Mark Barna
Machine Learning 2
Final Project Individual Report

1. Introduction. An overview of the project and an outline of the shared work.

After discussing several different dataset options to use for this project, we settled on the DeepSat (SAT-6) Airborne Dataset sourced from Kaggle. This dataset contains 405,000 28✕28-pixel satellite images of earth terrain split 80/20 between training and testing sets. The images were sourced from the National Agriculture Imagery Program (NAIP), which has a total of 330,000 images of 6000✕7000 pixels, covering the entire continental United States. The original dataset is over 65 terabytes in size so the SAT-6 authors sampled 1,500 images of California from the NAIP dataset to create their 28✕28 image segments, which were then manually one-hot labeled into six classes: barren land, trees, grassland, roads, buildings, and water. The images have four-channels: red, green, blue, and near-infrared (NIR) (Figure 1).

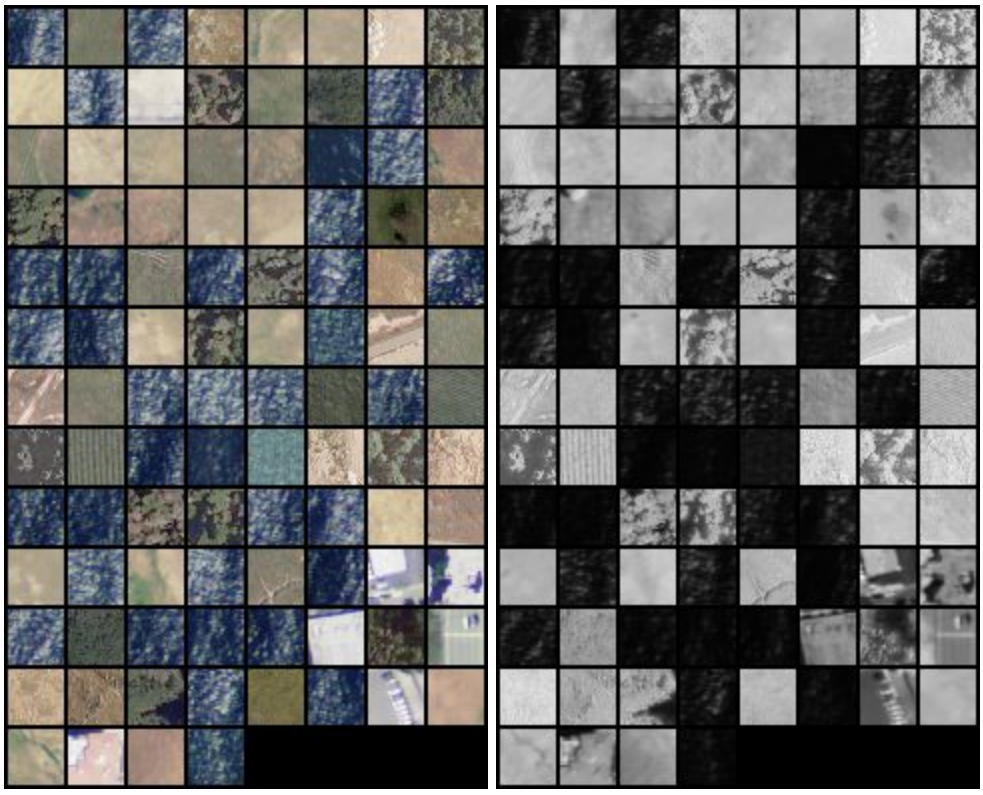


Figure 1: RGB channels (left), NIR channel (right)

Our project's goal was to train a neural network to classify the images into the six terrain types. Satellite imagery is employed across many industries including agriculture, infrastructure, disaster recovery, intelligence, defense, and others. The sheer quantity of data is hard for humans

to analyze efficiently so the ability for a computer algorithm to learn terrain classification could be extremely useful.

We divided up the work as follows:

- Data quality checking/cleaning: Thanh
- Review background material and other papers: All
- Write dataset loading, basic network architecture, and training code in PyTorch: Mark
- Experimenting with different network architectures in PyTorch: All
- Code to plot kernels and feature maps: Thanh
- Review training and testing results: All
- Group paper and presentation sections 1, 2 and 6: Hailey
- Group paper and presentation section 3: Mark
- Group paper and presentation sections 4 and 5: Thanh
- References list, finalize submission materials: All

2. Description of your individual work. Provide some background information on the development of the algorithm and include necessary equations and figures.

Since I enjoy coding, I volunteered to write the basic python code to load the dataset, train the network on a basic architecture, and output some plots of the training statistics. We then each separately adapted the code to create additional network architectures to test. I will describe my coding work in section 3.

Network Architecture

From the network development standpoint, we decided to start with the simplest possible convolutional neural network (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 Figure 2).

Simple Convolutional Network

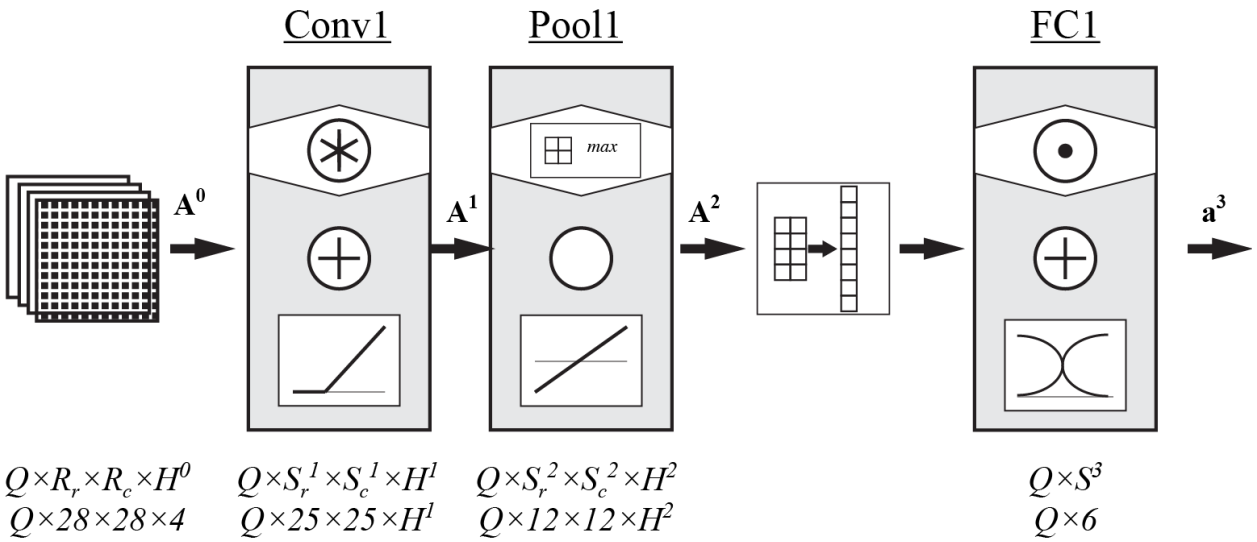


Figure 2

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.

[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.

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 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 Figure 3 below for comparison of architectures).

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):

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

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. As discussed further in section 4, we settled on a size of 100 for most of the training runs. The weight and bias update rules (using mini-batches of size Q) after completing forward pass k are (Demuth et al. 2014):

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

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

We initialized the network with a learning rate of $\alpha = 0.001$ and later increased to $\alpha = 0.005$. We also incorporated momentum into the update rule for certain trials. 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):

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

where $\dot{F}^m(N^m)$ 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.

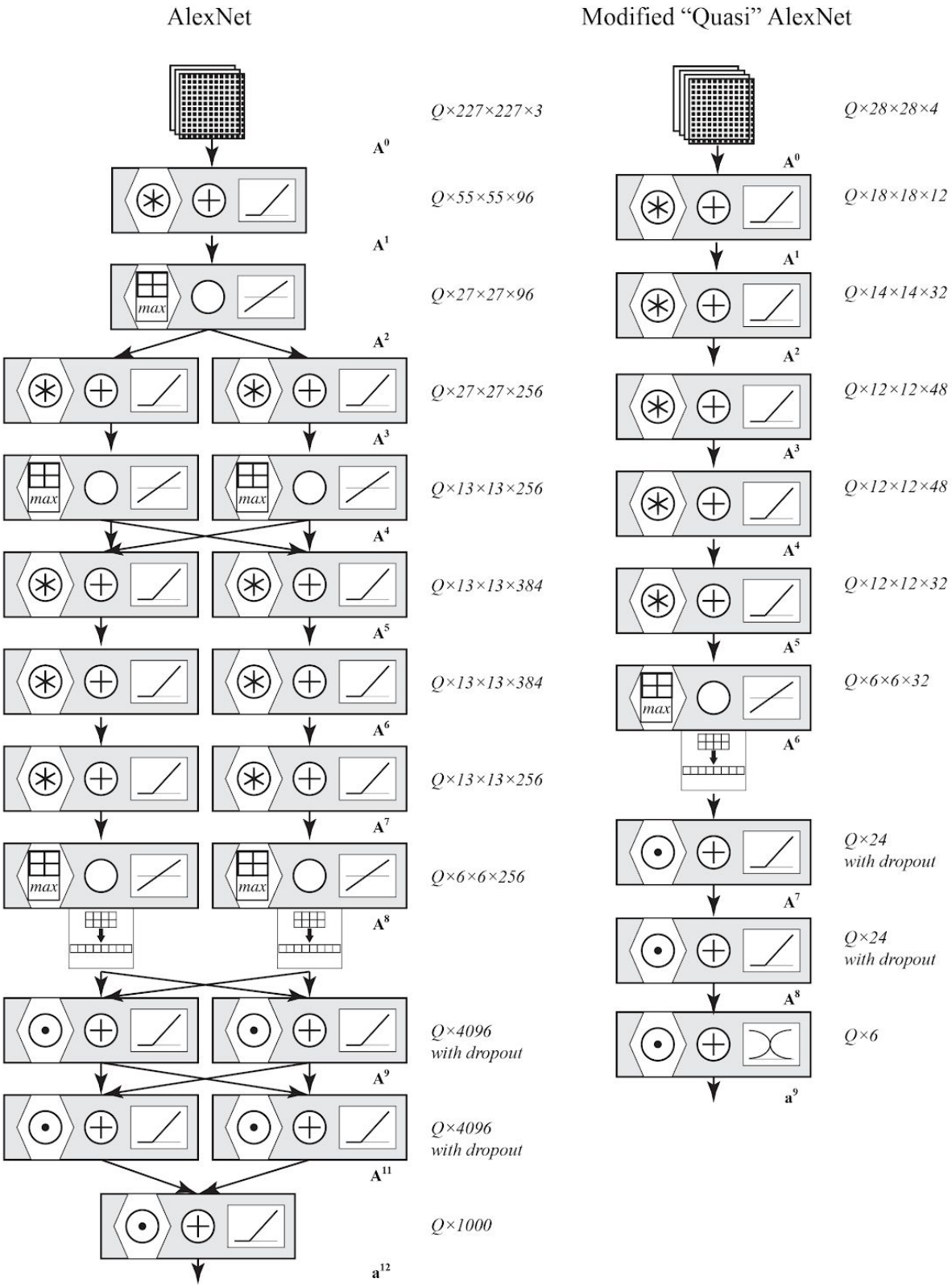

AlexNet
Modified "Quasi" AlexNet
Q×227×227×3
A0
Q×55×55×96
A1
Q×27×27×96
A2
Q×27×27×256
A3
Q×13×13×256
A4
Q×13×13×384
A5
Q×13×13×384
A6
Q×13×13×256
A7
Q×6×6×256
A8
Q×4096
with dropout
A9
Q×4096
with dropout
A11
Q×1000
a12
max
Q×28×28×4
A0
Q×18×18×12
A1
Q×14×14×32
A2
Q×12×12×48
A3
Q×12×12×48
A4
Q×12×12×32
A5
Q×6×6×32
A6
Q×24
with dropout
A7
Q×24
with dropout
A8
Q×6
a9

Figure 3

3. Describe the portion of the work that you did on the project in detail. It can be figures, codes, explanation, preprocessing, training, etc.

In addition to crafting the network architecture description above, I volunteered to code the dataset loading, network training, and some of the metrics plots. Below are the notable code excerpts.

The PyTorch documentation recommends writing a custom dataset class that is passed to the built-in `Dataloader` function. This allows for use of the batching and shuffling methods. Since our dataset comes as separate csv files for the images and labels, this custom class loads one row at a time and reshapes the images into three dimensions from row-wise entries. The code also converts the labels from one-hot encoding to single values representing the class numbers, as is required by the *cross-entropy* function in PyTorch.

```
# define dataset class
class DeepSAT6(Dataset):
    '''DeepSat6 Dataset'''

    def __init__(self, img_file, label_file, transform=None):
        self.img = pd.read_csv(img_file, header=None)
        self.label = pd.read_csv(label_file, header=None)
        self.transform = transform

    # built-in length
    def __len__(self):
        return len(self.img)

    # fetch a row from the csv files for the image and label
   # reshape the image row to numpy image format (image has 4 channels: RGB and near-infrared)
    # if a transform is passed, apply it and return the sample
    def __getitem__(self, index):
        img = self.img.iloc[index,:]
        img = img.values.reshape((input_size, input_size, 4))
        label = self.label.iloc[index,:]
        # convert labels from one-hot encoding to class number, by returning index of 1-value
        label = label.idxmax()
        label = int(label)
        sample = img, label

        if self.transform:
            sample = self.transform(sample)

        return sample
```

I also wrote a custom `transform` class that converts the images from Numpy arrays to Torch tensors and rescales the images from [0,255] to [0,1].

```
# define tensor transformation class
class ToTensor(object):
    '''Converts data in numpy arrays to tensor and reshapes images to CxHxW order'''

    def __call__(self, sample):
        img, label = sample

        # reverse order of axes from numpy image to tensor image format
        # normalize into the range [0,1]
        img = img.transpose((2,0,1))
        #img = img.astype(float)
        img = (img + 1)/256
        # convert image to Tensor, leave label as int.
        # DataLoader will convert label to Tensor of size=batch_size
        return torch.FloatTensor(img), label
```

As described in section 2, I wrote a basic network class and the modified AlexNet class (I called it "QuasiAlexNet") for training.

```
class SimpleCNN(nn.Module):
    '''Basic CNN Neural network'''

    def __init__(self):
        super(SimpleCNN, self).__init__()
                        self.conv1   =   nn.Conv2d(in_channels=input_chan,   out_channels=16,
kernel_size=kernel_size)
        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.fc1 = nn.Linear(in_features=12 * 12 * 16, out_features=num_classes)

    def forward(self, p):
        a = f.relu(self.conv1(p))
        a = self.pool(a)
        a = a.view(-1, 12 * 12 * 16)
        a = self.fc1(a)
        return a

class QuasiAlex(nn.Module):
    '''A simplified version of AlexNet'''

    def __init__(self):
        super(QuasiAlex, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=input_chan, out_channels=12, kernel_size=11)
        self.conv2 = nn.Conv2d(in_channels=12, out_channels=32, kernel_size=5)
        self.conv3 = nn.Conv2d(in_channels=32, out_channels=48, kernel_size=3)
```

```
        self.conv4 = nn.Conv2d(in_channels=48, out_channels=48, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(in_channels=48, out_channels=32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(in_features=6 * 6 * 32, out_features=24)
        self.fc2 = nn.Linear(in_features=24, out_features=24)
        self.fc3 = nn.Linear(in_features=24, out_features=6)

    def forward(self, p):
        a = f.relu(self.conv1(p))
        a = f.relu(self.conv2(a))
        a = f.relu(self.conv3(a))
        a = f.relu(self.conv4(a))
        a = f.relu(self.conv5(a))
        a = self.pool(a)
        a = a.view(-1, 6 * 6 * 32)
        a = f.dropout(f.relu(self.fc1(a)))
        a = f.dropout(f.relu(self.fc2(a)))
        a = self.fc3(a)
        return a
```

I then wrote code to define the loss and optimizer functions and train and test the network, as adapted from the PyTorch documentation. I included accuracy tests for both the training and test set.

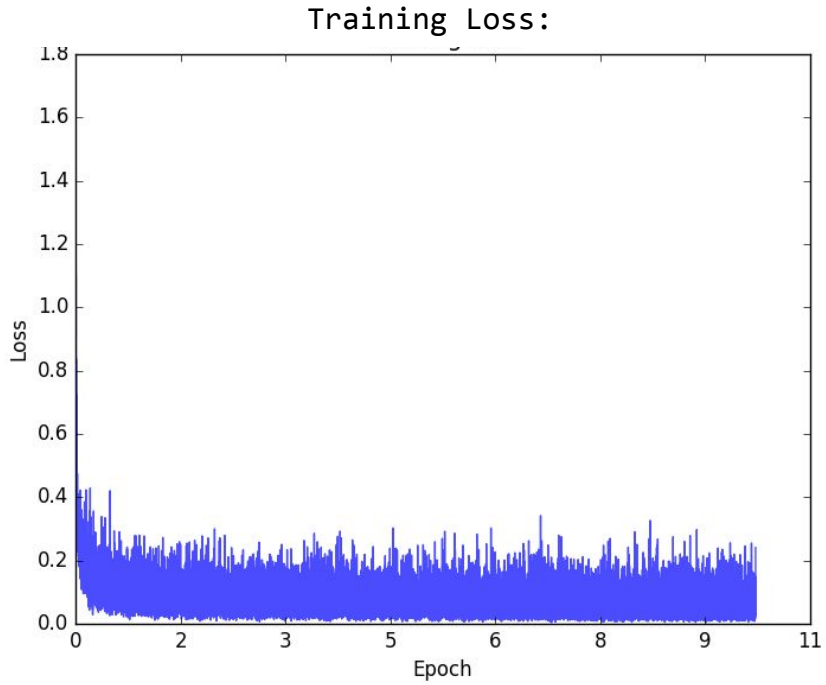
```
# define loss and optimiser
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(network.parameters(), lr=learning_rate, momentum=momentum)

############################
# TRAIN NETWORK
############################

# track the training loss at each iteration and training & testing accuracy at each epoch
running_loss = np.zeros((epochs, len(train_loader)))
running_train_acc = np.zeros((epochs))
running_test_acc = np.zeros((epochs))
# set-up confusion matrix of training accuracy
confusion_matrix = np.zeros((num_classes, num_classes), dtype=int)

def train():

    for epoch in range(epochs):

        # rest running totals of correct guesses
        train_correct = 0
        test_correct = 0

        for i, data in enumerate(train_loader):
```

```
        # get data and convert to Variable
        images, labels = data

        # zero gradient buffer
        optimizer.zero_grad()

        # forward pass
        output = network(Variable(images.cuda()))
        # calculate loss
        error = loss(output, Variable(labels.cuda()))
        # backprop
        error.backward()
        # optimization step
        optimizer.step()

        # get network prediction (neuron index with largest output)
        train_predict = torch.max(output.data, 1)[1]

        # compare each prediction in batch to label and increment test_correct tally
        train_correct += (train_predict.cpu() == labels).sum()

        # accuracy at each epoch: add to tracking
        running_train_acc[epoch] = train_correct / len(train_set) * 100

        # show training loss at every 100 data points
        if (i + 1) % 100 == 0:
            print('Training loss at epoch {} of {}, step {} of {}: {:.4f}'.format(
            epoch + 1, epochs, (i + 1), len(train_loader), error.data[0]
            ))

        # add to running loss
        running_loss[epoch, i] = error.data[0]


    # run testing set through network at each epoch
    for data in test_loader:

        # get data and convert images to Variable
        images, labels = data

        # calculate output
        output = network(Variable(images.cuda()))

        # get network prediction (neuron index with largest output)
        test_predict = torch.max(output.data, 1)[1]

        # compare each prediction in batch to label and increment test_correct tally
        test_correct += (test_predict.cpu() == labels).sum()

        # last epoch only
```

```
            # for each item in the batch, increment by one the corresponding row and column
            if epoch == epochs - 1:
                for i in range(batch_size):
                    confusion_matrix[test_predict[i], labels[i]] += 1

        # accuracy at each epoch: add to tracking, print
        running_test_acc[epoch] = test_correct / len(test_set) * 100
```

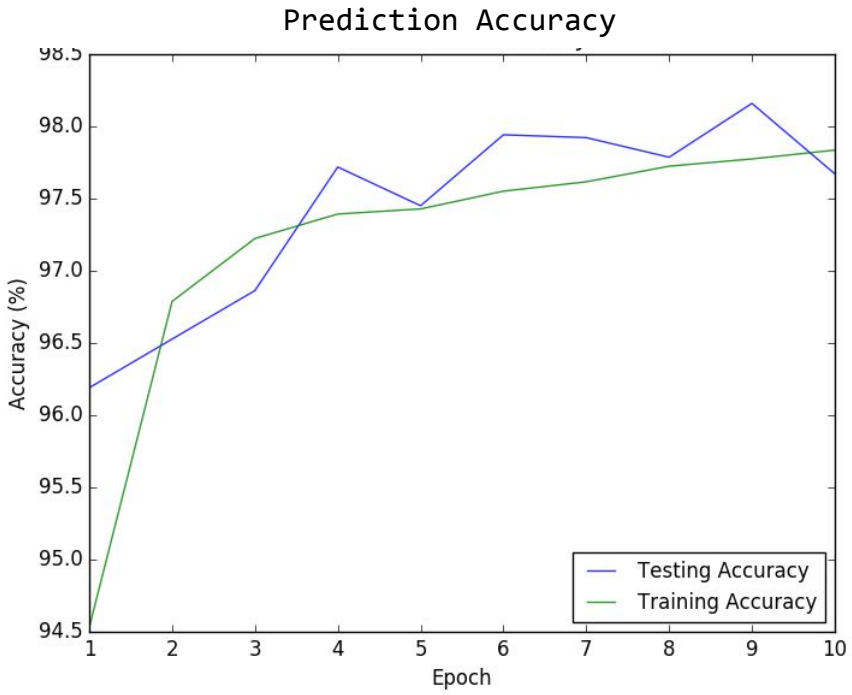
Finally, in addition to printing running statistics of the training to the console, I wrote code to output plots of the running loss, running accuracy, and a confusion matrix of the final results.

```
running_loss = running_loss.reshape(-1)

# plot running loss
fig, ax = plt.subplots(ncols=1, nrows=1)
ax.plot(running_loss, alpha=0.7)
# label x axis with epochs using formatter
ax.set_xticklabels(['{:0.0f}'.format(i)      for      i      in      ax.get_xticks()      /
(len(train_set)/batch_size)])
ax.set_title('Trianing Loss')
ax.set_ylabel('Loss')
ax.set_xlabel('Epoch')
fig.savefig('../Plots/training_loss.png')
print('Training loss plot saved in /Plots.')
```



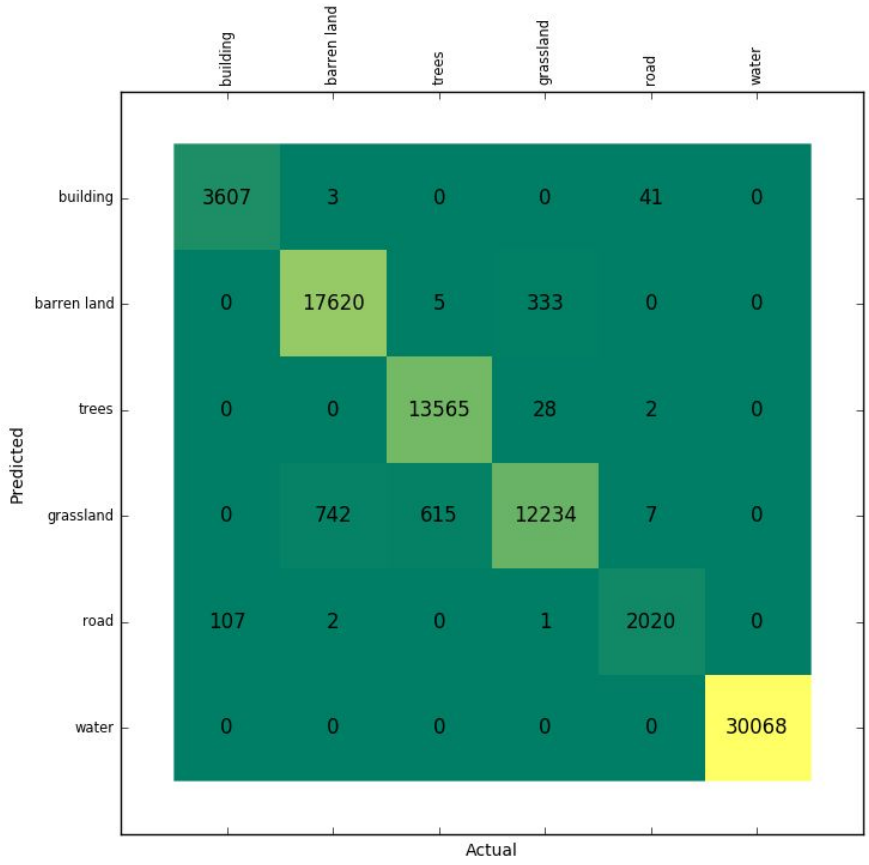

```
# plot comparison of testing and training accuracy
fig, ax = plt.subplots(ncols=1, nrows=1)
ax.plot(running_test_acc, label='Testing Accuracy', alpha=0.8)
```

```
ax.plot(running_train_acc, label='Training Accuracy', alpha=0.8)
#ax.set_ylim(0, 100)
ax.set_xticklabels('{:0.0f}'.format(i+1) for i in ax.get_xticks())
ax.set_title('Prediction Accuracy')
ax.set_ylabel('Accuracy (%)')
ax.set_xlabel('Epoch')
ax.legend(loc='lower right', fontsize='medium')
fig.savefig('../Plots/testing_accuracy.png')
print('Testing Accuracy plot saved in /Plots.')
```




```
# confusion matrix of predictions
# convert to dataframe and add column, index names
confusion_matrix_df = pd.DataFrame(confusion_matrix, index=classes, columns=classes)
print(confusion_matrix_df)

# calculate accuracy and misclassification rate
accuracy = confusion_matrix.diagonal().sum() / confusion_matrix.sum() * 100
wrong = 100 - accuracy
print('Accuracy rate: {:.2f}%\nMisclassifcation rate: {:.2f}%'.format(accuracy, wrong))

# plot confusion confusion
fig, ax = plt.subplots(ncols=1, nrows=1, figsize=[8,8])
ax.matshow(confusion_matrix, cmap='summer', vmin=confusion_matrix.min(),
           vmax=confusion_matrix.max())
ax.set_xlim(-1, 6)
ax.set_ylim(6, -1) # reverse order
ax.set_xticklabels(classes, rotation=90, size='x-small')
```

```
ax.set_xticks([i for i in range(len(classes))])
ax.set_yticklabels(classes, size='x-small')
ax.set_yticks([i for i in range(len(classes))])
ax.set_xlabel('Actual', size='small')
ax.set_ylabel('Predicted', size='small')

# label cells
for i in range(len(classes)):
    for j in range(len(classes)):
        ax.text(i, j, confusion_matrix[j, i], ha='center', va='center', color='black')

fig.savefig('../Plots/confusion_matrix.png')
print('Confusion matrix plot saved in /Plots.')
```



	building	barren land	trees	grassland	road	water
building	3607	3	0	0	41	0
barren land	0	17620	5	333	0	0
trees	0	0	13565	28	2	0
grassland	0	742	615	12234	7	0
road	107	2	0	1	2020	0
water	0	0	0	0	0	30068

```
Accuracy rate: 97.67%
Misclassification rate: 2.33%
```

4. Results. Describe the results of your experiments, using figures and tables wherever possible. Include all results (including all figures and tables) in the main body of the report, not in appendices. Provide an explanation of each figure and table that you include. Your discussions in this section will be the most important part of the report.

Simple CNN Architecture Trials

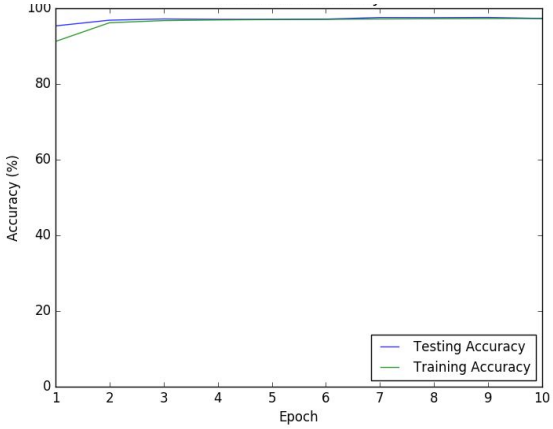
I started with the simplest possible CNN architecture to establish a baseline. Beginning with 10 epochs, at a learning rate of 0.001, I set the convolutional layer to 8 kernels (twice the number of channels in the image and a power of 2).

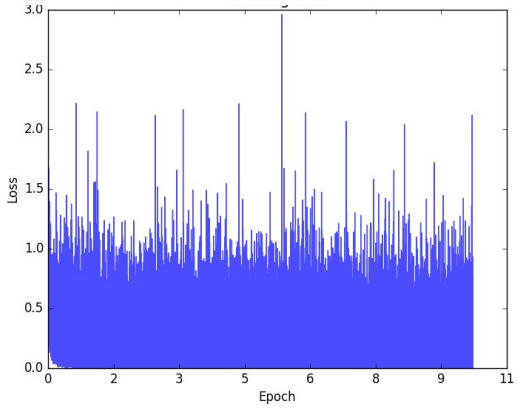
Batch Size	Epochs	Learning Rate	Momentum	# Conv Kernels	Training Time	Accuracy (%)
10	10	0.001	N/A	8	00:55:47.5	97.30%
100	10	0.001	N/A	8	00:47:10.2	95.34%
1000	10	0.001	N/A	8	00:37:50.8	87.15%
100	10	0.001	N/A	4	00:46:05.9	93.00%
100	10	0.001	N/A	16	00:45:44.9	96.71%
100	15	0.001	N/A	16	01:09:21.5	96.84%
100	20	0.001	N/A	32	01:33:14.8	97.28%
100	10	0.005	N/A	16	00:47:58.8s	97.30%
100	10	0.005	0.8	16	00:47:24.3	97.27%`
100	10	0.005	0.9	16	00:49:13.5	97.67%

Table 1

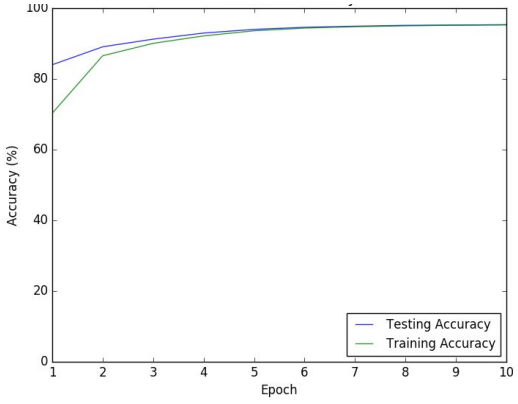
Initially, I wanted to determine the best batch size to use so, holding all other parameters constant, I tried batches of 10, 100, and 1000 (rows 1-3 of Table 1). As we can see in the comparisons of accuracy in Figure 4 (left panels), the batch size of 10 had the highest accuracy, but took 8.5 minutes longer to train than the batch size of 100. The batch size of 1000 took almost 9 minutes less time, but lost accuracy. We can also see that the training and testing curves are close together, indicating that overfitting is not an issue (Li et al. 2017).

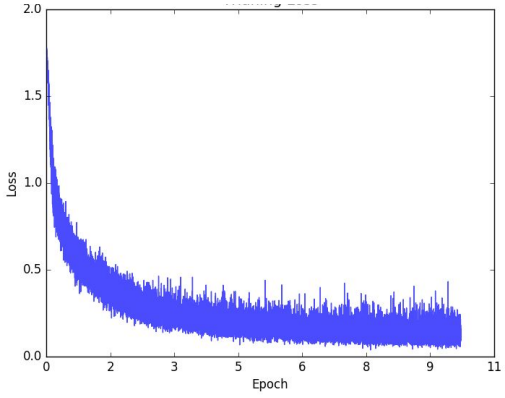
We can also see in the right-hand panels of Figure 4 that as the batch size increases, the oscillation of the training loss decreases since the algorithm has more information to use in adjusting the step direction after each update.



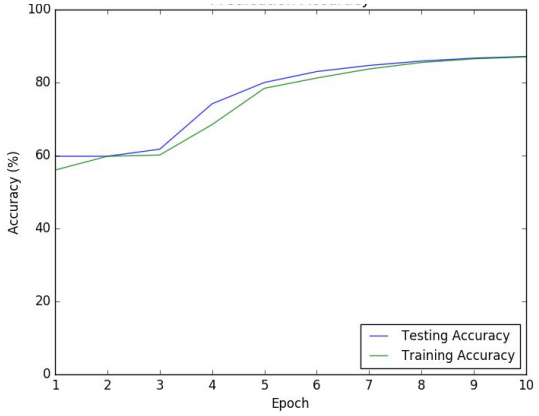


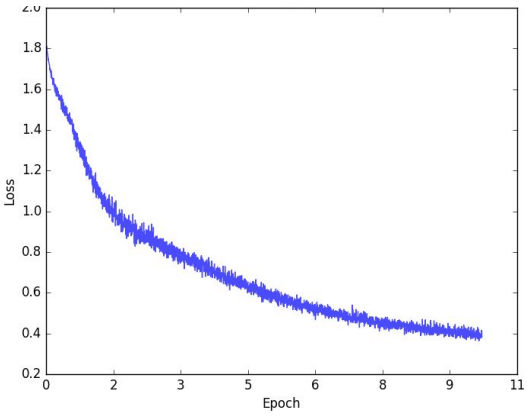
Batch size = 10





Batch size = 100





Batch size = 1000

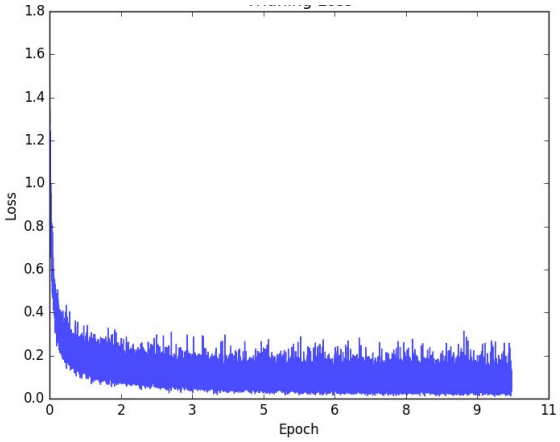
Figure 4

Since the batch size of 1000 suffered from poor accuracy relative to the others, I discarded this as an option for the next set of trials. While the batch size of 10 bested the size of 100 by two percentage points, it was slower.

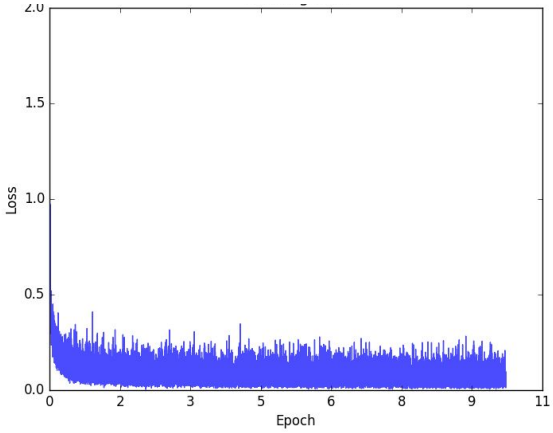
I wanted to see if I could improve the accuracy of the batch size of 100 by increasing the number of kernels in the convolutional layer. Rows 4-7 show these trials. I also noted that the network error was still decreasing at the end of 10 epochs so I extended the number to 15 and then 20 in rows 6-7. Of these trials, the network with 32 convolution kernels achieved the best accuracy, but took 20 epochs so the training time spiked to 1 hour and 33 minutes.

In order to reduce training time, I decreased the convolutional layer to 16 kernels and increased the learning rate to 0.005. This brought the accuracy up to 97.3% (row 8), the exact same level as in the test with batch size of 10, but nearly 8 minutes faster.

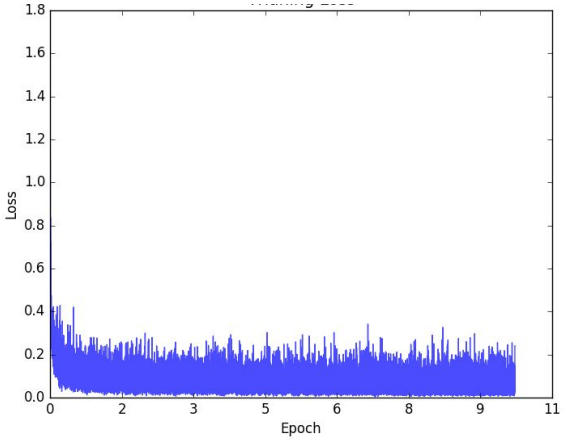
Finally, I added momentum into the gradient descent algorithm to attempt a further reduction in training loss oscillation (Figure 5). From these comparisons, it is not evident that adding momentum reduced the oscillations, however it did cause the loss to reduce faster in the early steps.



No momentum



Momentum = 0.8



Momentum = 0.9
Figure 5

Overall, the final trial achieved the best accuracy. I was pleasantly surprised that such a simple network was able to perform this well. Figure 6 shows a confusion matrix of the predictions from this trial. The network made the most errors in differentiating between grassland, trees, and barren land. It showed some bias toward grassland. It was also a bit biased in favor of roads over buildings.

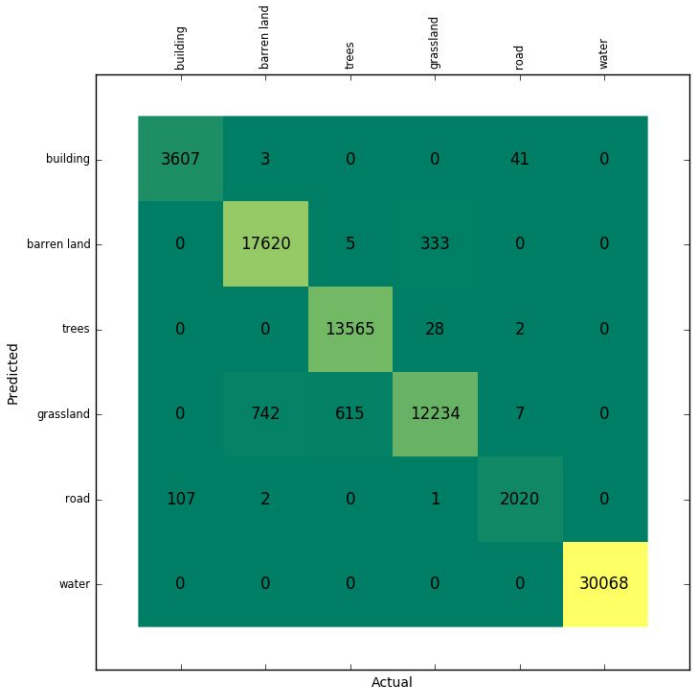


Figure 6

"QuasiAlexNet" Architecture Trials

After completing the trials above with a simple CNN, I implemented a more complex network to see if it was possible to improve performance even further. As discussed in section 2, I created a variation on the AlexNet architecture, using the same learning rate, momentum and weight decay as in the original paper. Table 2 shows the best attempts, which are similar to those achieved with the simple CNN, but because this network is more complicated, it takes longer to train.

Batch Size	Epochs	Learning Rate	Momentum	Weight Decay	Training Time	Accuracy (%)
100	10	0.01	0.9	0.0005	00:54:03.2	97.18%
100	10	0.005	0.95	0.0005	00:55:50.1	96.76%

Table 2

5. Summary and conclusions. Summarize the results you obtained, explain what you have learned, and suggest improvements that could be made in the future.

I did not expect the basic convolutional network to achieve the results that it did, peaking at an accuracy of 97.67%. My hypothesis for this is that the images were very small at 28✕28 so there was not a lot of information in each one that the network had to interpret. In addition, the images of roads and buildings were generally the only samples with many edges and complex shapes. The other pictures tended to have uniform patterns with distinct colors between classes. In the future, I would use larger, more complex images to test my theory.

With more time, I would also finetune the decision thresholds among certain classes to reduce bias in the network. ROC curve plots could assist by showing how adjusting the classification boundaries would influence the error.

Over the course of this assignment, I learned a lot about working with PyTorch, which I enjoyed. The package is very-well documented. Next time, I would write additional code in the QuasiAlexNet class to allow the user to specify the sizes of all of the convolution layers upfront rather than hard-coding them into the class. I would also break the custom classes into their own modules that are contained in separate files in order to keep the code files neater. Finally, while not the goal of this assignment, if given more time, I would improve some aspects of the plots, such as ensuring the information fills the entire figure area.

6. Calculate the percentage of the code that you found or copied from the internet. For example, if you used 50 lines of code from the internet and then you modified 10 of lines and added another 15 lines of your own code, the percentage will be $\frac{50-10}{50+15} \times 100$.

Lines of code from internet sources: 59

Lines modified: 48
Lines original: 133

$$\frac{59-48}{59+133} \times 100 = 5.73\%$$

7. References.

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).

Adapted Code

"Data Loading and Processing Tutorial." PyTorch tutorial. http://pytorch.org/tutorials/beginner/data_loading_tutorial.html (accessed April 2018)

"How can I plot a confusion matrix?" https://stackoverflow.com/questions/35572000/how-can-i-plot-a-confusion-matrix?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa (accessed April 2018).

"Training a Classifier." PyTorch tutorial. http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (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).