

Assignment 2a (Due 5/18/2017, 50 points)

Generalization and Representations

| | |
|---------------------|--|
| Team members | Michael Cho, Annie Didier, Dustin Fontaine, Dylan Fontaine |
| Comments | |

Reference for filter visualizations: <http://blog.keras.io/category/demo.html>

1. CNNs on paper (5 points)

Look at the definition of layers in the example net code and explain what each layer does individually and as a whole. Is this network deep enough? Are the convolution windows too small/large? Hint: Consider the size of the overall receptive field. Does the network see the whole input? Hint: How much does the network shrink from each layer? At what point does the information evaporate (vanishing convolutions)?

Give some examples on how the CNN definition can be changed to improve if we double the input image size (64x64x3). E.g. More layers, smaller/larger convolution sizes, pooling, zooming. Justify why your proposed changes can help. No need to show plots or run code for this part, but if you choose to do so, please show at most 4 plots.

The types of layers that show up in the example net code are Convolutional2D, MaxPooling2D, Dense, Flatten, and Batchnorm (if the user specifies). What each one does is described below.

Convolutional2D: Each convolutional layer applies a set of small (3x3) filters that “slide” across the grid of input data to detect various edges. With padding=‘same’, the output will have the same dimension of the input.

MaxPooling2D: Compresses the output from a previous layer, keeping only the most extreme/prominent information. For each 2x2 area in the preceding input, it keeps only the value that was highest of the four. This is a way to summarize the information and eliminate some noise. The width and height of the output is half that of the input.

Dense: a fully connected neural network layer where every input is connected to every output

Flatten: changes the shape of the data into a nx1 vector

Batchnorm: for each batch of data that runs through the model, this layer normalizes (makes the mean ~0 and the standard deviation ~1) the activations from the previous layer. This is currently set to false, so it not currently being applied.

As a whole, these layers work together to identify large objects in an image. The first layer looks at the raw pixel inputs and detects edges, but as you go deeper into the network the model can piece these edges together and detect larger, more complex object parts.

With 5 vgg layers, this is deep enough and can see the entire image. Starting at a 32x32 image, it is shrunk down after each set of convolutional layers with maxpooling to half the original width and height. So, by the end of the fifth layer, the size is cut down to 1x1 where going any deeper would be useless.

The window size stays the same for each convolution, even as the images shrink with pooling. After layer

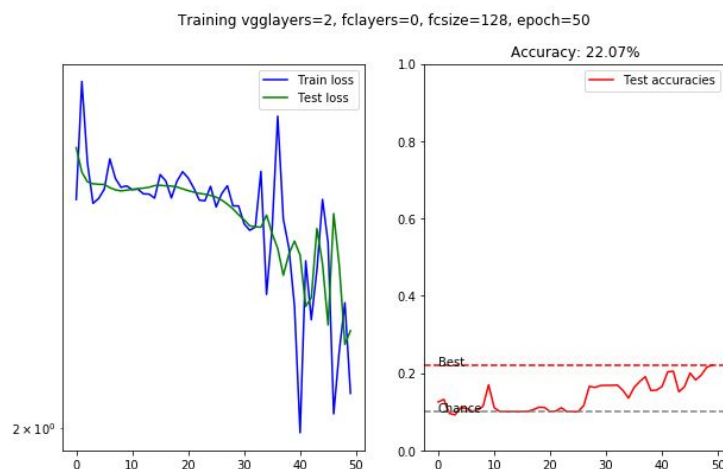
4, the image would be 2×2 . With padding, the image size is essentially 4×4 , so the window can step over, down, and over and down.

If we increase the size of the image to 64×64 we would need to add more layers to get down to the 1×1 granularity. Similarly, increasing the window size will provide the first few layers of the model with a broader overview of the image and may allow it to capture larger edge features. Keeping the same window size as for the 32×32 size will give the model more granularity.

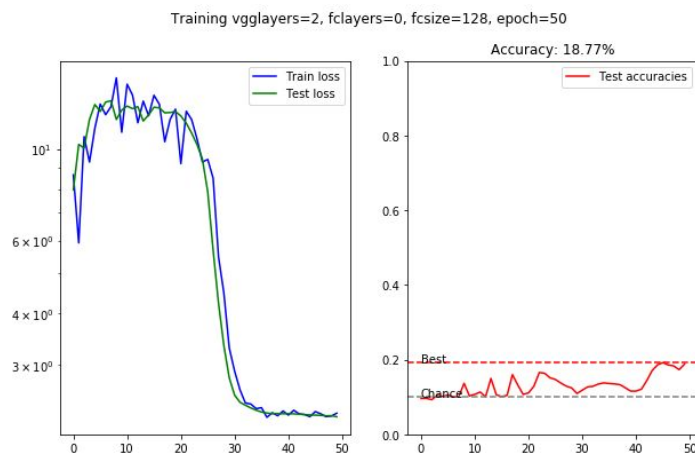
2. Transfer learning: Adapting pre-trained VGG weights (5 points)

Since training a good CNN takes a long time (~ 2 -3 weeks), we use a pre-trained a VGG network in Keras. In this exercise, we'll be using the VGG weights to pre-initialize a model for CIFAR 10 (a different dataset).

Step 1 – Load and run “assignment2-starter.py”. Let the network train for 50 epochs. Note how the loss curves, predictions and filters perform. You can right-click the console and print to a PDF file for reference.



Step 2 – Enable transfer learning by setting “`cfg.xferlearning`” to 2. Again, run the model for 50 epochs. Note the results and compare. What’s different when transfer learning is enabled? How does that affect the classifier? Which model trains faster?



When transfer learning is enabled, weights in the network are initialized to values identified from a “pre-trained” network. The pre-trained model ran slightly faster than the original, but generally had lower accuracy.

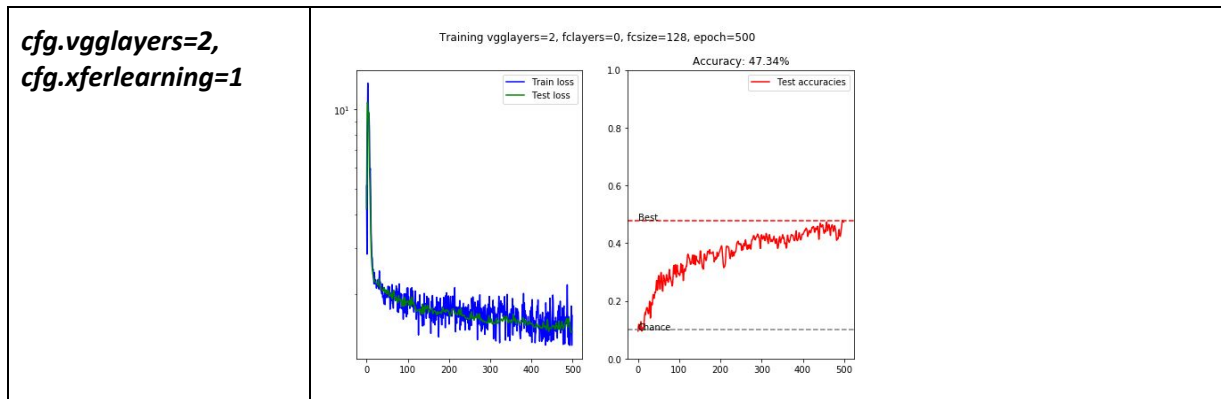
3. Optimizing Transfer Learning (10 points)

Experiment with the definition of the VGG net by changing the variable “*cfg.vgglayers*” to enable certain convolutional layers. Valid values are 0-5. Then change “*cfg.xferlayers*” for transfer. Valid values are 0-12; -1 disables transfer learning.

Step 1 – Which layers result in the best classification performance? How does that change the filters?

We tried many combinations of *cfg.vgglayers* and *cfg.xferlayers*. A few of our results are displayed below, the rest are stored in the appendix.

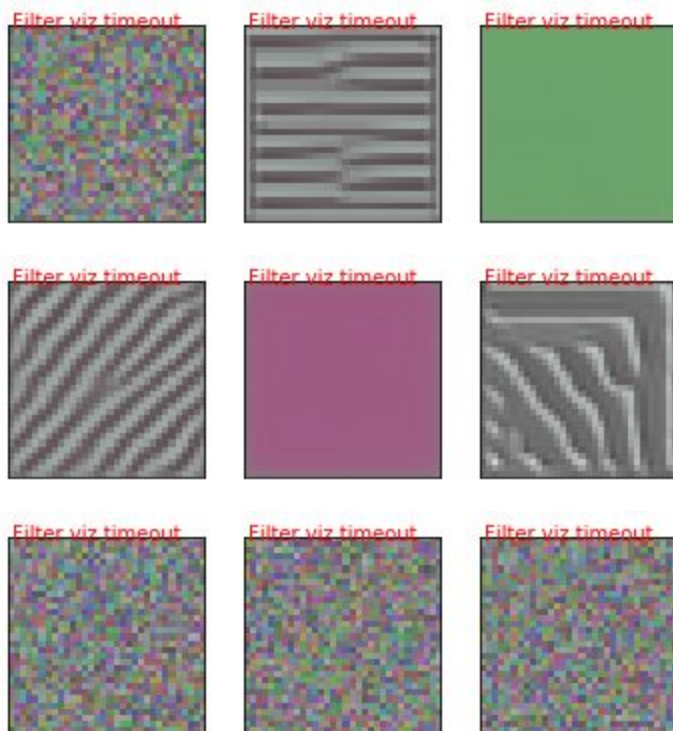
| Vgglayers,xferlearning | Loss and Accuracy Plots |
|--|---|
| cfg.vgglayers = 1 cfg.xferlearning = -1 | <p>Training vgglayers=1, fclayers=0, fsize=128, epoch=500</p> <p>Accuracy: 42.87%</p> |
| cfg.vgglayers = 1 cfg.xferlearning = 0 | <p>Training vgglayers=1, fclayers=0, fsize=128, epoch=500</p> <p>Accuracy: 37.40%</p> |
| cfg.vgglayers=2, cfg.xferlearning=-1 | <p>Training vgglayers=2, fclayers=0, fsize=128, epoch=500</p> <p>Accuracy: 40.45%</p> |



The best combination of vgglayers and xferlayers that we tried was `cfg.vgglayers=2` and `cfg.xferlearning=1`. The model reached an accuracy of over 47% after 500 epochs.

We could tell that the model does not perform very well when all of the layers are pre-trained. The pre-trained weights are locked in and not allowed to update, so the completely pre-trained model cannot learn at all from the dataset. The models that worked the best had a small amount of pre-trained layers, followed by a few untrained layers. The pre-trained layers can detect edges easily, but the untrained layers can learn specific shapes associated with each image class in the dataset.

The filters for our best combination of layers are shown below.
vgg 2 transfer 1:

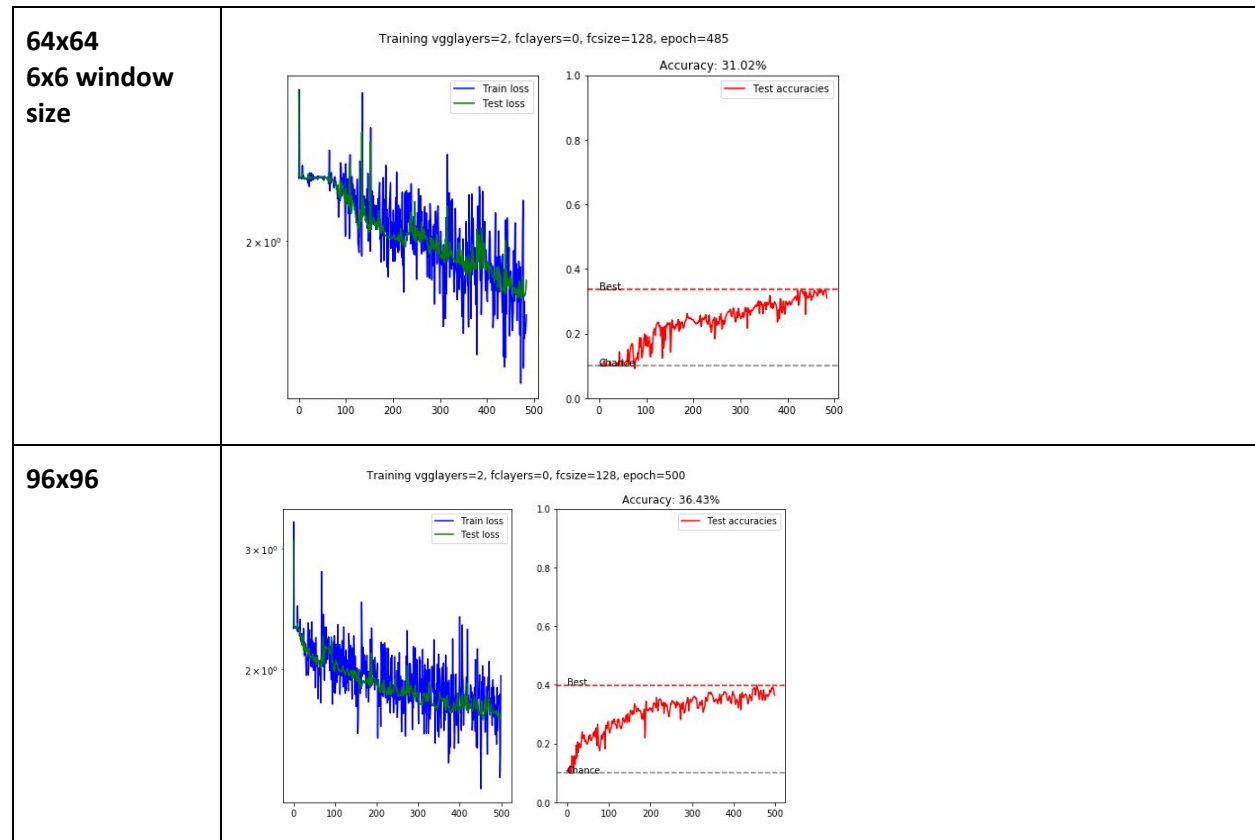




The first two layers were pre-trained, so you can see a clear pattern in the first two rows of filter visualizations. The filters shown look like good edge detectors. The next two layers were not pretrained, so patterns in the next two rows of filter visualizations are much less apparent. The model only ran for 500 epochs, so the random weights did not have as much time to converge as the pre-trained weights did.

Step 2 – Change the `cfg.imsz` setting to make the images larger, e.g. (64,) or (128,). Does a 2x (64) zoom help? (>4x (128) makes this very slow, so use 3x (96) or less)

| Image Size | Loss and Accuracy Plots |
|--------------|--|
| 32x32 | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 39.04%</p> |
| 64x64 | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 39.54%</p> |



Increasing the image size gives a slight boost in model performance when going from 32px to 64px. While it does take longer for the model to run, there is more information captured in the images with more pixels. The accuracy was lower when we tested 96px, but it still seemed to be on it's way up. If we trained for longer than 500 epochs, we believe that the accuracy would have been higher.

Step 3 – What is the best combination of zoom and convolutional layers? Is there an optimal match for these two settings? Hint: do convolutions vanish without zooming?

Hint: Conversely with your projects, you can downsample to 32x32 for faster training at first

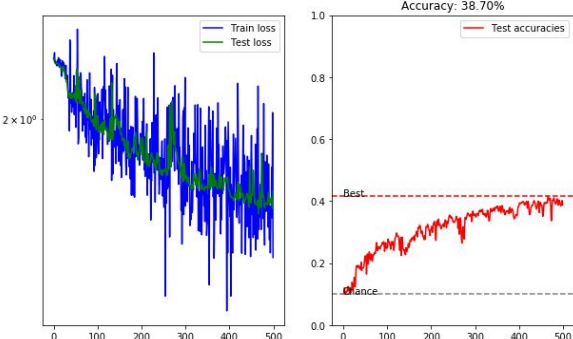
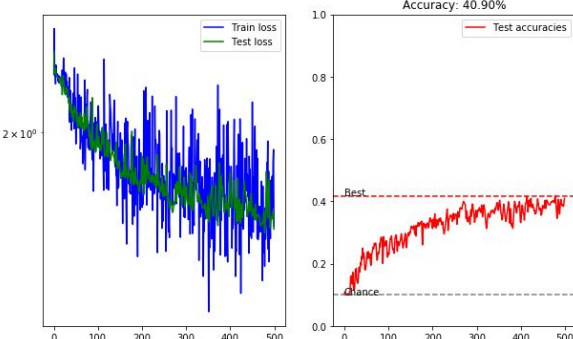
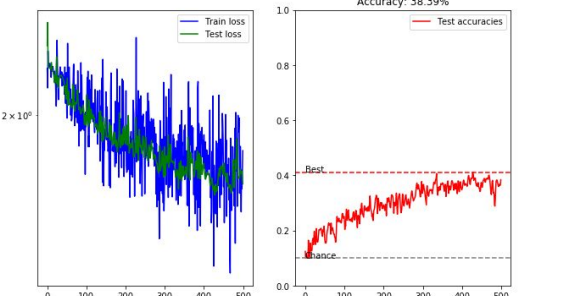
Does transfer learning help more with shallower networks or deeper networks? Is the use of transfer learning justified by the need to “tune” the network further?

Transfer learning seems to help more with deeper networks, but only up to a certain point. As shown in the appendix, running the model for 3 vgglayers and no transfer learning had 20% accuracy and this jumped to 32% accuracy with a xferlearning value of 0 and up to 39% accuracy with a xferlearning value of 1. However, if we use more layers accuracy seems stuck at 10%. Transfer learning helps images learn certain image components really fast, but at a certain point we have to leave certain layers to reweight on their own to learn their own idiosyncrasies.

4. Optimizing Classification (5 points)

The network currently uses two fully-connected layers before the softmax to perform classification. Justify why this is optimal or not. If not, how many fully connected layers should be at the end? What should their sizes and activations be? Change `cfg.fclayers` to adjust the number of layers. Remember that

0 layers is also a valid option. Note: optimal is defined as the fewest/smallest layers without lowering accuracy.

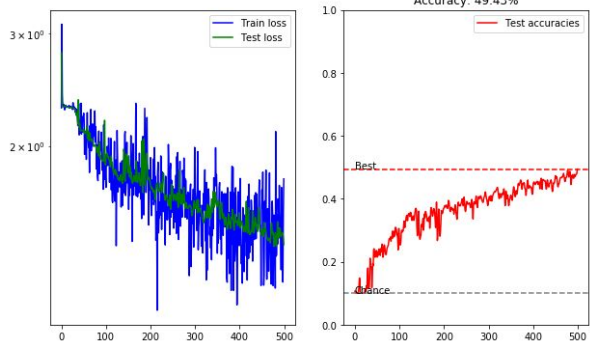
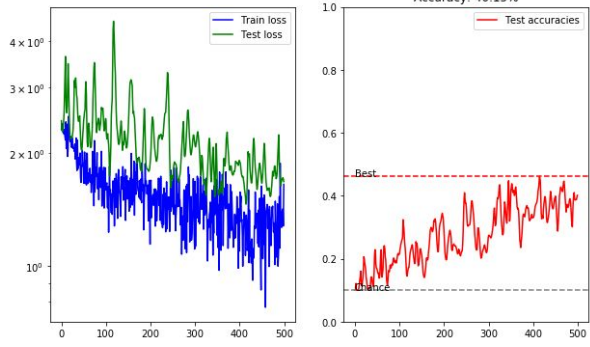
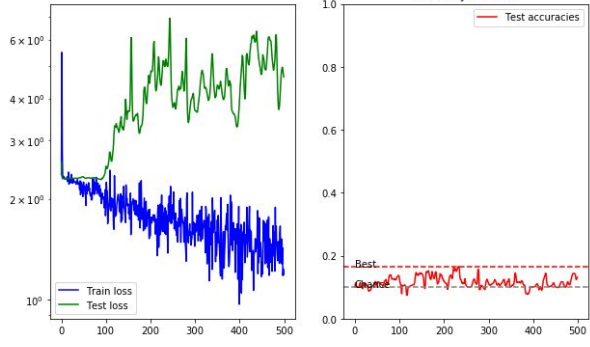
| cfg.fclayers | Loss and Accuracy Plots |
|------------------|--|
| cfg.fclayers = 0 | <p>Training vglayers=2, fclayers=0, fcsz=128, epoch=500</p>  <p>Accuracy: 38.70%</p> |
| cfg.fclayers = 2 | <p>Training vglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 40.90%</p> |
| cfg.fclayers = 4 | <p>Training vglayers=2, fclayers=4, fcsz=128, epoch=500</p>  <p>Accuracy: 38.39%</p> |

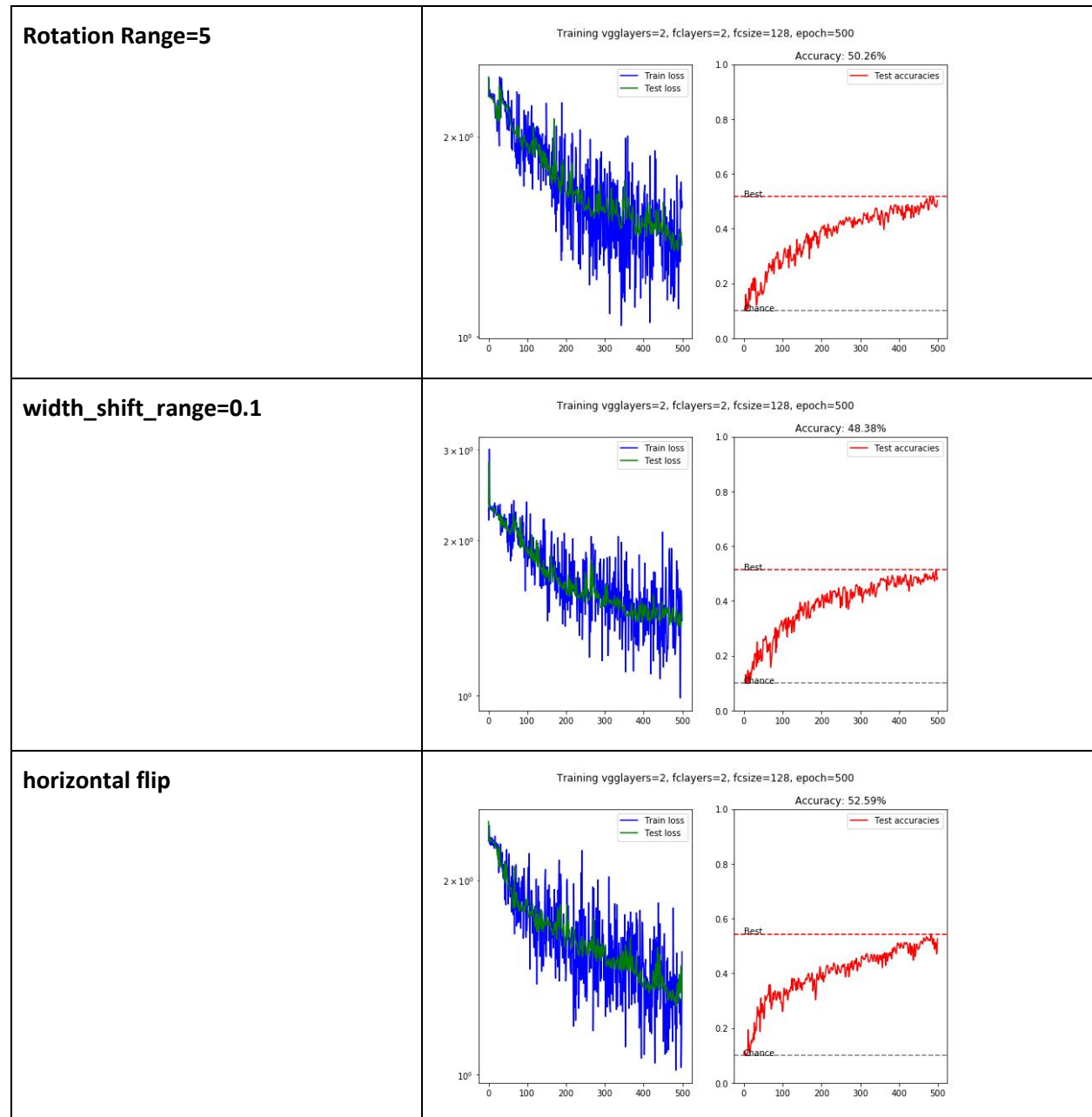
After running different configurations with various number of fully connected layers, it was determined that 2 layers is optimal for this dataset (with other parameters set as default). This is optimal because it achieved the highest test accuracy, and had the smallest number of layers without lowering the accuracy (thus eliminating unnecessary complexity). Removing any layers could hurt the overall efficacy of the model, while adding more would require additional overhead because of the additional parameters and may take longer to train because of issues such as the vanishing gradient.

The size of the fully connected layer should be the same size as the output of the immediately preceding convolutional layer. The activations should be in the active region for training to be quickest.

5. CNN, augmentation, color representations and Dropout (10 points)

So far, our CNN has only been trained using the original CIFAR 10 data (bootstrapped with the pre-trained VGG model) without augmentation. Try changing some of the augmentation parameters and describe what happens to the training loss, testing loss, and accuracy score. Which augmentations matter the most? Hint: try one at a time, don't try setting everything to the max, that gets very slow.

| Configuration | Performance |
|-------------------|--|
| none | <p>Training vglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 49.43%</p> |
| featurwise_center | <p>Training vglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 40.15%</p> |
| ZCA whitening | <p>Training vglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 13.15%</p> |



The modifications that had the biggest effects were rotation, width and height shift, as well as horizontal and vertical flip. For these modifications, both the training and test losses decreased relatively steadily (with some noise) and the accuracy steadily increased (with some noise). Given this very diverse dataset, it makes sense that these various transformations could have a big impact on performance.

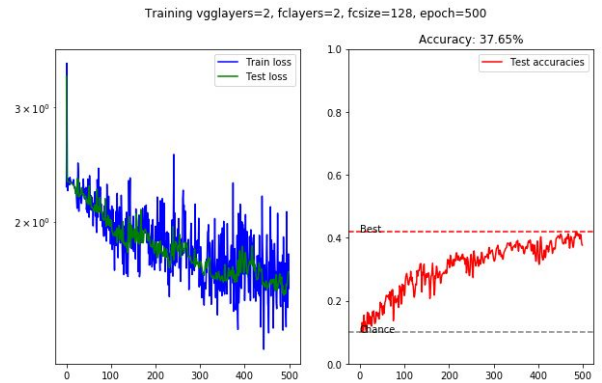
With the other modifications, training loss followed a similar pattern of decreasing throughout. However, the test loss did not track with the training loss like it did in the other trials. Test loss was either decreasing but significantly above train loss, or not decreasing at all.

Apply various levels of dropout to the fully connected layers using `cfg.fcdropout`. Describe what happens, and determine the “optimal” level of dropout to achieve the best performance. Hint: change the default

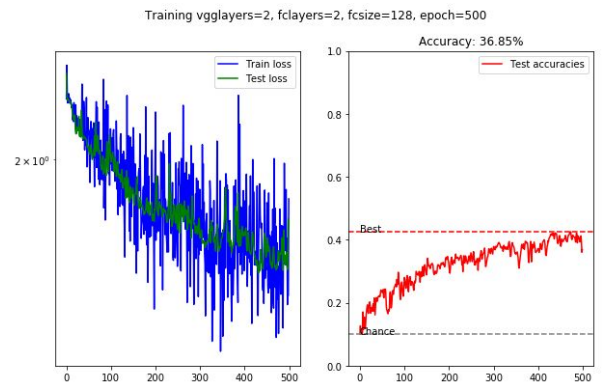
dropout settings. Show plots of accuracy and loss to justify your position. Is there a setting that's too high or too low?

| Dropout | Performance |
|-------------|--|
| dropout=0.0 | <p>Training vgglayers=2, fclayers=2, fcsz=128, epoch=500</p> <p>Accuracy: 40.90%</p> |
| dropout=0.1 | <p>Training vgglayers=2, fclayers=2, fcsz=128, epoch=500</p> <p>Accuracy: 40.23%</p> |
| dropout=0.2 | <p>Training vgglayers=2, fclayers=2, fcsz=128, epoch=500</p> <p>Accuracy: 39.14%</p> |

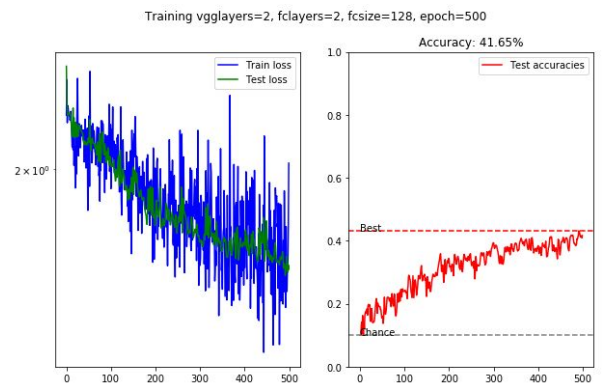
dropout=0.35



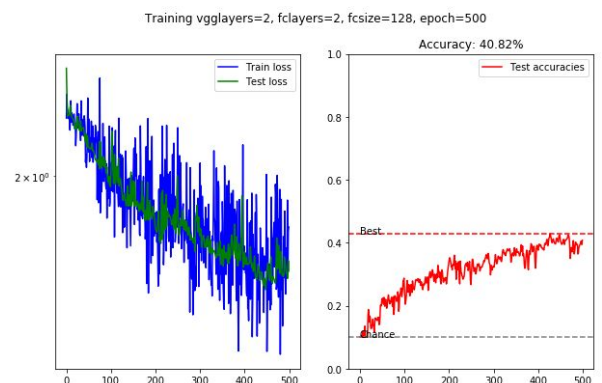
dropout=0.45



dropout=0.5



dropout=0.7



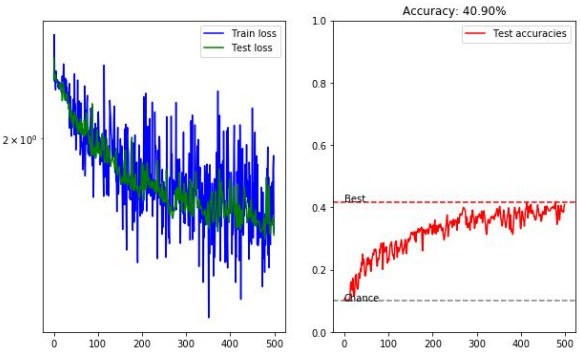
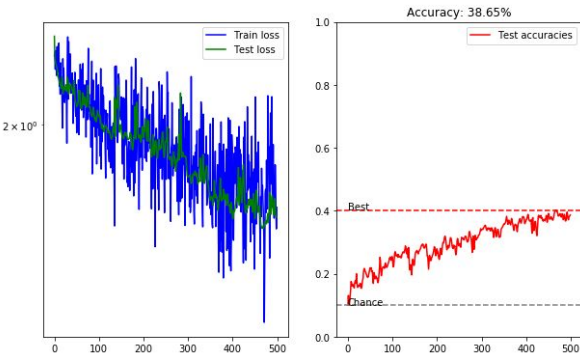
The best level of dropout after extensive testing was found to be 0.5. This had the highest accuracy and lowest loss after 500 epochs of training. The dropout levels above 0.5 and below 0.5 performed worse, showing that you must achieve a good balance of dropping out enough to ensure robustness but not dropping out so much that training is inhibited.

Try comparing training and accuracy with the `cfg.hsv` parameter. HSV stands for hue, saturation and value (brightness). In HSV mode, the mapping is $H \rightarrow R$, $S \rightarrow G$, $V \rightarrow B$. Which HSV channel achieves robustness to color changes? Which channel picks up on colors? Is HSV or RGB a better choice for color representation?

The Value channel achieves robustness to color changes. Even if the hue changes, this channel should remain similar in similar images.

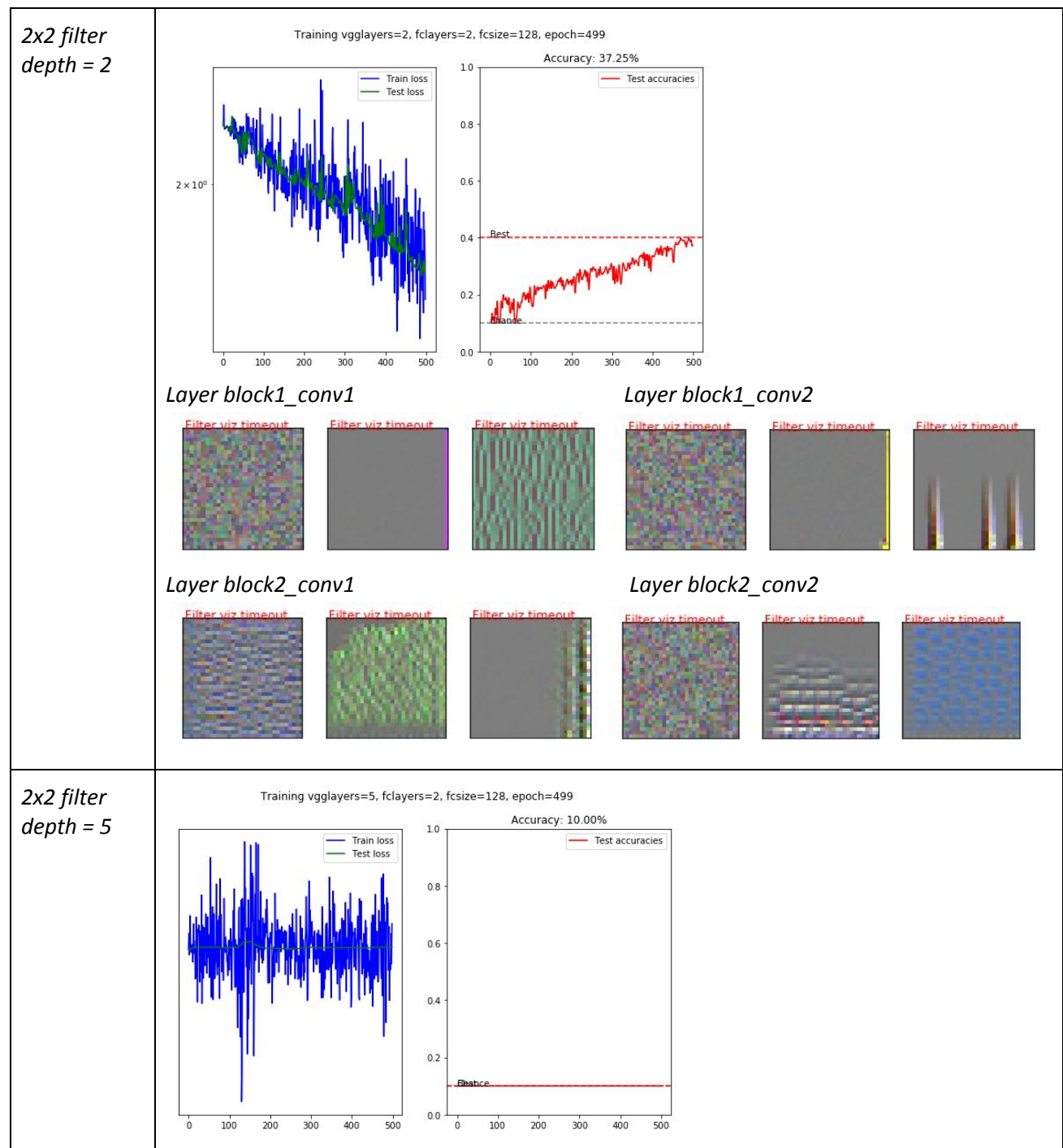
The Hue channel picks up on colors. Even if lighting changes, the Hue channel will remain constant if the picture is of the same object.

From our single run of 500 epochs, the RGB color space was found to perform better in terms of loss and accuracy. After some research, we found that the VGG network used the RGB representation. The VGG network was also classifying natural images, so it makes sense that this classifier would work best with the same representation choice used by VGG.

| Color | Performance |
|-------|---|
| RGB | <p>Training vgglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 40.90%</p> |
| HSV | <p>Training vgglayers=2, fclayers=2, fcsz=128, epoch=500</p>  <p>Accuracy: 38.65%</p> |

6. Representations & Heatmaps (5 points)

Take the network and visualize the maximum activations in each layer (note: this is already done during training, see the filter visualizations). Argue why certain filters are optimal for the original CNN dataset. Use plots/images to show the activations in the conv* filters. Consider your answers in part 1 about the convolution window size and depth. After looking at the filters learned, do you still agree with your earlier assessment?



Layer block1_conv1



Layer block1_conv2



Layer block2_conv1



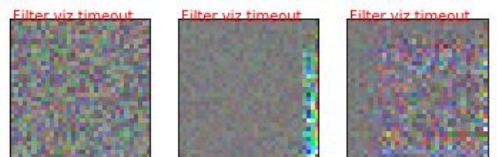
Layer block2_conv2



Layer block3_conv1



Layer block3_conv2



Layer block3_conv3



Layer block4_conv1



Layer block4_conv2



Layer block4_conv3


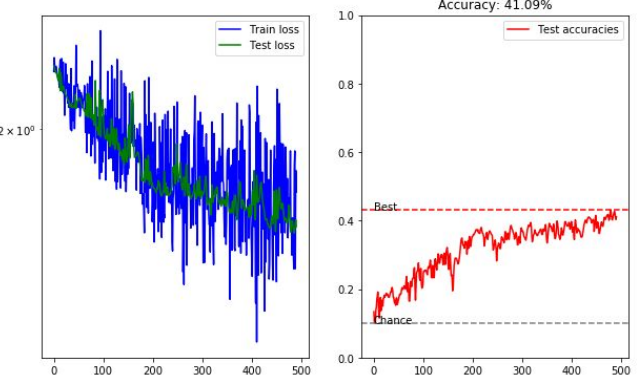
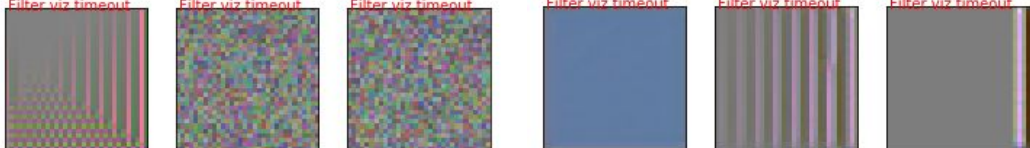
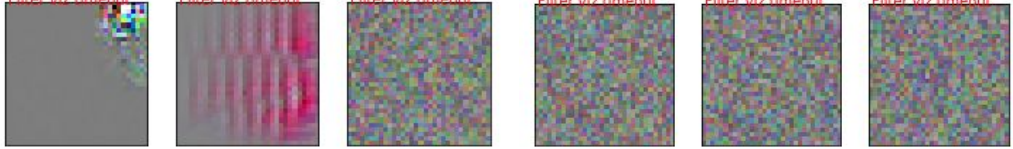
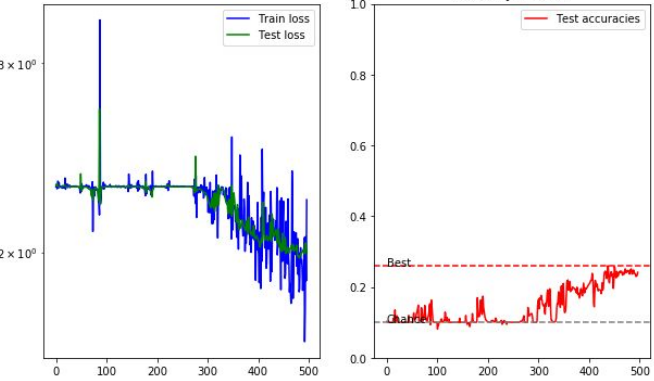


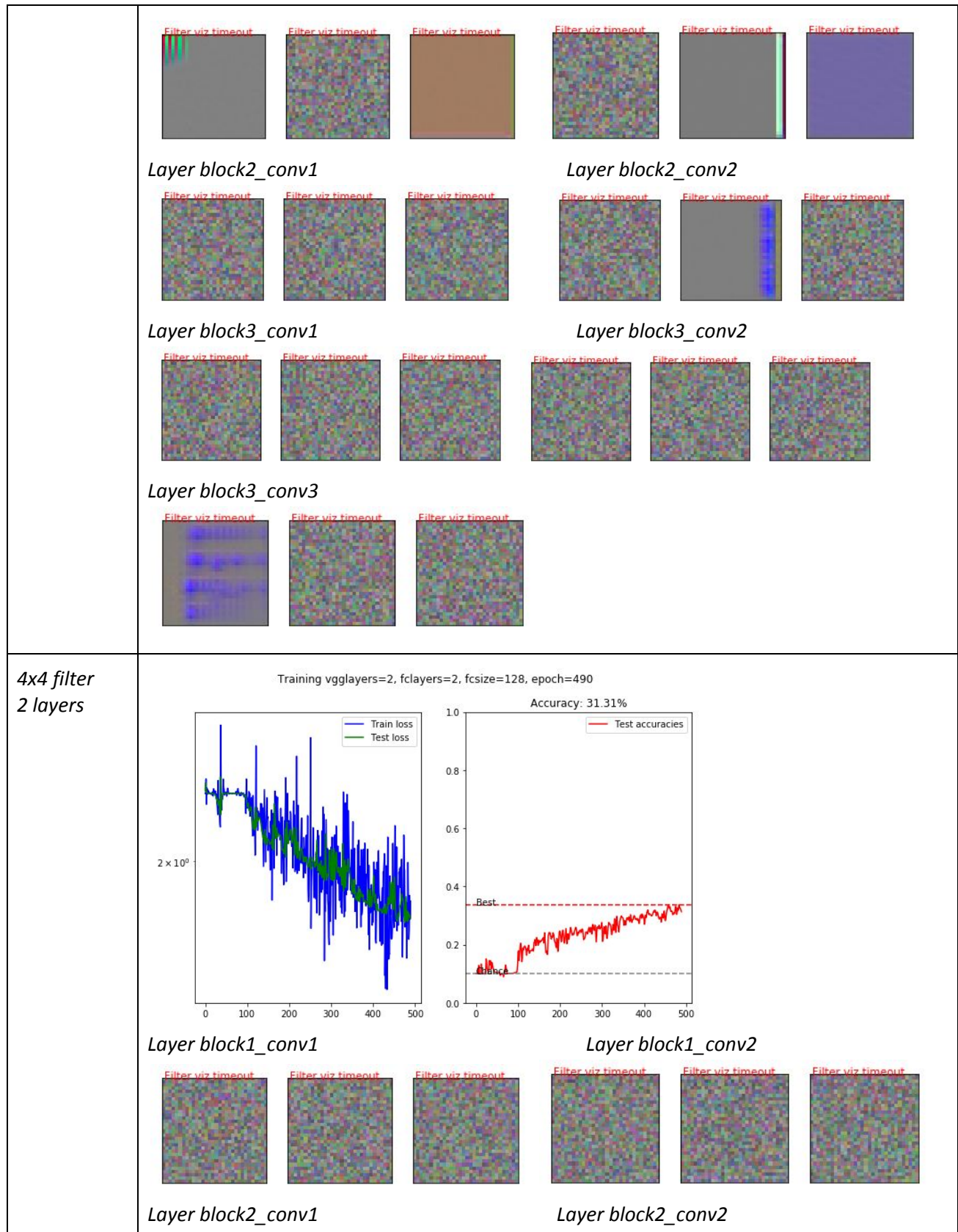
Layer block5_conv1

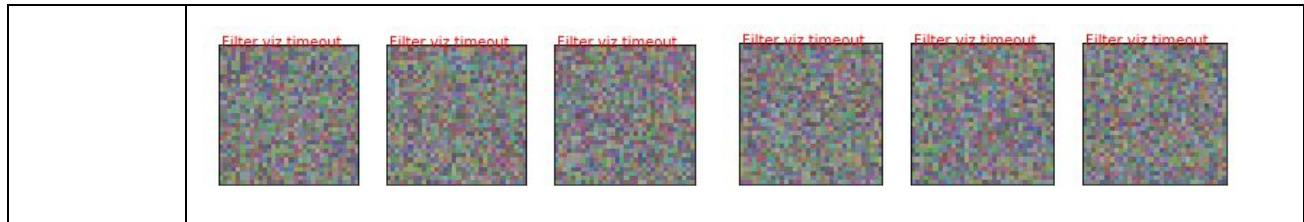


Layer block5_conv2



| | |
|---------------------------------------|--|
| | <p><i>Layer block5_conv3</i></p>  |
| <p><i>3x3 filter 2 layers</i></p> | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=491</p>  <p><i>Layer block1_conv1</i> <i>Layer block1_conv2</i></p>  <p><i>Layer block2_conv1</i> <i>Layer block2_conv2</i></p>  |
| <p><i>3x3 filter 3 layers</i></p> | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=497</p>  <p><i>Layer block1_conv1</i> <i>Layer block1_conv2</i></p> |





Using a 2x2 filter with a depth of 2 resulted in decent accuracy. The filter in block 1.1 seems to pay attention to edges on the right hand side of the image and vertical edges throughout. Layer 1.2 also pays attention to an edge on the right hand side, but also picks up on vertical colored edges at the bottom of the image. The filters in layer 2.1 pay attention to a larger portion of the right hand side of the image and look for blue or green areas in the center. The final layer seems to refine this, activating for blue to green transitions in small horizontal features, and another filter looks at finer detail and color shifts at the bottom of the image.

Increasing the number of layers to 5 resulted in much poorer performance. Similarly, as the depth of the layers increased with a 3x3 filter the performance decreased. We can see that In the first two layers the filters pay attention to the edges of the images or activate for specific colors. However, in deeper layers the activations appear to be random. They no longer pick out edges or finer details of edges and there is no clear pattern to the color activation. With a 32x32 image, by the third layer the size of the image has been reduced to 8x8 by pooling, so the filter, irrespective of its size, cannot make enough steps, or convolutions over non-zero (i.e. non-padding) values , to pick out meaningful features.


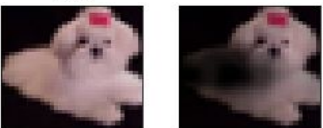








We found that a 3x3 filter with 2 layers had the best accuracy. The first layer seems to look for corners along the bottom right and vertical edges along the right side of the image. Layer 2.1 looks for some sort of light spot in the top right and smaller red edges.

Finally, increasing the filter size to 4x4 decreases accuracy and results in completely random activations. Now, with a 32x32 image and 4x4 filter, the ratio of the image to filter size is too small to allow for sufficient convolutions to detect edges or meaningful object parts.

The optimal filter size and depth is such that the filter is large enough to pick out features like edges and the image to filter size ratio is large enough for the filter to convolve for several steps.

Show a few example heatmaps where the network classifies correctly and some where it misclassifies. Explain how this changes as the network is trained more, and qualitatively how your modifications (regularization, augmentation, dropout, etc) help improve the heatmaps. Do you think the network has truly learned how to classify, or is it just memorizing training examples?

Good Examples

| <p>dog dog: 21.1%</p>  <p>RGB</p>  <p>focuses on face of dog</p> | <p>automobile automobile: 72.4%</p>  <p>RGB</p>  <p>looks at profile of car, as well as the headlights and tires</p> | <p>frog frog: 81.4%</p>  <p>RGB</p>  <p>focuses on round shape and rough green/brown skin pattern</p> |
|---|---|--|
| Misclassifications | | |
| <p>bird airplane: 28.0%</p>  <p>RGB</p>  <p>the prominence of the blue sky causes model to think bird is an airplane</p> | <p>airplane deer: 24.1%</p>  <p>RGB</p>  <p>green grass causes model to predict deer, thus it was focusing on background instead of object in foreground</p> | <p>truck automobile: 48.6% truck: 45.1%</p>  <p>RGB</p>  <p>classes are very similar, so this is understandable. It sees tires and windows of truck, and has hard time of differentiating finer details of truck and automobile</p> |

As the model is trained more, the heat maps become more interpretable as they begin to highlight some more granular features of the images. At the beginning of training, most are completely black or nonsensical. After a few hundred epochs, it is clear what the model is focusing on.

The various techniques that improve the performance of the model also have an effect on the heatmaps. Regularization helps to keep the models sparse, and only focus on the most important parts of an image. This helps the heat maps to highlight the prominent features that are inherently common to images of a certain class, rather than focusing on some attributes (such as the background) that may be somewhat frequent but could also be a red herring that will cause misclassification of images of different classes. Augmentation helps the heatmaps by giving the model more examples to learn from. This keeps the model from honing in on a particular part or orientation of the image. By seeing the same object parts in different places, the model can learn to pick them out from anywhere in the image. Dropout helps by randomly removing certain connections in the network, so that the model can become more robust. This means that the model will not become too reliant on a particular feature but can generalize to understand more features that make up an object.

From examining the heatmaps, it appears that the model is truly learning how to classify at least some classes, but can still be easily thrown off for others. For example, the model has been seen to focus on the correct things for automobiles and trucks (wheels, windows, profile view) no matter the orientation. However, it appears that the model's interpretation of a deer relies heavily on a green background. This may have to do with the heavy presence of green background in the training set of deers. For the deer case, it may be memorizing the examples but other classes show more signs of true learning.

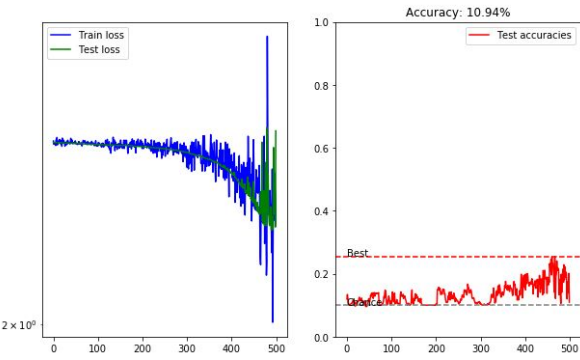
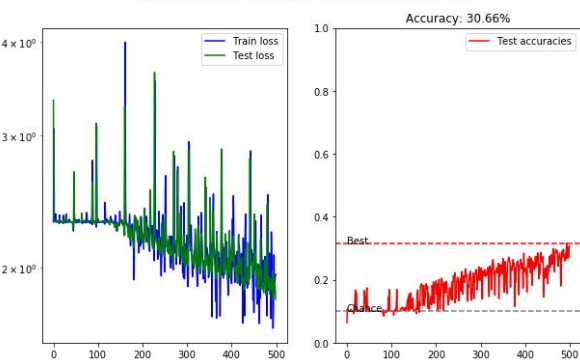
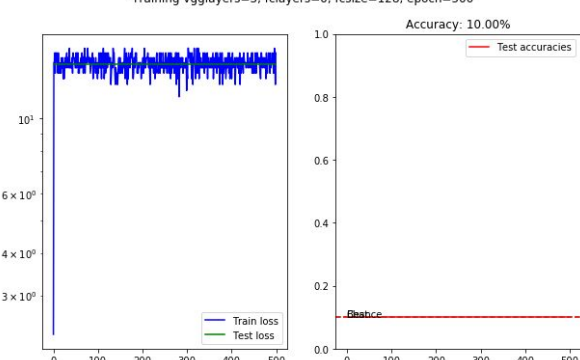
7. Tuning the Optimizer (5 points)

Try different optimizers, e.g. SGD, RMSprop, Adagrad and Adam. See the settings for the “optimizer” variable. Report which optimizers work better given different configurations. For instance, which optimizer seems to work better for deeper networks with more convolutional layers? Do the optimizers learn similar filters? Show plots and filter visualizations to justify your position.

All optimizers were used with no transfer learning, 3 vgg layers, and 500 epochs in order to isolate the effect of optimizer on performance. Three vgg layers seemed to be the deepest number of layers before performance started to deteriorate significantly.

The best optimizer seems to be RMSProp with 31% accuracy. Furthermore, SGD seems to be very unstable as its accuracy goes up in wide swings between each epoch. The optimizers don't learn the same filters; in fact, it seems that the higher the accuracy the clearer the filters are. For example, SGD which had the lowest accuracy had filters with a pretty random pattern, but we can make out edges and image components for filters from other optimizers.

| Optimizer | Performance |
|-----------|--|
| Adam | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 24.08%</p> |

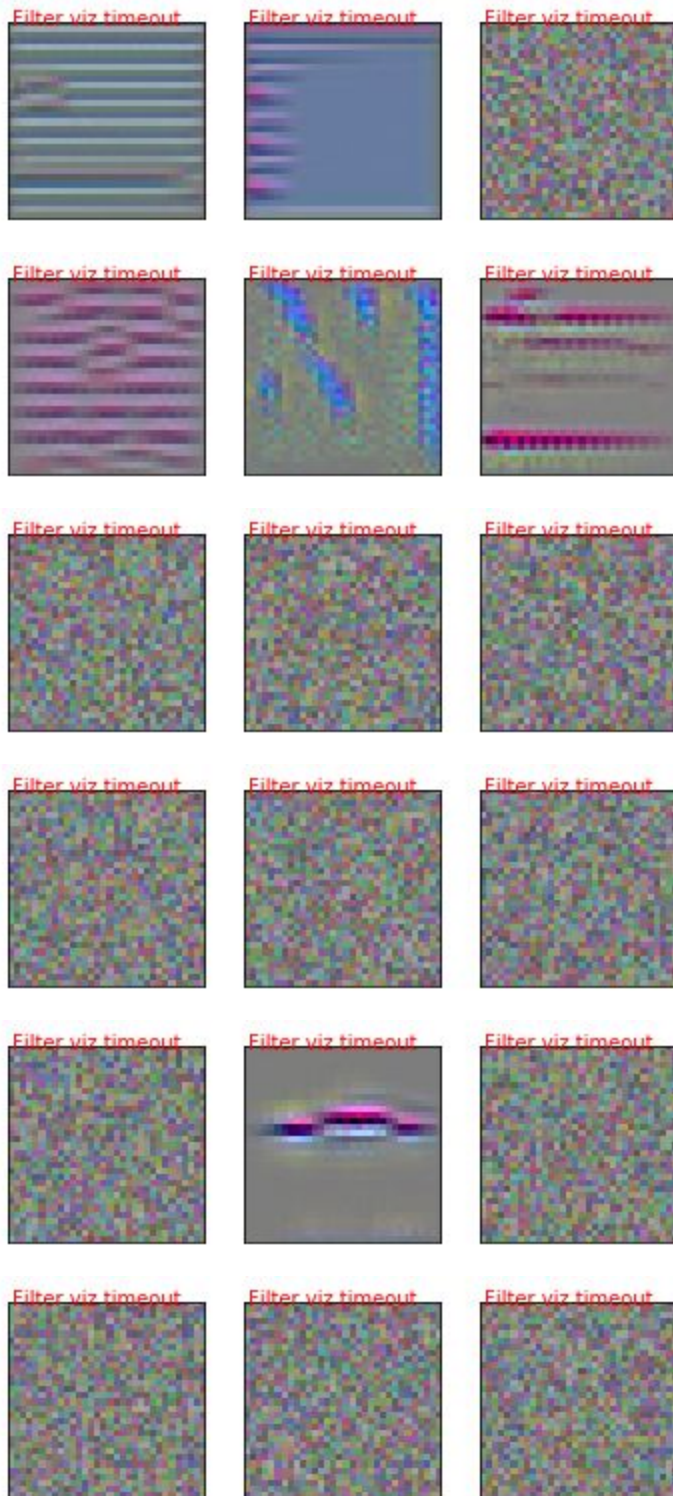
| | |
|----------------|---|
| SGD | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p>  <p>The SGD training plots show a noisy decrease in both train and test loss over 500 epochs, with a final accuracy of 10.94%. The test accuracy plot shows a noisy increase from near zero to approximately 0.25.</p> |
| RMSprop | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p>  <p>The RMSprop training plots show more frequent oscillations in loss compared to SGD, but achieve a higher final accuracy of 30.66%. The test accuracy plot shows a steady increase from near zero to approximately 0.35.</p> |
| Adagrad | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p>  <p>The Adagrad training plots show a very noisy and slow decrease in loss, resulting in a low final accuracy of 10.00%. The test accuracy plot shows a very slow increase from near zero to approximately 0.10.</p> |

Filters

Adam

Some distinct edge detectors appear to be present in the filters used by the first few layers.





SGD

The filters do not appear to show any distinct patterns or edge detectors.

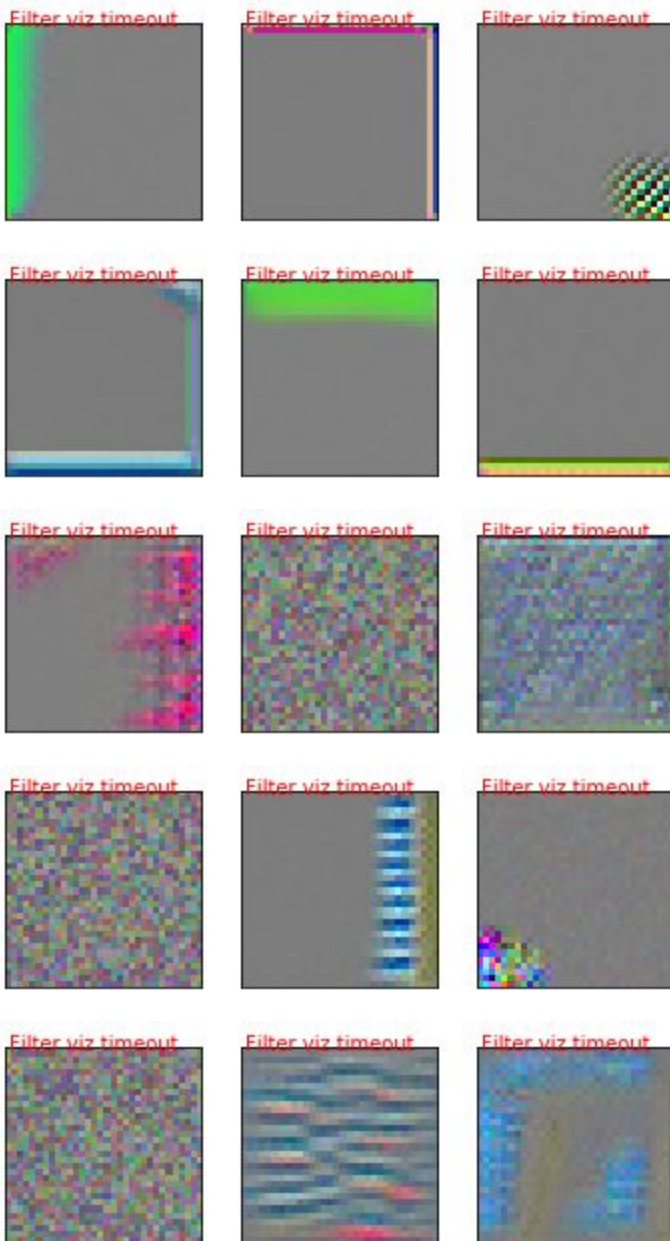
MSIA 490-30 Deep Learning
Spring 2017

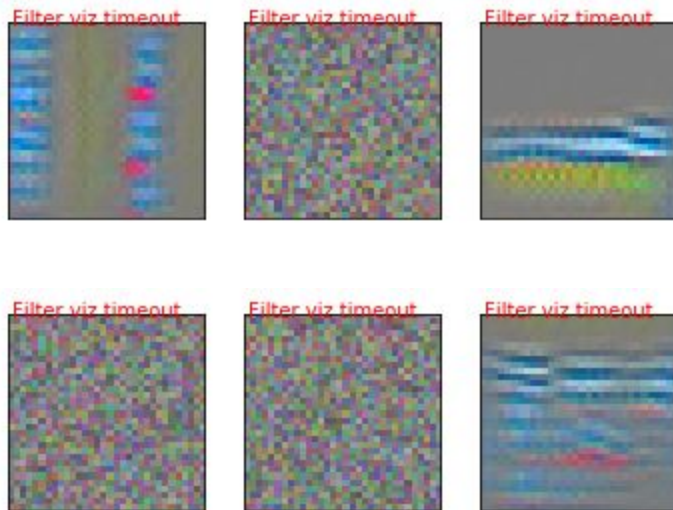




RMSprop

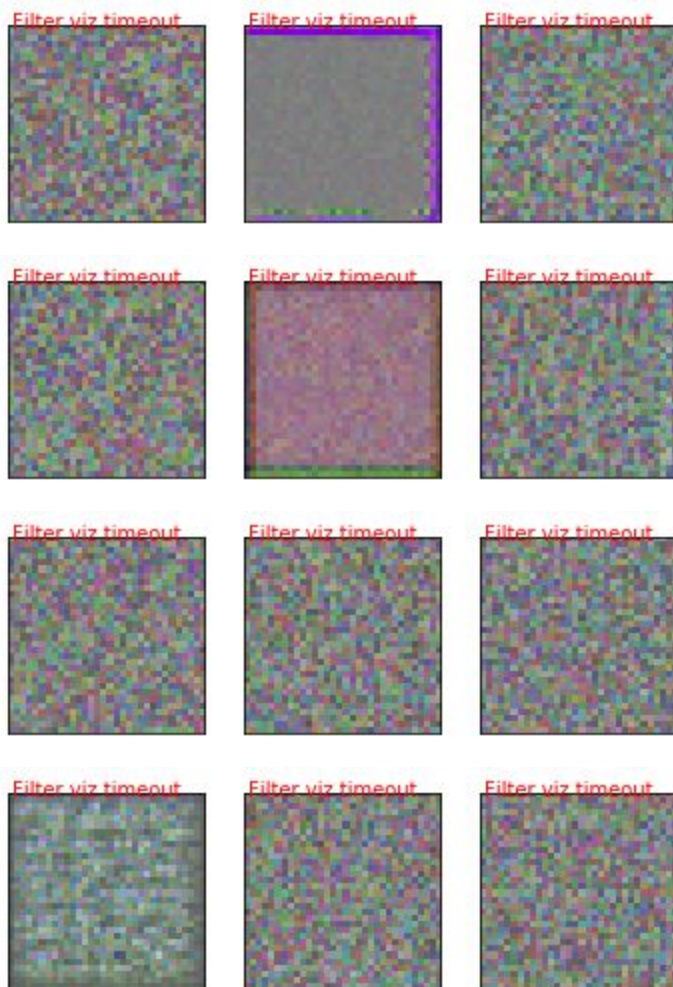
Like with the Adam optimizer, the RMSprop optimizer produces filters with distinct patterns (edge detectors) in the first few layers.

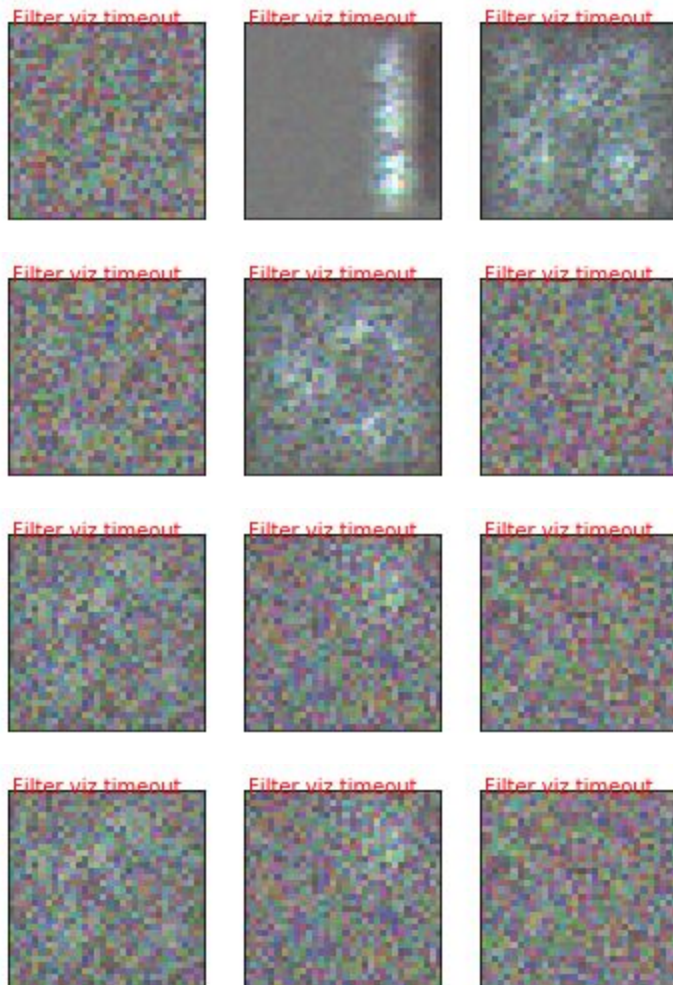




Adagrad

Some of these filters have some rough patterns visible, but not nearly as clean as in the other optimizers.





8. Model Capacity (5 points)

Using the guidelines in class regarding model capacity, justify why your model is sized and regularized accordingly. Recall that a model is too small if it can't overfit the training data, and a model is too large (or not sufficiently regularized) if it can fit random labels easily. Argue whether augmentation or regularization (L1, L2, dropout, early stopping) is better.

Please submit a single Python file with all your code, and use comments to describe the code.

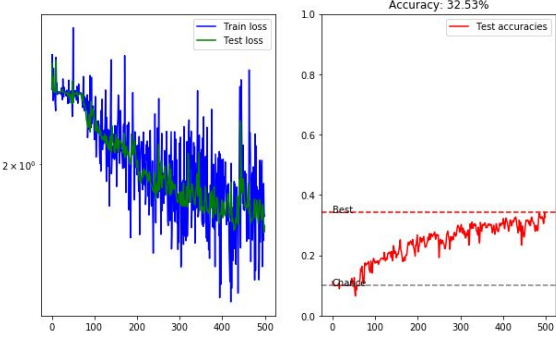
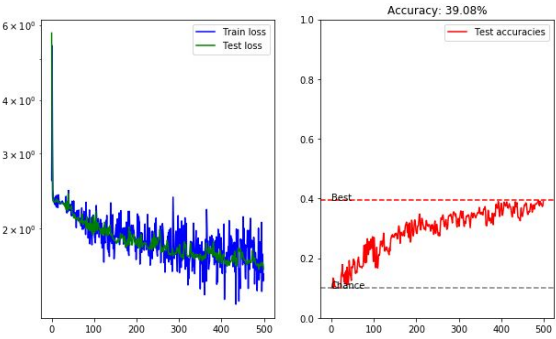
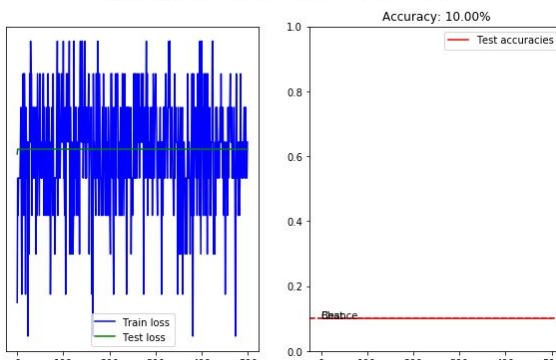
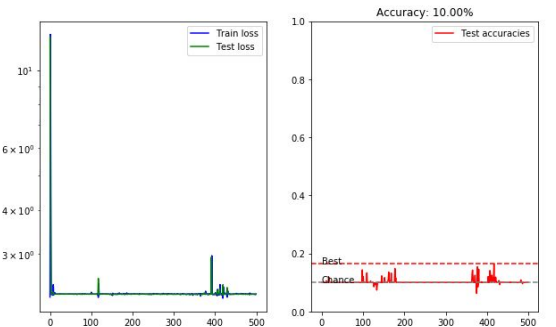
Appendix

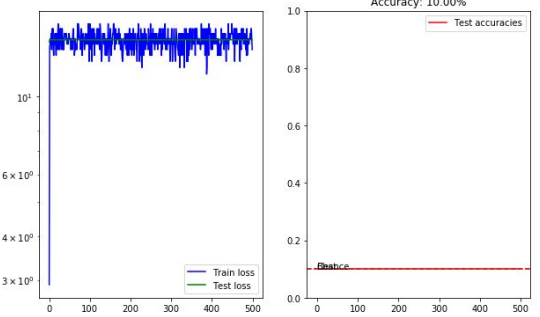
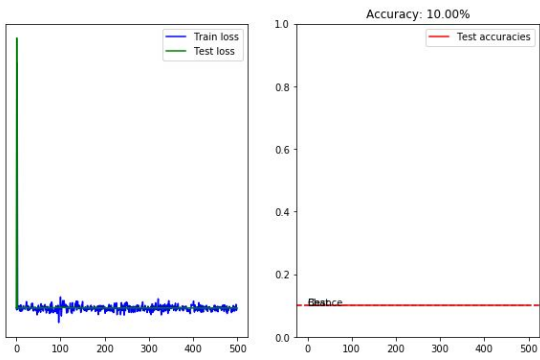
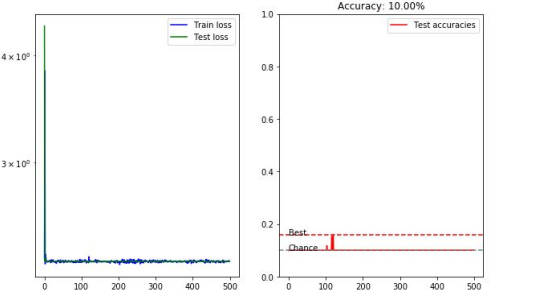
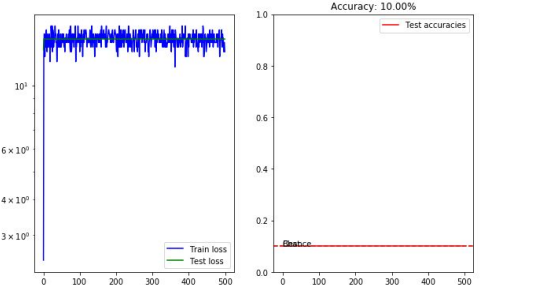
#3 combinations of vgglayers and xferlayers

| | |
|------------------------|-------------------------|
| Vgglayers,xferlearning | Loss and Accuracy Plots |
|------------------------|-------------------------|

| | |
|--|---|
| cfg.vgglayers = 1 cfg.xferlearning = -1 | <p>Training vgglayers=1, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 42.87%</p> <p>The left plot shows Train loss (blue) and Test loss (green) on a scale of 10^3 (y-axis, 0 to 2×10^3) against epoch (x-axis, 0 to 500). Both losses decrease over time, with Train loss being noisier. The right plot shows Test accuracies (red) on a scale of 0.0 to 1.0 against epoch (0 to 500). The accuracy increases from near 0 to approximately 0.43, with a dashed line indicating the 'Best' accuracy at ~0.43 and a 'Chance' level at ~0.1.</p> |
| cfg.vgglayers = 1 cfg.xferlearning = 0 | <p>Training vgglayers=1, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 37.40%</p> <p>The left plot shows Train loss (blue) and Test loss (green) on a scale of 10^3 (y-axis, 0 to 6×10^3) against epoch (0 to 500). The right plot shows Test accuracies (red) on a scale of 0.0 to 1.0 against epoch (0 to 500). The accuracy increases to approximately 0.37, with a dashed line indicating the 'Best' accuracy at ~0.37 and a 'Chance' level at ~0.1.</p> |
| cfg.vgglayers = 1 cfg.xferlearning = 1 | <p>Training vgglayers=1, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 10.00%</p> <p>The left plot shows Train loss (blue) and Test loss (green) on a scale of 10^3 (y-axis, 0 to 10^3) against epoch (0 to 500). The right plot shows Test accuracies (red) on a scale of 0.0 to 1.0 against epoch (0 to 500). The accuracy remains very low, near the 'Chance' level of ~0.1, with a dashed line indicating the 'Best' accuracy at ~0.1.</p> |
| cfg.vgglayers=2, cfg.xferlearning=-1 | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 40.45%</p> <p>The left plot shows Train loss (blue) and Test loss (green) on a scale of 10^3 (y-axis, 0 to 2×10^3) against epoch (0 to 500). The right plot shows Test accuracies (red) on a scale of 0.0 to 1.0 against epoch (0 to 500). The accuracy increases to approximately 0.40, with a dashed line indicating the 'Best' accuracy at ~0.40 and a 'Chance' level at ~0.1.</p> |

| | |
|--|--|
| <i>cfg.vgglayers=2, cfg.xferlearning=0</i> | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 44.54%</p> |
| <i>cfg.vgglayers=2, cfg.xferlearning=1</i> | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 47.34%</p> |
| <i>cfg.vgglayers=2, cfg.xferlearning=3</i> | <p>Training vgglayers=2, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 10.00%</p> |
| <i>cfg.vgglayers=3, cfg.xferlearning=-1</i> | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 20.55%</p> |

| | |
|--|---|
| <p><i>cfg.vgglayers=3,</i> <i>cfg.xferlearning=0</i></p> | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 32.53%</p>  |
| <p><i>cfg.vgglayers=3,</i> <i>cfg.xferlearning=1</i></p> | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 39.08%</p>  |
| <p><i>cfg.vgglayers=3,</i> <i>cfg.xferlearning=2</i></p> | <p>Training vgglayers=3, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 10.00%</p>  |
| <p><i>cfg.vgglayers=4</i> <i>cfg.xferlearning=1</i></p> | <p>Training vgglayers=4, fclayers=0, fcsz=128, epoch=500</p> <p>Accuracy: 10.00%</p>  |

| | |
|--|---|
| <p><i>cfg.vgglayers=4</i> <i>cfg.xferlearning=3</i></p> | <p>Training vgglayers=4, fclayers=0, fcsz=128, epoch=500</p>  |
| <p><i>cfg.vgglayers=5</i> <i>cfg.xferlearning=-1</i></p> | <p>Training vgglayers=5, fclayers=0, fcsz=128, epoch=500</p>  |
| <p><i>cfg.vgglayers=5</i> <i>cfg.xferlearning=2</i></p> | <p>Training vgglayers=5, fclayers=0, fcsz=128, epoch=500</p>  |
| <p><i>cfg.vgglayers=5</i> <i>cfg.xferlearning=5</i></p> | <p>Training vgglayers=5, fclayers=0, fcsz=128, epoch=500</p>  |

cfg.vgglayers=5
cfg.xferlearning=12

