Convolutional Neural Networks

Project 1: CMPEN 454 (Computer Vision)

Christopher Carney, Connie Couey, Christian Myers, Darshan Shah

Table of Contents

1. Introduction

The purpose of this project was to introduce groups to image processing in MATLAB and apply computer vision concepts learned in class to create a program that is relatively simple when broken down into its individual parts, but does something meaningful when put together. This program, a convolutional neural network (CNN), allowed our group to familiarize ourselves with the basics of MATLAB (arrays, graphing, image processing tips & tricks). Saving intermediate steps of the CNN output layers also allowed us to see how the filters and other functions change the pixel values and we can begin to see the different patterns and patches that emerge. Furthermore, it gave our group a simple real world application of computer vision. This included knowledge gained from experience of the underlying convolution filters learned from class and homework, plus coding examples given from both this and other classes.

Some of the tasks we performed included interpreting the different parts of each algorithm and implementing it in MATLAB, creating several functions to call the layers in their respective order, and to create several "top-level" functions, whether it be to debug our output, categorize all ten-thousand images, or categorize a custom image. After playing around with different inputs on our CNN we analyzed different layer outputs and used performance metrics like a confusion matrix and the image itself to verify all layers are working properly and also to hypothesize *why* the CNN was doing what it was doing for each step and how it reached its eventual classification. Ultimately, we expected to be able to sort the images into their correct class with some degree of accuracy.

2. Procedural Approaches

Below we document the different design decisions we made while implementing our project in MATLAB. Additionally, we also try to provide a basic outline to how our code works and why we programmed it that way.

2.1 Outline

For our program, the `Main` function that loads the images from the `cifar10testdata` and then calls another function that calls each individual layer function. The pseudocode shows the control of the functions, what each does, and where they are called.

2.1.1 Main()

```
Load the cifar10 images

Initialize the confusion matrix (10x10) called out

For the class index 1 to 10
       Find the true class

       For the images of that class from 1 to 1000
              Load image(im) into imrgb
              Call ConvNeuralNet with param imrgb and store result as out
              Find max value of probability vector
              Place in confusion matrix based at spot (true class, highest prob)
       End for loop
End for loop
```

In the main function, we compute the confusion matrix which we will present below. We do this be creating a blank 10x10 matrix and incrementing the values in the matrix cells according to their `(trueclass, probability-class)`. Chris, who is a computer science major, wrote a lot of these subroutines and it is difficult for him to not use the C-style thinking, loops, and straightforward approach to implementing algorithms, so that will be a recurring theme in the code.

2.1.2 ConvNeuralNet(input_im)

```
Load bias vectors and filter banks

Create matrix called open to store outputs of intermittent steps
Change input_im to double and store in variable im

Call ImageNormilization() and store result in out{1,1}
Call Convolution() and store result in out{1,2}
Call reLU() and store result in out{1,3}
Call Convolution() and store result in out{1,4}
Call reLU and store result in out{1,5}
Call Maxpool and store result in out{1,6}
Call Convolution() and store result in out {1,7}
Call reLU and store result in out{1,8}
Call Convolution() and store result in out{1,9}
Call reLU() and store result in out{1,10}
Call Maxpool() and store result in out{1,11}
Call Convolution() and store in out{1,12}
Call reLU() and store in out{1,13}
Call Convolution() and store in out{1,14}
Call reLU() and store in out{1,15}
Call Maxpool(out{1,15,biasvectors{15},filterbanks{15});store out{1,16}
Call fullyConnected( out{1,16},biasvectors{16},filterbank{16});store out{1,17}
Call softMax(out{1,17},biasvectors{17},filterbank{17}) and store in out{1,18}
```

Our CNN takes a single image and then returns the resulting layers in a `cell(1,18)` table. This was an important design decision because we can call this subroutine from any of our Main, debugging, or custom testing functions and since it returns all layers, we can easily compare these results to the `debuggingTest.mat` to ensure correctness, and also we can get intermediate layers from images we process.

2.1.3. ImageNormalization(im)

```
Divide input im by 255.0 - 0.5 and store in variable out
Return out back to ConvNeuralNet
```

This was taken from the project description, which approximately scales each color channel's pixel values into the output range -0.5 to 0.5.

2.1.4 Convolution(im,biasvector,filterbank)

```
Set variable (s) to size of im
Create array (out)
Set variable (sf) to the size of the filterbank
initialize variable (conv)

For i = 1 to sf(4)
        Set variable (filt) equal to filter i from filter bank
       Create matrix (tempConv) to store intermediate values for summing

       For j = 1 to sf(3)
              Imfilter('conv') each layer with its layer filter
       End for loop

       For k = 1 to sf(3)
              Sum all intermediate results to form 1 layer and store in conv
       End for loop
       Add bias vector value and store in conv
       Store conv in out's 3rd dimension
End for loop
End Convolution
Return to ConvNeuralNet with variable out
```

Because the convolution used for the CNN was not as straightforward as using just `imfilter` and returning the output, we broke down the problem into small sections which were described in the algorithm presented in the project description. We do a lot of storing intermediate output and processing through loops. Although this is definitely *not* the most efficient means of implementing this it is very readable and easy to understand while comparing the description of the design document to the implementation.

2.1.5 ReLU(im)

```
Initialize variable (ind) to 1
Set variable (s) to the size of im

For i = 1 to the length of s
       ind is equal to ind * size of i
End for loop

For k = 1 to ind
       If the pixel at im(k) is less than 0
              Set im(k) equal to 0
       End if
End for loop
Set im equal to out

End reLU
Return out to ConvNeuralNet
```

The basic execution of the `ReLU` function (otherwise known as Rectified Linear Unit) is to have the input equal the output (array size), with any negative number set to 0 in the output. Otherwise, the input and the output are exactly the same besides this "minor" detail. For this layer, we implemented for loops in order to find the negative numbers in the array and then set them equal to zero. The reason why for loops were implemented, was because we wanted to be able to see where any number was negative, and then set that number to zero. Otherwise, we would set that number equal to what it was in the input (multiplying that number by 1 in a separate loop, so that any non-negative number stayed the same).

2.1.6 Maxpool(im)

```
Set variable (s) to the size of the image
Create matrix (out)
For i = 1 to s(3)
       Set variable curY to 1
              For j = 1 to 2 to s(2)
                     Set variable curX to 1
                     For k = 1 to 2 to s(1)
                            Temp1: max of pixels im(j,k,i) and im(j,k+1,i)
                            Temp2: max of pixels im(j+1,k.i) and im(j+1,k+1,i)
                            Matrix out(curX,curY,i) set to max of t1 and t2
                            Increment curX
                     End for loop
                     Increment curY
              End for loop
       End for loop
Return matix out to ConvNeuralNet
```

This implementation is slow and has a running time of $O(n^3)$. Chris chose not to use the MATLAB-appropriate implementation described in the project description which had O(n) run time because this implementation was extremely straightforward and easy to understand. We just wanted to get a working CNN so we can analyze inputs and outputs, rather than having the most efficient code.

2.1.7 fullyConnected(im,biasvector,filterbank)

```
Initialize a 1x1x10 matrix (image)
For i = 1 to 10
       Ima = the filterbank(i) of its fourth dimension
       Imb = dot product of im and ima
       Imc = sum of all values in imb
       3rd dimension of image is equal to imc
End for loop

End fullyConnected
Return image to ConvNeuralNet
```

The fully connected layer is, in short, is an output with each filter the same size as the input image, computed as a single, scalar valued output. For this layer, it was decided to use a dot product operation followed by adding a scalar bias value. This was used in a basic for loop to ensure that the function was implemented correctly throughout the program/design.

2.1.8. softmax(im)

```
Set variable (alpha) to the max of im
Set im = im - alpha
Initilaize variable (sum)

For i = 1 to the length of im
       Sum = sum + exponential(3rd dimension of im)
End for loop

For i = 1 to the length of im
       expI = exponential(3rd dimension of im)
       3rd dimension of im = expI / sum
End for loop
Set out = im
Return out to ConvNeuralNet
```

As far as `softmax` is concerned, the input is an array size 1 x 1 x D, with an output array size 1 x 1 x D (same size). The output is the maximum across all values in the input vector, and takes the arbitrary real numbers from the vector, and converts them into numbers that can be viewed as probabilities. Since this is the case, and since the output is in the 3rd dimension, it was decided to set the maximum values of the input, and sum up the probabilities from 0 to 1, that add up to 1. This was done in for loops, and from the output mathematics as explained in depth in the project description on Angel.

2.1.9. Debug()

Debug is a standalone function that can be run without any input. It was used primarily to test `ConvNeuralNet.m` against the output in `debuggingTest.mat` to make sure that our layers were correct. It uses a `SETDIFF` operation on the "correct" set and the CNN's set to make sure everything is correct.

2.1.10 CustomTest()

This is another standalone function which can be called without any inputs. It will loop through all `*.jpg` images in the `custom_test` subdirectory and process them through the `ConvNeuralNet.m` and output the max probability class (and file name, which presumably will hold the actual object class of the image). Images sent in must be size 32x32 before they put into the subdirectory. This is a design limitation, but it still allowed us to test our CNN appropriately, albeit adding more manual work for our group to do.

2.2 Flowchart

The overall flow of our program is shown in the flowchart below, all functions stem from the `ConvNeuralNet` function. After all layers are called, the final result is then sent back to the main program.

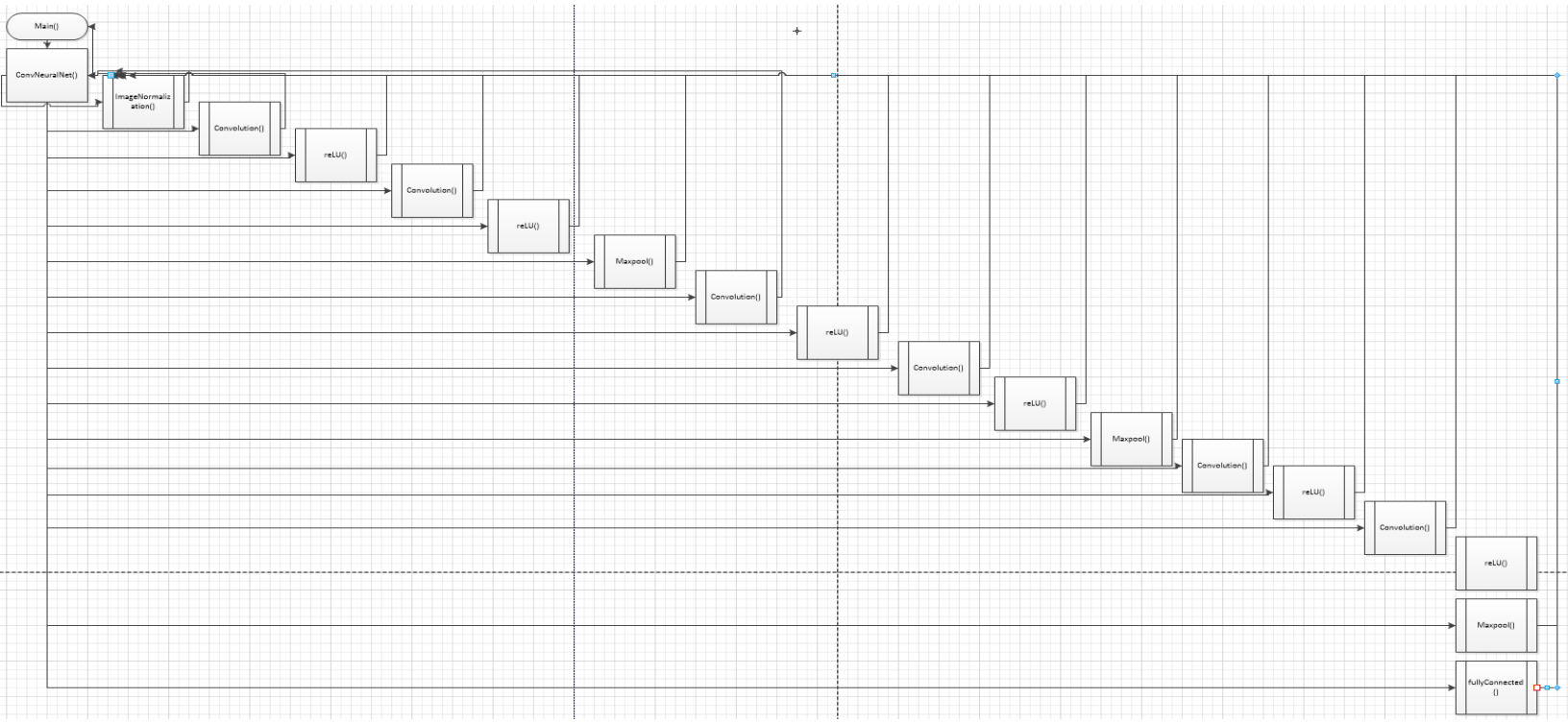

Figure 2-1: The control flowchart of our MATLAB convolutional neural network code.

3. Experimental Observations

3.1 Debugging Airplane Case Study

We will now look at the airplane image which was given to the group to test our neural network with. We will observe key aspects of some key layers to not only show our CNN works, but also explain how what we have learned in class applies to what we did in the project.

Figure 3-1: Image normalization (first layer) for the debug airplane provided.

Layer 2 (figure 3-2) starts to distort the image of the plane slightly, which is expected because the image is convolved with a filter. We believe the convolution layer is correct because these images are very similar to the output of convolving images with a filter in class. The images also seem to conform to the biases. For example, the fourth image from the left is the darkest, which makes sense because it has the lowest pixel values bias added to it.

Figure 3-2: The ten layers of output for convolution layer 2 on the airplane.

Layer 3 (ReLU) appears to be verified in the fourth image over (figure 3-3), where compared to the second layer, the contrast between the plane and the sky is much less.

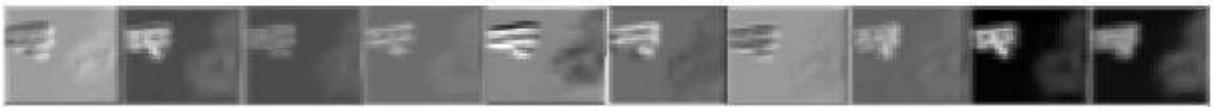

Figure 3-3: The ten layers of output for convolution layer 4 on the airplane.

Layer 9 (figure 3-4) has been through a few convolution layers, ReLU, and a maxpool layer. The output of layer 9 appears correct, because in the maxpool layer, the sky has a constant value (not accounting for noise) so the pixel values throughout the sky should be constant. When the maxpool has a set of pixels near the plane, the edges of the plane should be the max value, so they are chosen. In the layer 9 output it is very clear that edges of the plane are the only pixels with deviation after they were sent through the convolution layer. With the first

image of this, it is understandable that the image is almost constant because after inspection of the filter applied to it, most of the values are relatively close to zero as well as very close to each other.

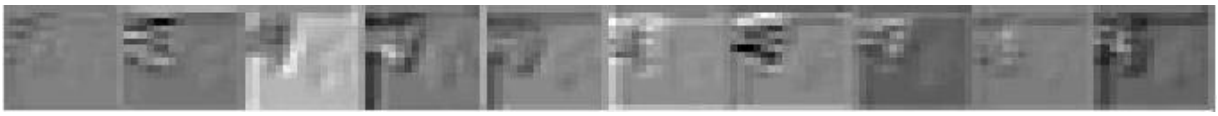

Figure 3-4: The ten layers of output for convolution layer 9 on the airplane.

After going through another maxpool and reducing the size of the image again, the plane features should take up more of the image, which is depicted in the layer 14 convolution (figure 3-5). It appears that most of the images comprise of the edges of the plane, because they were the only pixels that were not of constant value. All the images appear to be different which is expected when each has a separate bias vector added to it.

Figure 3-5: The ten layers of output for convolution layer 14 on the airplane.

Finally, after all 18 layers the probabilities come out correctly, as shown in figure 3-6, with the airplane having the highest probability. This confirms that all the layers work as they were designed to. The highest probabilities are between two types of large vehicles, which makes sense because they share some of the same structures fundamentally.

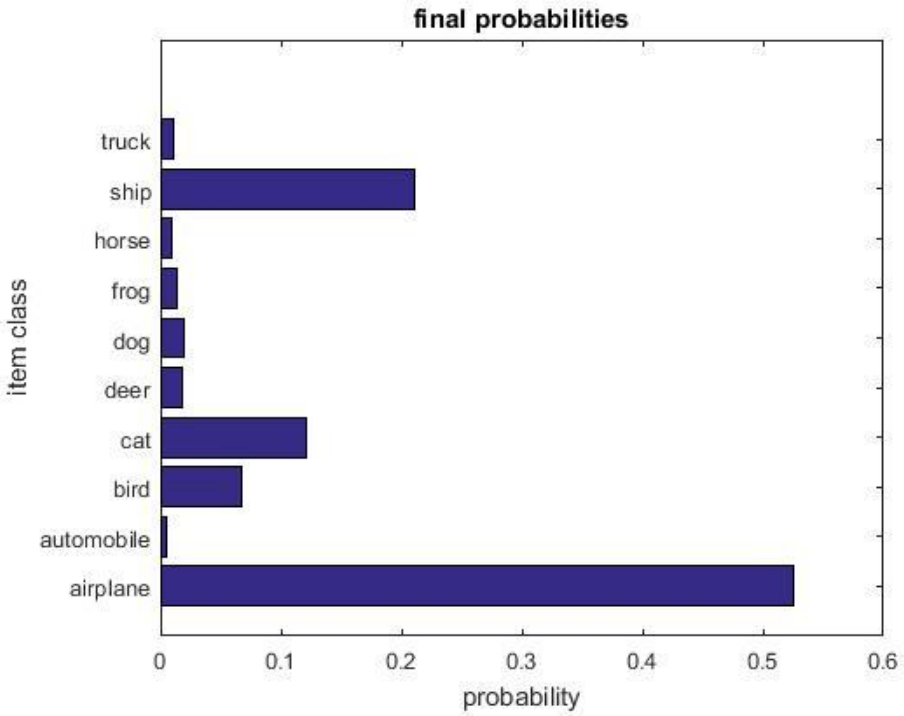

Figure 3-5: The probability results after the softmax layer 18.

3.2 Horse Case Study

Next, we'll look at an image which we downloaded from the internet of a horse. This image had the distinction of being the highest match of *all* the twenty images which we downloaded and tested, in the end it had a probability of over 80%.

Figure 3-6: Image normalization for the horse image obtained from the internet.

Convolution layer 2 (figure 3-7) of the horse resembles the layer 2 of the plane, the image values appear to be based off the bias vector they are added with, such as image 4 appears to be the lowest value pixels because they had the most negative bias value.

Figure 3-7: The ten layers of output for convolution layer 2 on the horse.

ReLU took the highest value pixel and created a smaller image from the values. After the convolution layer 4 (figure 3-8) it appears to be functioning correctly because the sky behind the image of the horse in all the images appears to be the lowest value pixels, which is expected because they are constant. Images of the horse are also all starting to look the same, with slightly different pixel values, which is expected.

Figure 3-8: The ten layers of output for convolution layer 4 on the horse.

The image resolution is progressively becoming worse due to the implementation of the maxpool function, which reduces the overall size of the image. The image is also not smoothed between convolution layers, so pixels become more apparent in layer 14 (figure 3-9). We can begin to see the different patches and regions that the CNN is focusing on for our image.

Figure 3-9: The ten layers of output for convolution layer 14 on the horse.

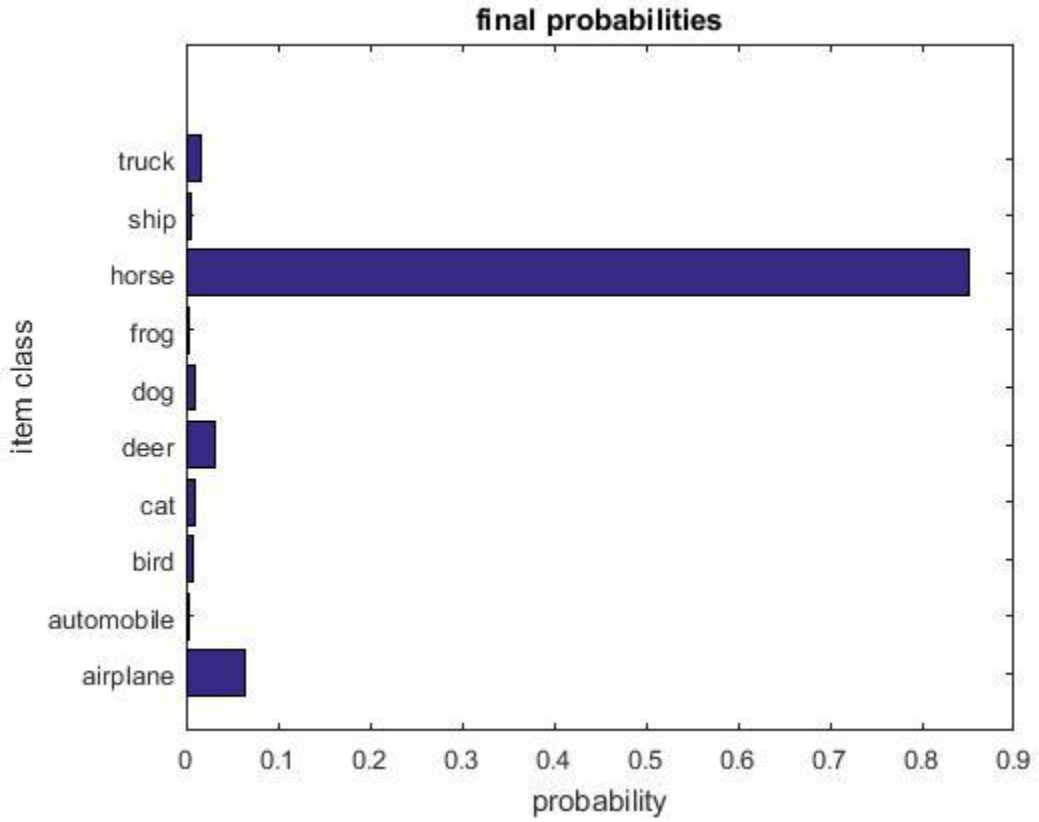

Figure 3-10: Final probability scores for the horse image after passing through all 18 layers of the CNN. He highest score is near 85%.

3.3 Face Case Study

Next we'll look at the effects the convolutional neural network had on an image that we as operators knew that the net was not trained for. In this case, we used the face of one of our group members, Chris.

Figure 3-11: From left to right: image normalization, convolution (layer 2), convolution (layer 9).

The image of Chris is characterized as a bird. Although Chris was quite insulted by the result, we recognize the interesting part about the final probabilities is that all animals had higher probabilities than objects. This makes sense because animals will have more complex features than vehicles. Vehicles tend to have either a flat surface, which will have a relatively constant pixel value throughout, or constantly curing surface, which will cause concentric waves of gradually changing pixel values. So we can see that although the CNN was incorrect overall, it was still applying some of its patches *relatively* correctly. Mainly facial features and other "living-organism" features.

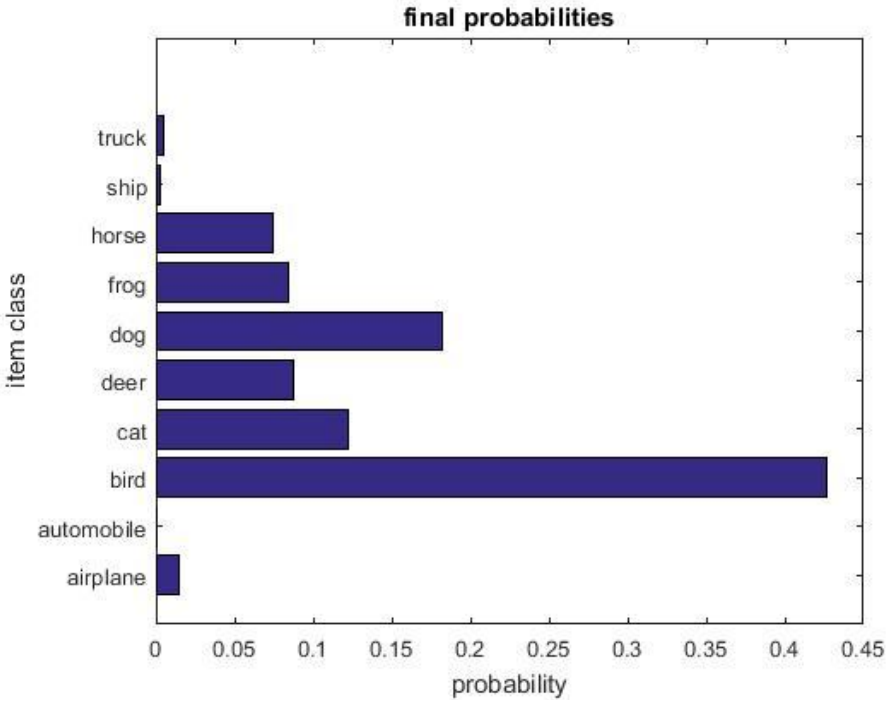

Figure 3-12: Chris' ten probability scores from layer 18.

4. Performance Evaluations

Next, we will present our experimentally generated performance evaluations, showing the different statistical data that we gathered. We will then interpret the data and show how it applies to what we have learned.

4.1 Confusion Matrix

	'airplane'	'automobile'	'bird'	'cat'	'deer'	'dog'	'frog'	'horse'	'ship'	'truck'	TOTAL	Accuracy
airplace	**531**	41	65	37	10	8	18	38	**210**	42	1000	0.531
automobi	40	**519**	9	26	10	7	19	29	**111**	**230**	1000	0.519
bird	87	8	**386**	117	97	70	**104**	88	25	18	1000	0.386
cat	39	18	**127**	**325**	45	**136**	**186**	89	13	22	1000	0.325
deer	53	6	**270**	69	**259**	38	**162**	**114**	22	7	1000	0.259
dog	19	7	**151**	**222**	49	**281**	**111**	**125**	20	15	1000	0.281
frog	10	7	**120**	**125**	93	23	**557**	33	9	23	1000	0.557
horse	32	7	73	98	77	94	54	**533**	13	19	1000	0.533
ship	**192**	84	35	44	7	8	10	16	**542**	62	1000	0.542
truck	69	**191**	23	41	4	9	30	68	**127**	**438**	1000	0.438
TOTALS												
Correct	Incorrect	Accuracy										
4371	10000	0.4371										

Figure 4-1: The confusion matrix for all 10,000 images in the given dataset and their individual and total accuracies

The confusion matrix was run on our program and is displayed above. From this we find that our accuracy rate is 43.71%, which seems pretty low. The categories that seem to be easiest to identify are airplanes, automobiles, horses, and ships. This may because they have distinct lines that are easy to characterize, whereas animals such as cats, dogs, and frogs all have similar features making them harder to distinguish between. It appears the most common confusions include: automobiles and trucks, airplanes and ships, bird and cats, cats and frogs, deer and birds, dogs and cats, and horses and cats.

4.2 Classification Plot

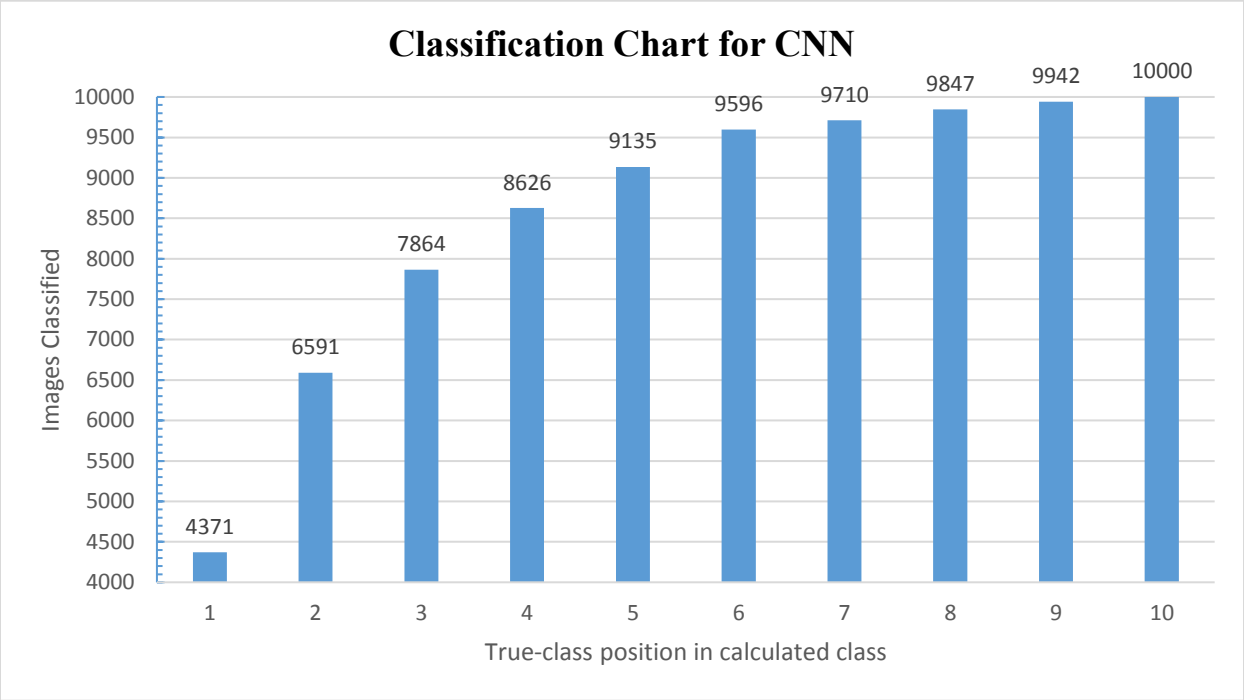

Figure 4-2: The classification chart for all ten-thousand images. As a way to put it: how many "guesses" it took for the CNN to get the correct class, 1 being it got it on the first try.

Figure 4-2 above shows the classification chart for our network. Interestingly, after two tries the CNN gets the correct image 65% of the time. This is a huge difference from the 43% of the first try. Moreover, by the 5th try the CNN can get the right image 91% of the time. After this point it seems that if the CNN has not gotten the correct image, it will very unlikely be able to get the image.

./

5. Exploration with Our Convolutional Neural Network

5.1 Using Custom Images Within Given Classes

`CustomTest.m` is within the `.zip` file that processes images we have created and gathered from around the internet and placed in the `custom_test` folder. The accuracy of these images is similar to the accuracy of running the `cifar10` (about 40-50%, and much worse for animals) image set which is as expected because we have a variety of images encompassing different portions of the images. The figure below shows the images that we gave to our program and what it classified them as.

Figure 5-1: The custom images of objects within the defined classes which we inputting into our neural network, along with their calculated classes and probability scores.

From the pictures obtained from the internet above, our CNN got 8 correct and 12 incorrect. That is a 40% success rate which is almost exactly in line with our experimental results which were discussed in section 3.1. Some of the classifications are interesting as well. For example, in the bottom right, the CNN classified the truck as an automobile, which isn't

necessarily incorrect. The CNN also can generally tell the difference between animals and different types of vehicles, which is also in line with our earlier findings. Interestingly, images like the car with a white background (row 1, column 4) respond negatively, possibly due to their background, because of the white. Maybe the CNN thought it was clouds which led it to be classified as an airplane. The picture of the bird (row 2, column 1), however, responds positively because of the blue sky behind it. Likewise, the plane with a blue background was successfully classified but the flying over the ocean was classified as a ship. So the background noise behind the object could play an important role when the CNN tries to classify an image.

5.2 Exploring a "Binary" Convolutional Neural Network

If we could call our CNN a "binary" neural network we want to know if it can tell the difference between a living organism and an object, we hypothesize that our network will perform very well. Using the custom data set in section 5.1 the CNN was 95% correct in this regard. If we use the data from the confusion matrix in section 4.1 to calculate Organism vs. Object, the CNN can distinguish between them 89.3% of the time; this statistic could prove moderately useful to someone wanting to a CNN. Below is the figure for the results of a CNN using *only* images that are not in a defined object class, which could help prove our CNN is more useful on a broader scale.

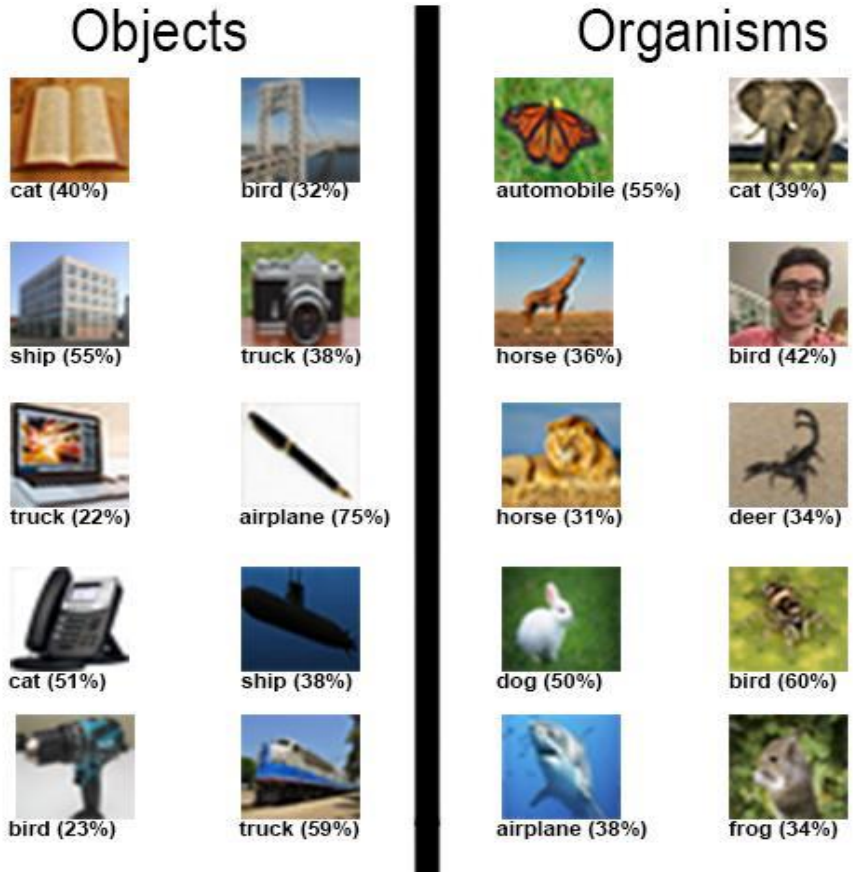

Figure 5-1: The custom images of objects outside the CNN's defined classes; separated into objects and organisms, along with their computed class and probability scores.

We can summarize the results of our binary convolutional neural network as follows. Using the images above the CNN was able to guess the correct class (object or organism) 70% of the time. This is higher than the total accuracy of about 43%. So we can say that we may have a significantly better chance to guess the class of an image if we are choosing between only object or organism. Things like eyes, fur, and body structure may lead the CNN find patches similar to living things. Whereas in objects things like hard corners could be a characteristic of "object-like" things.

6. Conclusion

Our convolutional neural network that we implemented in MATLAB is not extremely accurate. A 43% accuracy rating is not very high; but is still much higher than the 10% if we were to randomly assign one of the ten classes for each image; so we can say that our CNN somewhat accomplishes its goal. As we discussed in class; some CNN's span hundreds of layers, which would help add to accuracy. Because ours was only eighteen layers and used only 32x32 images as input, we expected that our CNN would not perform perfectly or optimally.

However, we did learn a lot throughout the course of the project. Not only did we gain some really good MATLAB experience, we also got to see how the CNN layers evolved through each in/out layer. We began to see different patterns emerge regarding how the CNN transformed different images, such as how it differentiates between objects and organisms. Overall this was an interesting and fun project which has a lot of applications to today's technology; since CNN's are much of the future and cutting edge of image processing and computer vision.

7. Contributors

Our group split up the functions between the members. Each member's contributions is itemized in the sections below.

7.1 Christian Myers

- Wrote FullyConnected, helped write Main/Debug
- Wrote bulk of Introduction
- Wrote Section 2: Flowchart and Procedural Output Pseudo-Code
- Wrote bulk of analysis in section 3
- Wrote analysis of confusion matrix
- Proofread and added thoughts throughout report

7.2 Connie Couey

- Wrote ReLU and helped write Convolution
- Helped contribute to Introduction and Conclusion
- Proofread and added thoughts throughout report

7.3 Christopher Carney

- Wrote Maxpool, Main, ConvNeuralNet, Debug, CustomTest
- Connected layers together and fixed bugs in subroutines for debuggingTest.mat to work
- Created graphs, images, tables, and figures for the report
- Executed experiment and analysis in section 5
- Proofread and added thoughts throughout report
- Helped contribute to introduction and conclusion

7.4 Darshan Shah

- Wrote Softmax function
- Proofread and added thoughts throughout report