set = 20,000
- validation _set = 5,000

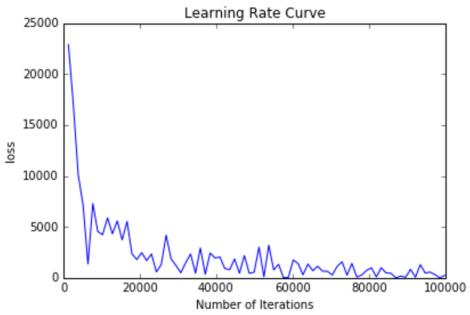
For our second CNN model, we achieved an accuracy of 96.94% and F_1 score of 0.9691 on the validation set, which is an improvement from our benchmark accuracy. The training time for 100,000 iterations of stochastic

[9]The display_step determines how often we report the accuracy and cost of the model. With a display step of 10, we display the accuracy and cost after every $128 * 10 = 1280$ iterations.

[10]There are 28*28 pixels in each picture, and 4 map features (red, green, blue, near-infared). Therefore, we have $28 * 28 * 4 = 3136$ inputs

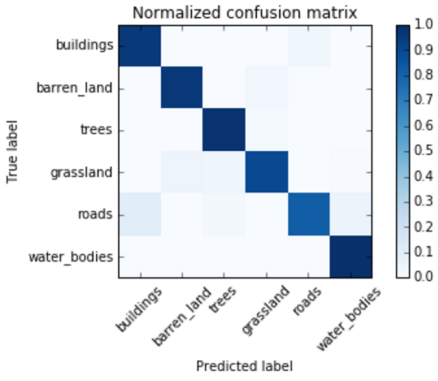
gradient descent was 475.54 seconds and prediction time on the validation dataset was 7.67 seconds. The figure below shows the value of our loss for each iteration at each display step.

Figure 8: Learning rate curve for cnn_2



Judging by our figure, it seems that we reach an optimal minimum around 90,000 iterations.
Since our accuracy and F_1 scores are comparable, we can roughly conclude that our prediction is uniformly accurate for different land cover classes. We can verify this by looking at our normalized confusion matrix:

Figure 9: Normalized Confusion Matrix for cnn_2



The matrix shows that this algorithm performed the worst for the roads land cover class similar to the first model. However, by visual inspection it looks that our second model performed better for roads than our first model.
Next, let's add 1x1 convolutions to improve the performance of our model. The convolutions that we currently are using essentially run a linear classifier on a small patch of an image. We can add a 1x1 convolution to each of these

patches, which increases the depth of our model without adding too many extra parameters [11].
Judging by our learning curve, it seemed that we didn't need 100,000 iterations, since our loss flatlined. However, since we are adding more parameters to learn because of our 1x1 convolutions, let's increase the number of iterations to 150,000 and increase our learning rate to 0.005. We can also increase our batch size to 256 in order to reduce the noise when training our data on our smaller batch of examples.
Our final hyperparameters for the third model are:

- learning_rate = 0.005
- training_iters = 150,000
- batch_size = 256
- display _step = 10 [12]
- n_input = 3136 [13]
- n_classes = 6
- dropout = 0.5
- training _set = 20,000
- validation _set = 5,000
- 1x1 convolutions following both convolutions.

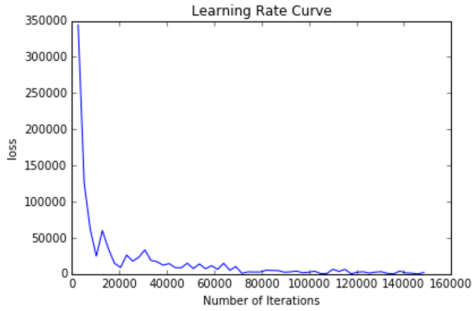
For our third CNN model, we achieved an accuracy of 97.10% and F_1 score of 0.9705 on the validation set, which is slightly better than our second model. The training time for 150,000 iterations of stochastic gradient descent was 529.34 seconds and prediction time on the validation dataset was 7.40 seconds. The figure below shows the value of our loss for each iteration at each display step.
Judging by our figure, it seems that we reach an optimal minimum around 150,000 iterations, so our selection was appropriate.

[11] https://www.udacity.com/course/deep-learning–ud730

[12] The display_step determines how often we report the accuracy and cost of the model. With a display step of 10, we display the accuracy and cost after every $128 * 10 = 1280$ iterations.

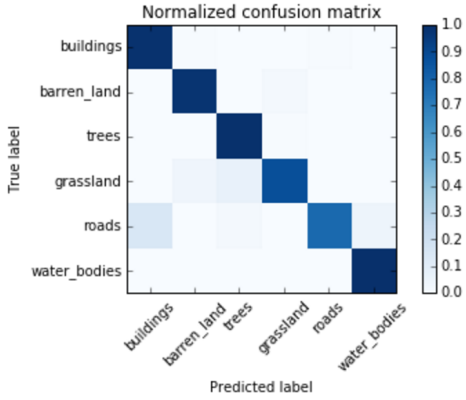
[13] There are 28*28 pixels in each picture, and 4 map features (red, green, blue, near-infared). Therefore, we have $28 * 28 * 4 = 3136$ inputs

Figure 10: Learning rate curve for cnn_2



Since our accuracy and F_1 scores are comparable, we can roughly conclude that our prediction is uniformly accurate for different land cover classes. We can verify this by looking at our normalized confusion matrix:

Figure 11: Normalized Confusion Matrix for cnn_2



The matrix shows that this algorithm performed the worst for the roads land cover class similar to the first model and performed poorly when predicting examples that were labeled with grassland land cover.
Ideally, we would want to a gridsearch over a combination of different hyperparameters. Since we are limited by the long runtimes of the CNN, we opt to only try these three models.

4.3 Conclusion

After testing a total of three different models, the third model performed the best. The table below shows a comparison of our three models.

Table 1 - Comparison Table for the Three Models

	Accuracy	F_1 Score	Training Time (s)	Prediction Time (s)
cnn_ 1	96.90%	0.9689	244	8.89
cnn_ 2	96.94%	0.9691	475.54	7.67
cnn_ 3	97.10%	0.9705	529.34	7.40

Since cnn_ 3 performed the best on our validation set, we can use the model's parameters on our test set. Cnn_ 3 scored an accuracy of 97.66% and a F_1 score of 0.9766 on the test set.
By increasing dropout from 0.75 to 0.5, we increased regularization which allowed the model to iterate over several more epochs and reach a local minimum without overfitting. We were also able to increase the complexity of the model without sacrificing performance by adding 1x1 convolutions to each convolutional layer. The model accurately predicted water_ bodies the most accurately, probably because there are the most water _ bodies examples in our dataset and they look the most different (blue color v. green/brown/grey colors). Since we didn't see as many road and building examples, our model may not be robust if we used a test set of pictures from an urban area.

The project could be expanded in a few different ways. First, we could test several hyperparameters and parameters using a validation set. We could try different architectures, mixing and matching layers to see which one gave us the best result. We could also change some of the parameters. For example, instead of using a stride size of 1 we could use a stride size of 2 or increase the depth of the output from one of our convolutional layers. We could try using less iterations to prevent overfitting, since it seemed our CNN model converged to an optimal cost around iteration 70,000. Finally, we could use the entire dataset. CNNs work well when we use lots of data, so using the entire dataset could increase the accuracy of our model.

In summary, our initial problem was to accurately classify satellite images into 6 different land types. We decided to use a convolutional neural network since we had a large amount of images and features ($28 * 28 * 4 = 3136$ features per example). Our initial cnn had 2 convolutional layers and 1 fully connected layer, which used stochastic gradient descent, an adam optimizer which used both momentum and learning rate decay, and dropout for regularization.

Our first model produced an accuracy of 96.90% and F_1 score of 0.9689 on the validation set. We decided to increase regularization by decreasing our dropout value and increase iterations of our stochastic gradient descent. Our second model produced an accuracy of 97.06% and F_1 score of 0.97013 on the validation set.

For our third model, we introduced 1x1 convolutions that followed max-pooling for each convolutional layer and increased our batch size. Our third model produced an accuracy of 97.1% and F_1 score of 0.9705 on the validation set.
This project provided me with a great introduction to convolutional neural networks. I was impressed with the power of the algorithm, given we did not perform most of the preprocessing steps that other comparable computer vision classifiers require. Unfortunately, I felt that I was memory requirements of the algorithm and had to severly limit the amount of training examples I used.