Part A

This part of the assignment uses a sample of the CIFAR-10 dataset:
https://www.cs.toronto.edu/~kriz/cifar.html

The dataset contains RGB colour images of size 32 X 32 and their labels from 0 to 9. You will be using the batch-1 of the dataset, which contains 10,000 training samples. Testing is done on 2,000 test samples. The training data and testing data are provided in files 'data_batch_1' and 'test_batch_trim' files, respectively. Sample code is given in file start_project_2a.py

Design a convolutional neural network consisting of:

• An Input layer of 3x32x32 dimensions

• A convolution layer $C1$ of 50 filters of window size 9x9, VALID padding, and ReLU neurons. A max pooling layer $S1$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.

• A convolution layer $C2$ of 60 filters of window size 5x5, VALID padding, and ReLU neurons. A max pooling layer $S2$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID'.

• A fully connected layer $F3$ of size 300.

• A softmax layer $F4$ of size 10

1. Train the network by using mini-batch gradient descent learning. Set batch size =128, and learning rate α = 0.001. Images should be scaled.

a. Plot the training cost and the test accuracy against learning epochs.

For this question, I tried to run two twice, each time with a different stddev (values below).

First run (last 10 iterations printed):

iter 990: test accuracy 0.404, training loss 0.915385

iter 991: test accuracy 0.3895, training loss 0.904191

iter 992: test accuracy 0.3825, training loss 0.910298

iter 993: test accuracy 0.3765, training loss 0.897333

iter 994: test accuracy 0.367, training loss 0.907967

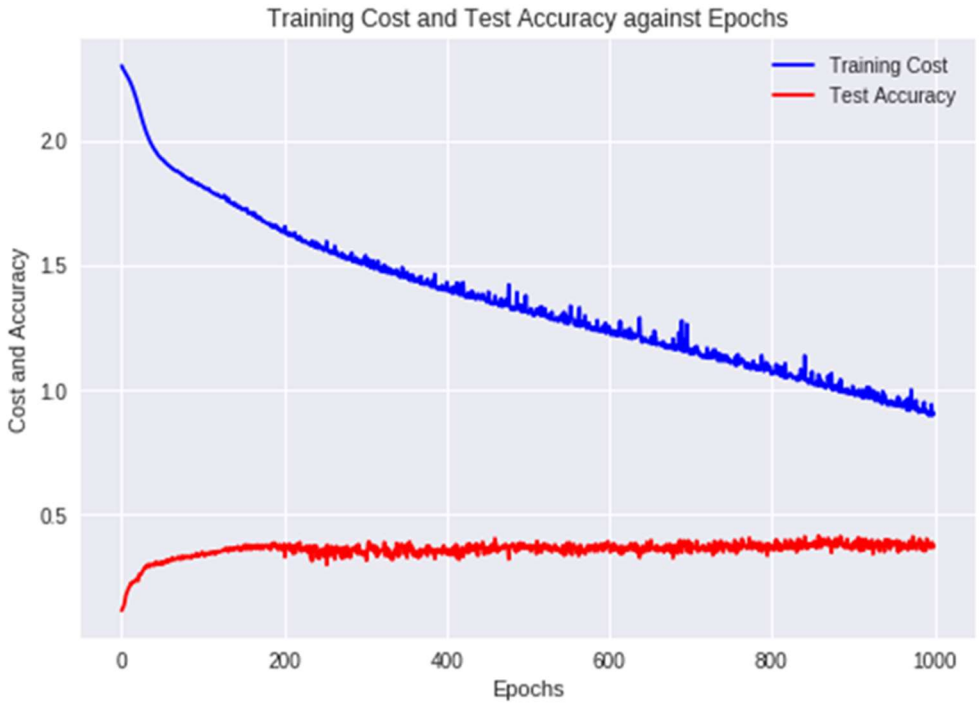
iter 995: test accuracy 0.386, training loss 0.90573

iter 996: test accuracy 0.389, training loss 0.94335

iter 997: test accuracy 0.388, training loss 0.898074

iter 998: test accuracy 0.368, training loss 0.910017

iter 999: test accuracy 0.377, training loss 0.904029



Number of epochs used: 1000

Stddev used for conv1_w => stddev=1.0 / np.sqrt(NUM_CHANNELS * 9 * 9)

Stddev used for conv2_w => stddev=1.0 / np.sqrt(cl1_num * 5 * 5)

Stddev used for full_w => stddev=1.0 / np.sqrt(dim)

Stddev used for softmax_w => 1.0/ np.sqrt(300)

Second run (last ten iterations printed):

iter 990: test accuracy 0.422, training loss 0.535053

iter 991: test accuracy 0.406, training loss 0.524679

iter 992: test accuracy 0.4085, training loss 0.525227

iter 993: test accuracy 0.3995, training loss 0.525629

iter 994: test accuracy 0.3905, training loss 0.512408

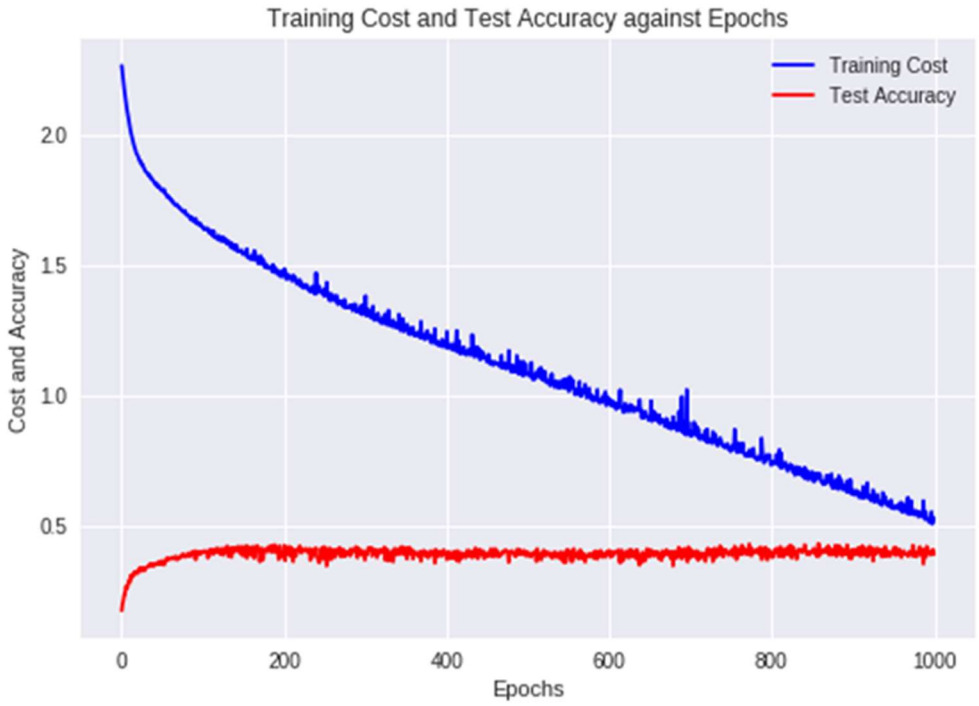
iter 995: test accuracy 0.3895, training loss 0.526694

iter 996: test accuracy 0.404, training loss 0.554735

iter 997: test accuracy 0.4005, training loss 0.50811

iter 998: test accuracy 0.4095, training loss 0.524067

iter 999: test accuracy 0.3915, training loss 0.530446



Number of epochs used: 1000

Stddev used for conv1_w => stddev= np.sqrt(2) / np.sqrt(NUM_CHANNELS * 9 * 9)

Stddev used for conv2_w => stddev= np.sqrt(2) / np.sqrt(cl1_num * 5 * 5)

Stddev used for full_w => stddev= np.sqrt(2) / np.sqrt(dim)

Stddev used for softmax_w => np.sqrt(2) / np.sqrt(300)

Conclusions:

As shown from the two diagrams above, it seems like the test accuracy has not much difference but the training cost for the second run is better. In the second run, the standard deviation of a zero-mean Gaussian distribution is used instead of 1.0.

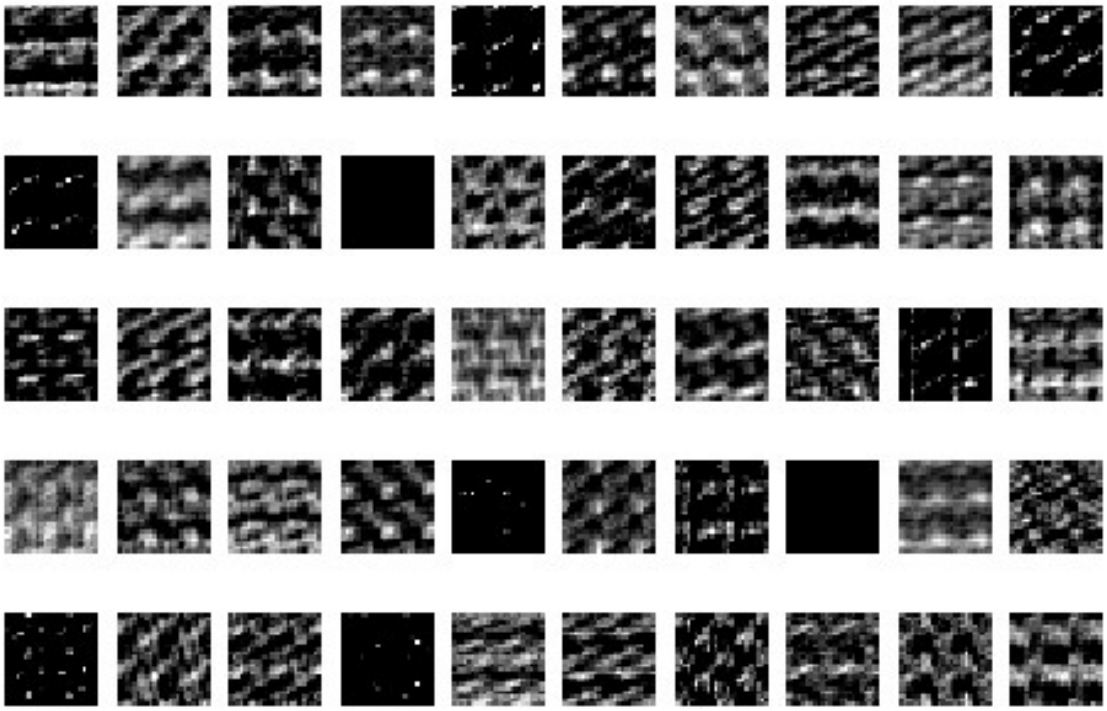
Also, from the diagrams, it appears that after 200 epochs the test accuracy does not seem to improve any more. The test accuracy is around 0.4. This may be because the image itself is of too low-resolution.

If the images are not just of size 32 x 32, perhaps higher resolution images can be filtered or pre-processed first before feeding in to the convolutional neural network.

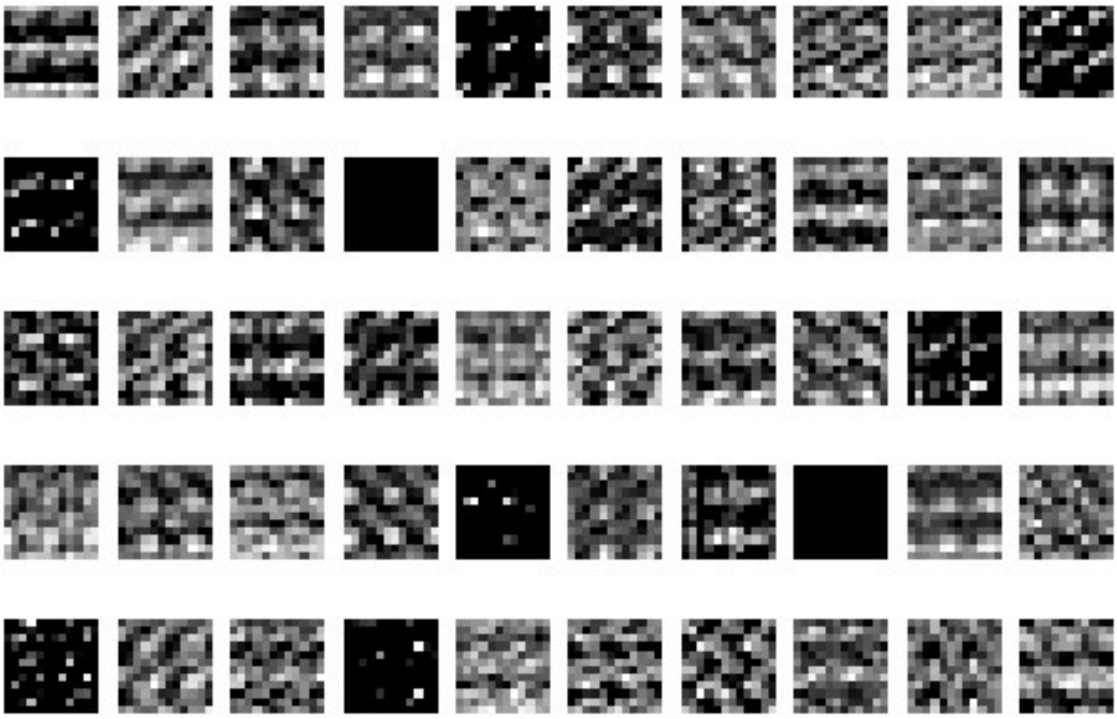
b. For any two test patterns, plot the feature maps at both convolution layers ($C1$ and $C2$) and pooling layers ($S1$ and $S2$) along with the test patterns.

Test pattern 1:

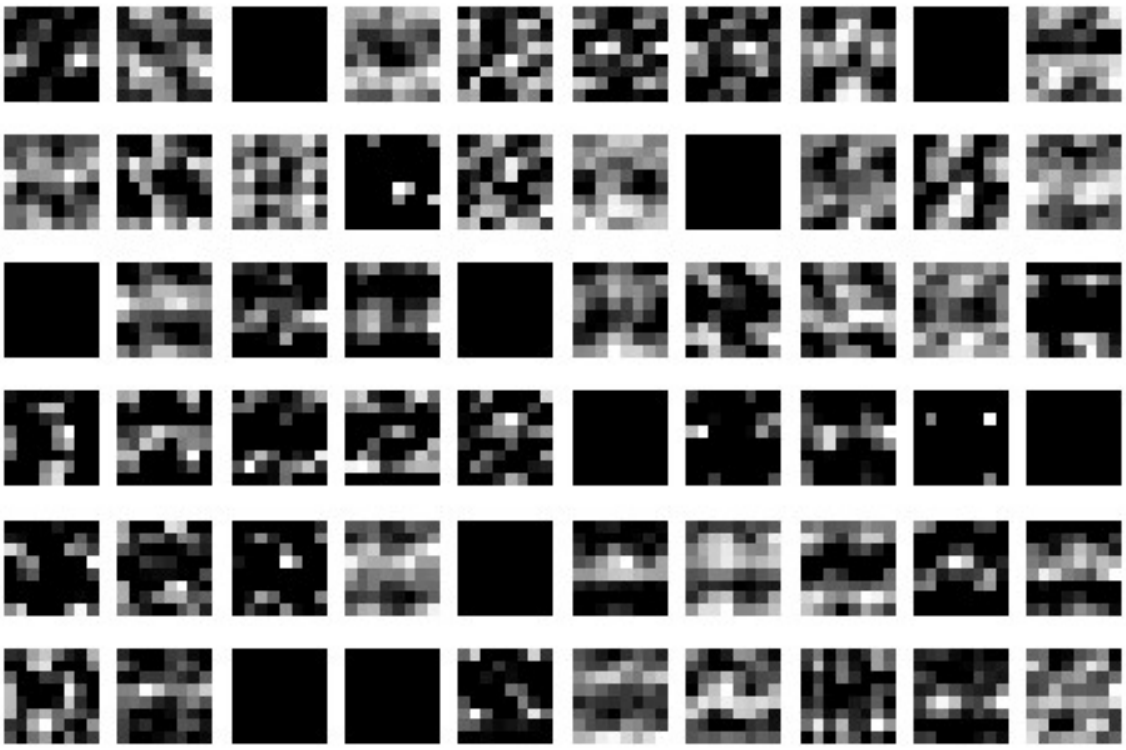
Convolution layer $C1$ of 50 filters of window size 9x9, VALID padding, and ReLU neurons:



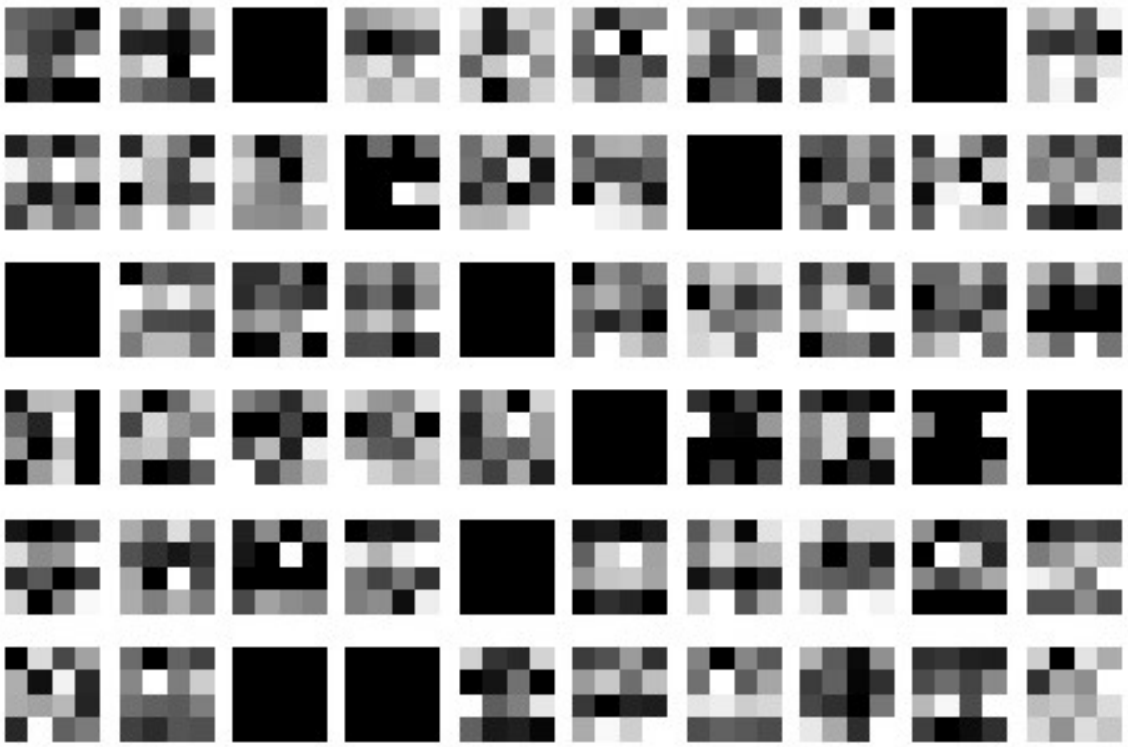
Max pooling layer $S1$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID':



Convolution layer $C2$ of 60 filters of window size 5x5, VALID padding, and ReLU neurons:

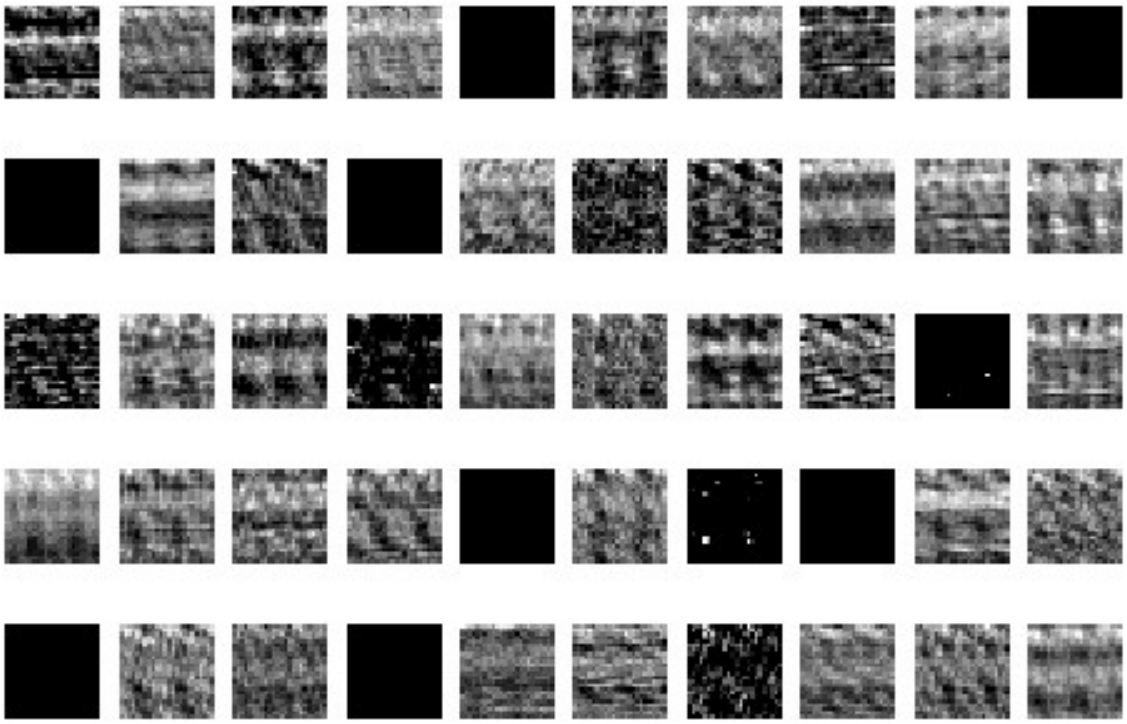


Max pooling layer $S2$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID':

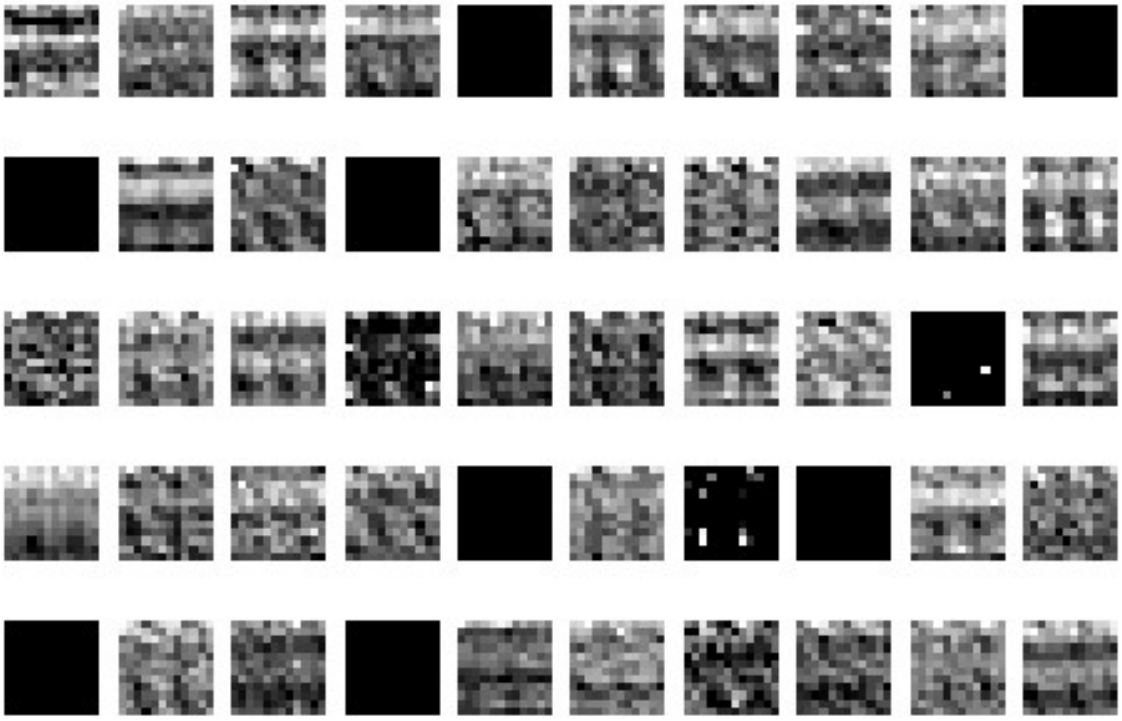


Test pattern 2:

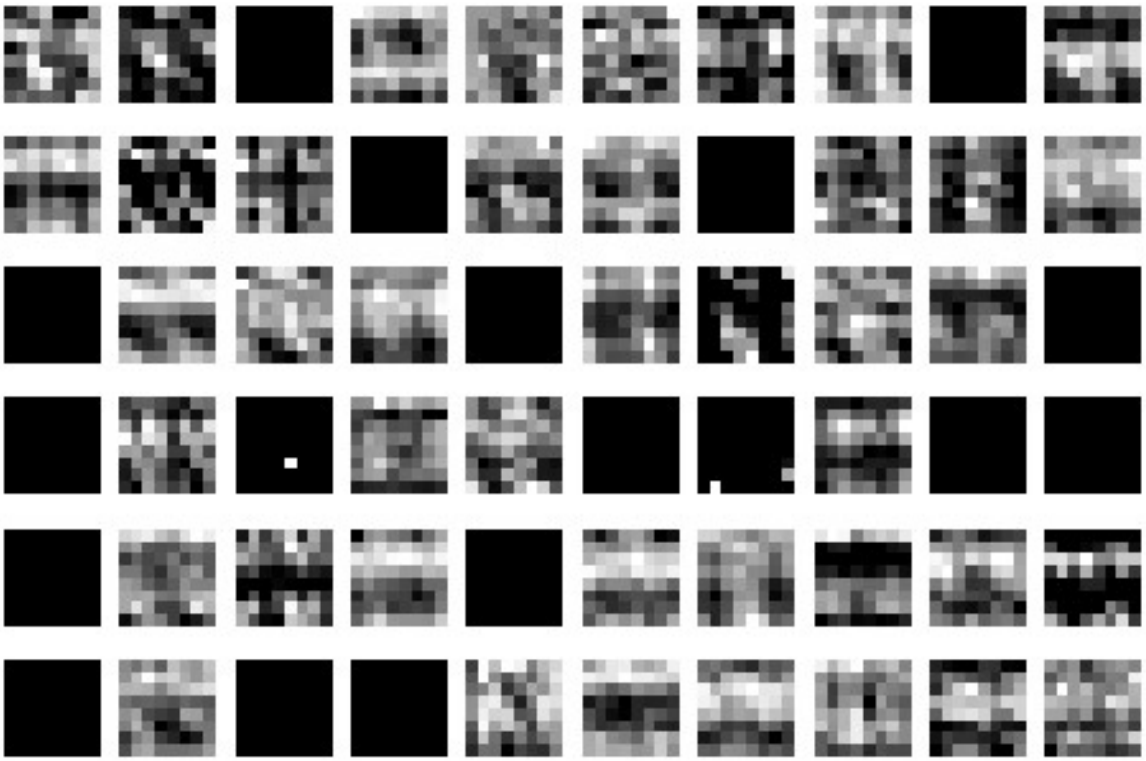
Convolution layer $C1$ of 50 filters of window size 9x9, VALID padding, and ReLU neurons:



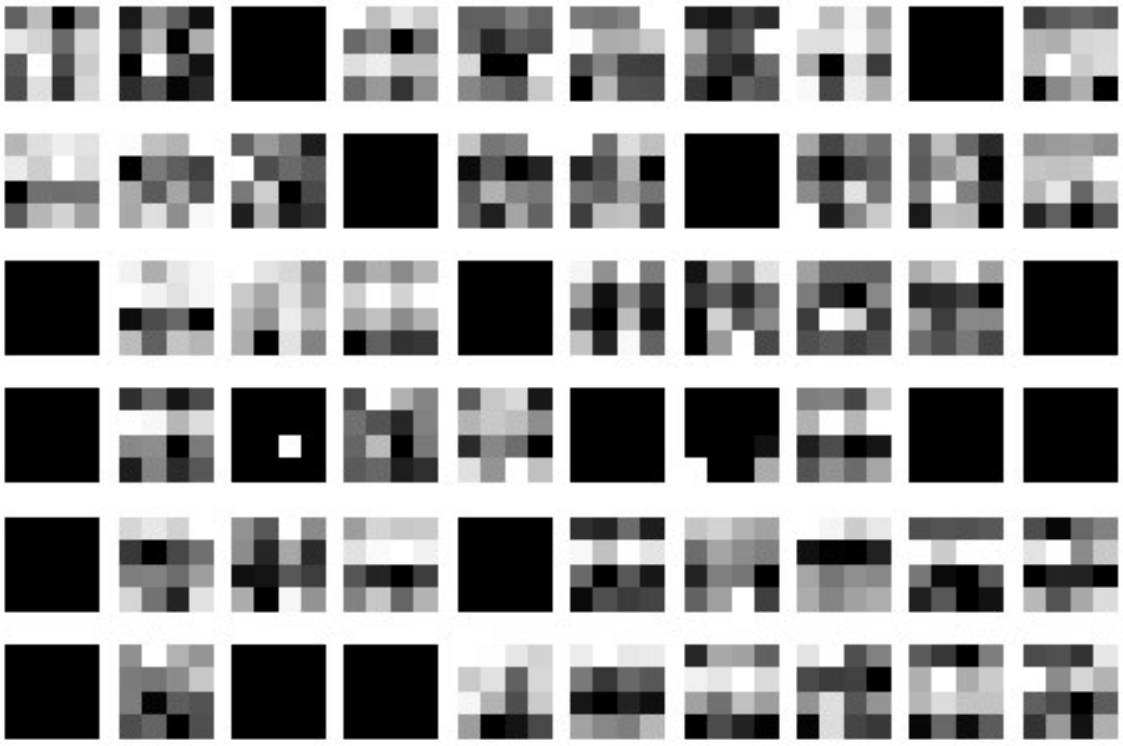
Max pooling layer $S1$ with a pooling window of size 2x2, with stride = 2 and padding = 'VALID':



Convolution layer $C2$ of 60 filters of window size 5x5, VALID padding, and ReLU neurons:



Max pooling layer $S2$with a pooling window of size 2x2, with stride = 2 and padding = 'VALID':



2. Using a grid search, find the optimal numbers of feature maps for part (1) at the convolution layers. Use the test accuracy to determine the optimal number of feature maps.

Grid search is a way of performing hyperparameter optimization, which is an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm.

In this case, I will be doing a broader grid search first to have a better idea of which filters works better, and then narrow it down with more iterations to see which is the most optimal number of feature maps.

Usually the initial layers represent generic features, while the deeper layers represent more detailed features (of the specific dataset that is used for training the model). Hence, for this grid search, I will be using less filters for C1 and more layers for C2.

The richness of possible representations increases because a given layer feeds directly from a layer below to form new kernels by combinations of the features in the layer below. For example, the first hidden layer forms it's kernels by combinations of pixel values from the input, the combination of pixels is much richer than the pixels themselves, while the combination of pixel combinations (second hidden-layer) is even richer. And so on for the consecutive layers.

So the number of filters are incremented so as to be able to properly encode the increasingly richer and richer representations as the signal moves up the representational hierarchy in order to avoid the bottleneck effect.

For the grid search, I will be running:

- C1 with 40 filters and C2 with [60,80,100,120,140] filters
- C1 with 60 filters and C2 with [60,80,100,120,140] filters
- C1 with 80 filters and C2 with [80,100,120,140,160,180,200] filters
- C1 with 100 filters and C2 with [100,120,140,160,180,200] filters

There are most tests for C1 with 80 filters and C2 with 100 filters because they seem to perform better.

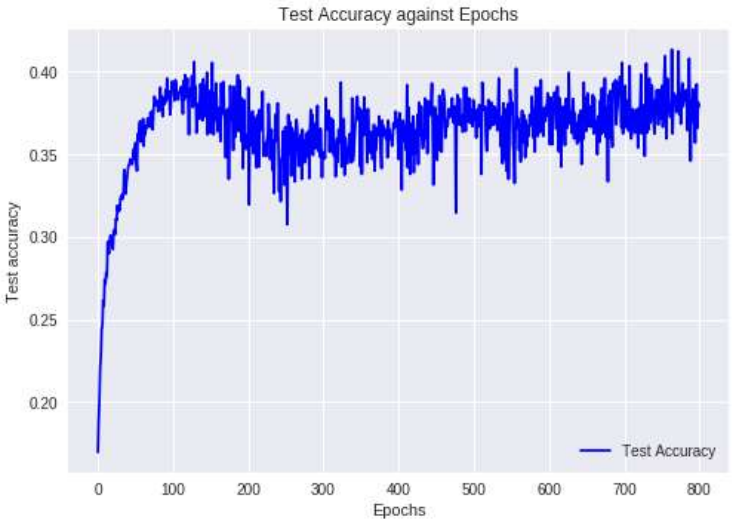
In addition to each graph, the last 10 iteration's test accuracies are included as well.

epochs = 800

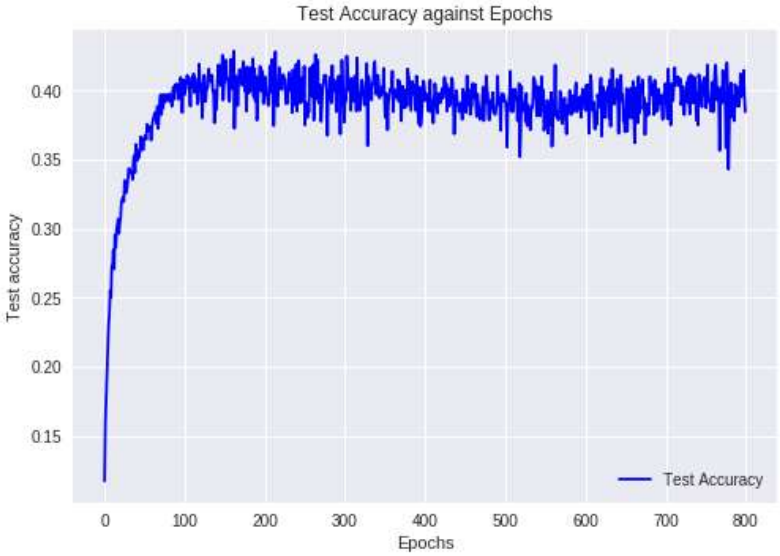
NUM_FEATURE_MAPS1 = [40]

NUM_FEATURE_MAPS2 = [60,80,100,120,140]

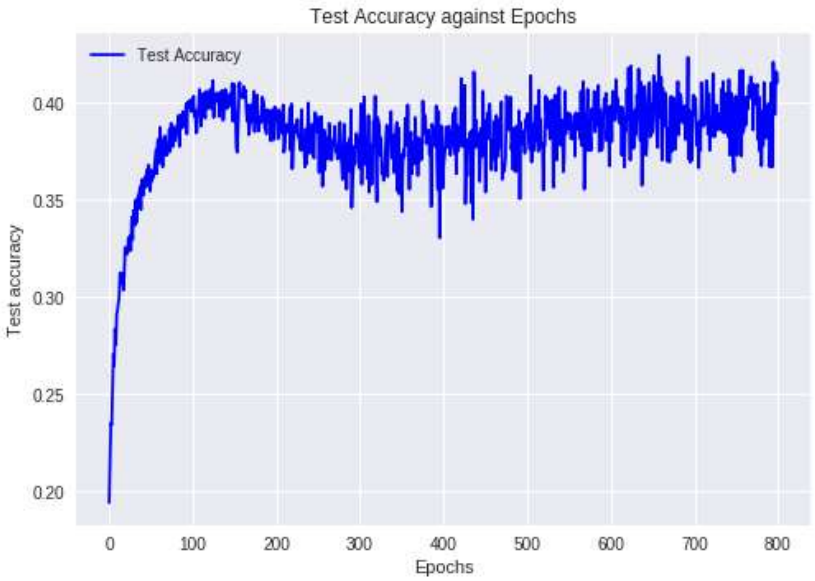
iter 700: test accuracy 0.366
iter 710: test accuracy 0.3625
iter 720: test accuracy 0.3735
iter 730: test accuracy 0.363
iter 740: test accuracy 0.3625
iter 750: test accuracy 0.3915
iter 760: test accuracy 0.3755
iter 770: test accuracy 0.379
iter 780: test accuracy 0.3825
iter 790: test accuracy 0.3695



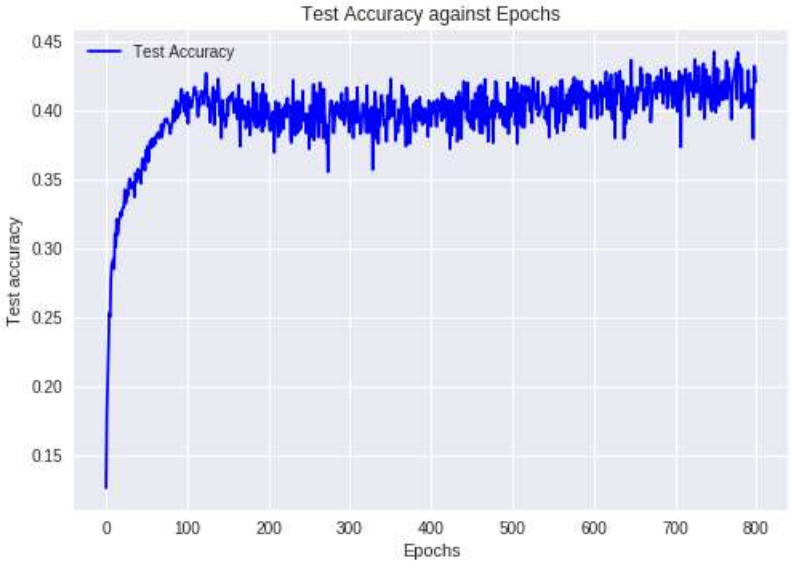
iter 700: test accuracy 0.392
iter 710: test accuracy 0.403
iter 720: test accuracy 0.393
iter 730: test accuracy 0.383
iter 740: test accuracy 0.375
iter 750: test accuracy 0.3875
iter 760: test accuracy 0.4
iter 770: test accuracy 0.391
iter 780: test accuracy 0.3975
iter 790: test accuracy 0.402



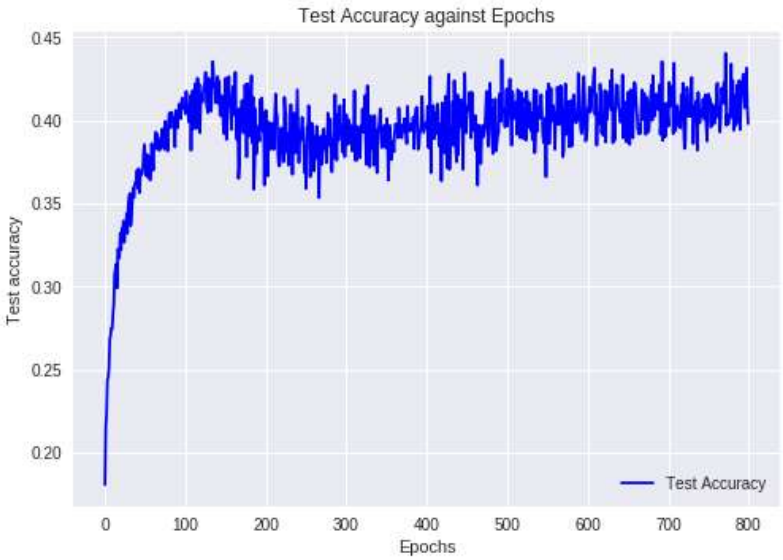
iter 700: test accuracy 0.394
iter 710: test accuracy 0.3935
iter 720: test accuracy 0.3825
iter 730: test accuracy 0.3895
iter 740: test accuracy 0.3825
iter 750: test accuracy 0.4005
iter 760: test accuracy 0.4025
iter 770: test accuracy 0.386
iter 780: test accuracy 0.3675
iter 790: test accuracy 0.367



iter 700: test accuracy 0.4215
iter 710: test accuracy 0.4235
iter 720: test accuracy 0.4215
iter 730: test accuracy 0.427
iter 740: test accuracy 0.4095
iter 750: test accuracy 0.3985
iter 760: test accuracy 0.4215
iter 770: test accuracy 0.411
iter 780: test accuracy 0.433
iter 790: test accuracy 0.429



iter 700: test accuracy 0.3965
iter 710: test accuracy 0.405
iter 720: test accuracy 0.383
iter 730: test accuracy 0.402
iter 740: test accuracy 0.4065
iter 750: test accuracy 0.416
iter 760: test accuracy 0.421
iter 770: test accuracy 0.41
iter 780: test accuracy 0.424
iter 790: test accuracy 0.4075

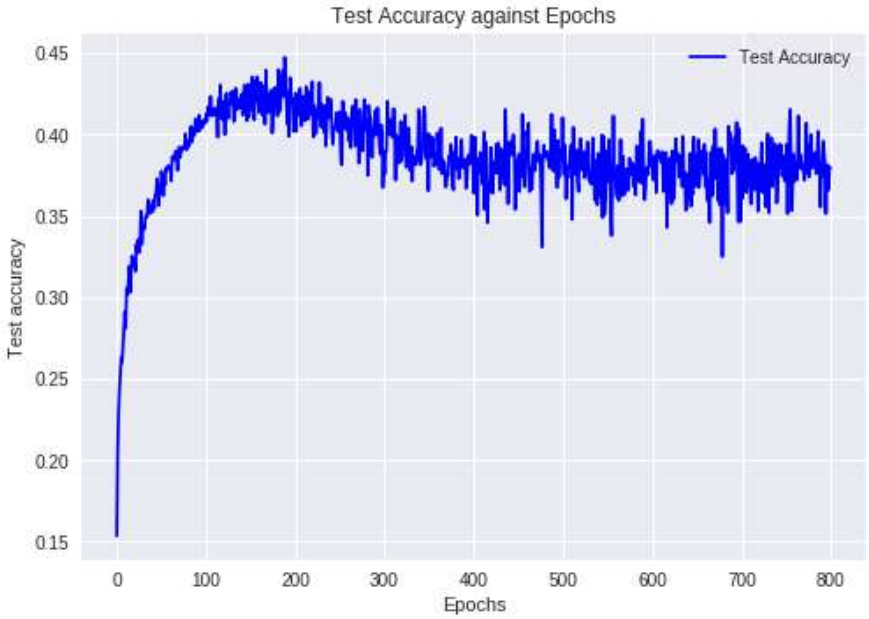


epochs = 800

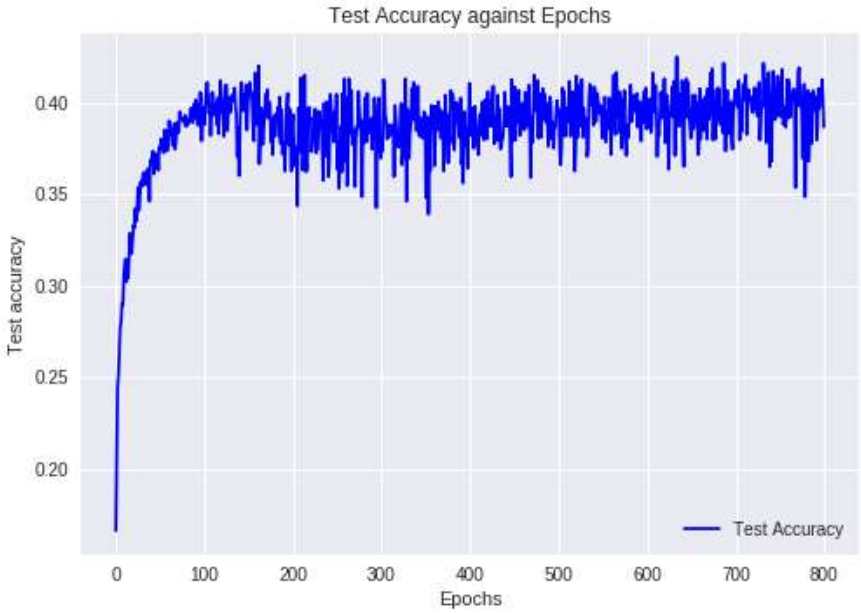
NUM_FEATURE_MAPS1 = [60]

NUM_FEATURE_MAPS2 = [60,80,100,120,140]

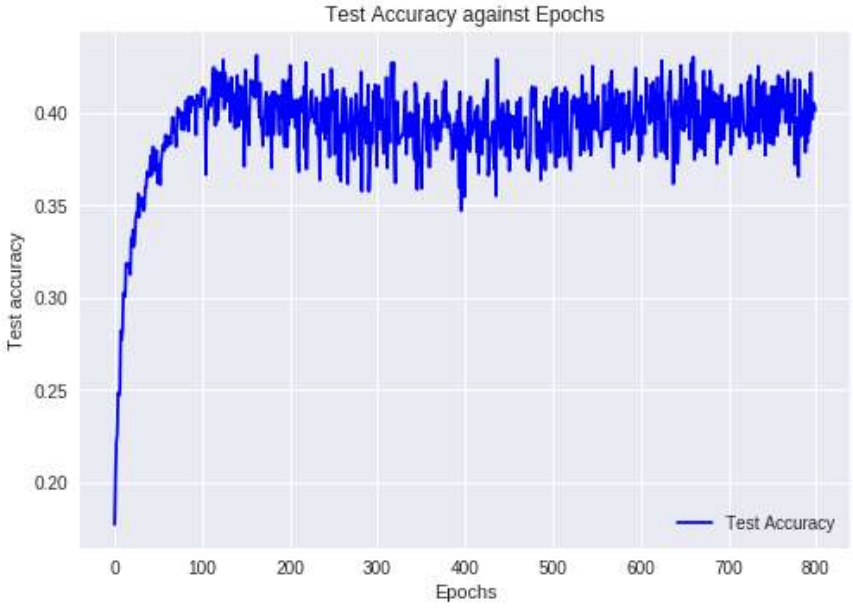
iter 700: test accuracy 0.3605
iter 710: test accuracy 0.3575
iter 720: test accuracy 0.382
iter 730: test accuracy 0.352
iter 740: test accuracy 0.3655
iter 750: test accuracy 0.394
iter 760: test accuracy 0.383
iter 770: test accuracy 0.372
iter 780: test accuracy 0.3825
iter 790: test accuracy 0.374



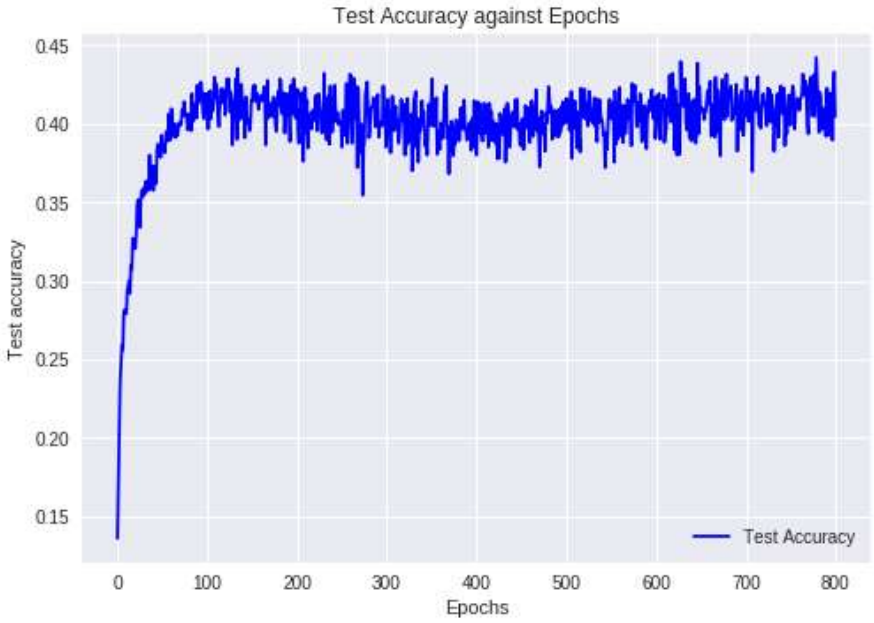
iter 700: test accuracy 0.4065
iter 710: test accuracy 0.4035
iter 720: test accuracy 0.3985
iter 730: test accuracy 0.398
iter 740: test accuracy 0.368
iter 750: test accuracy 0.406
iter 760: test accuracy 0.412
iter 770: test accuracy 0.416
iter 780: test accuracy 0.399
iter 790: test accuracy 0.3965



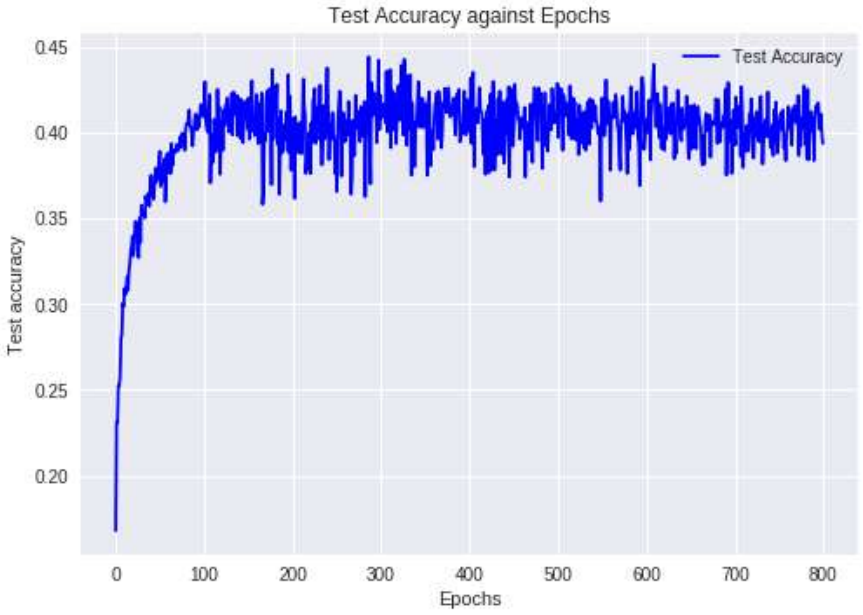
iter 700: test accuracy 0.406
iter 710: test accuracy 0.403
iter 720: test accuracy 0.3905
iter 730: test accuracy 0.4025
iter 740: test accuracy 0.3895
iter 750: test accuracy 0.414
iter 760: test accuracy 0.4075
iter 770: test accuracy 0.388
iter 780: test accuracy 0.3655
iter 790: test accuracy 0.384



iter 700: test accuracy 0.4295
iter 710: test accuracy 0.4205
iter 720: test accuracy 0.4125
iter 730: test accuracy 0.409
iter 740: test accuracy 0.4065
iter 750: test accuracy 0.415
iter 760: test accuracy 0.415
iter 770: test accuracy 0.4225
iter 780: test accuracy 0.411
iter 790: test accuracy 0.4225



iter 700: test accuracy 0.3925
iter 710: test accuracy 0.3985
iter 720: test accuracy 0.386
iter 730: test accuracy 0.401
iter 740: test accuracy 0.391
iter 750: test accuracy 0.4065
iter 760: test accuracy 0.415
iter 770: test accuracy 0.405
iter 780: test accuracy 0.41
iter 790: test accuracy 0.4065



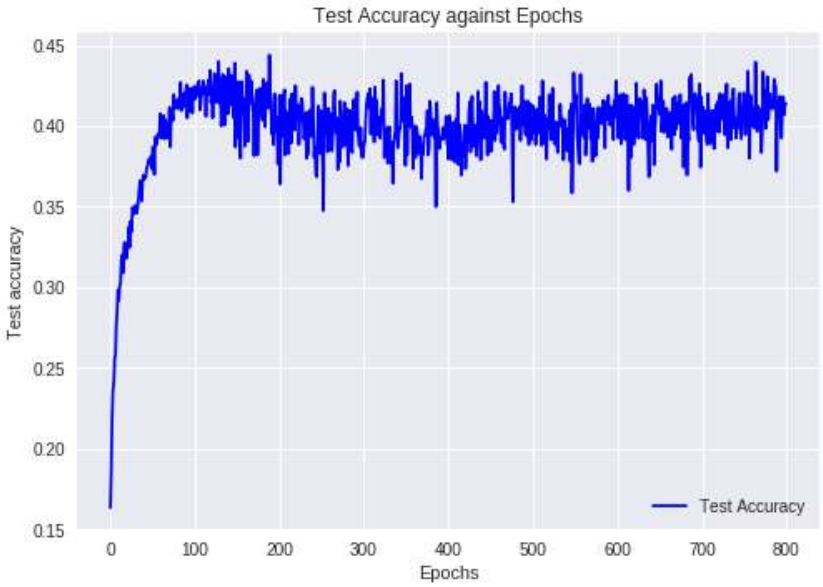
epochs = 800

NUM_FEATURE_MAPS1 = [80]

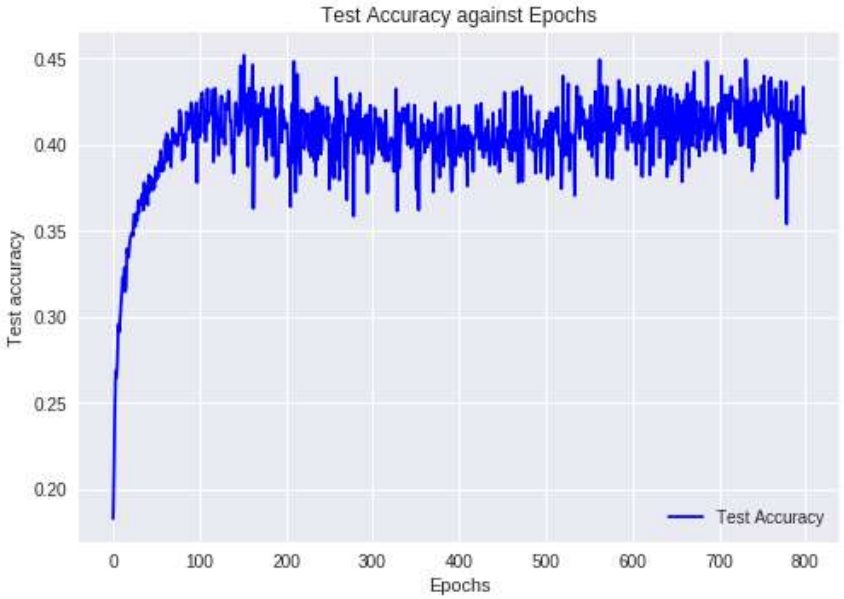
NUM_FEATURE_MAPS2 = [80,100,120,140,160,180]

iter 700: test accuracy 0.3955
iter 710: test accuracy 0.389
iter 720: test accuracy 0.396
iter 730: test accuracy 0.3865

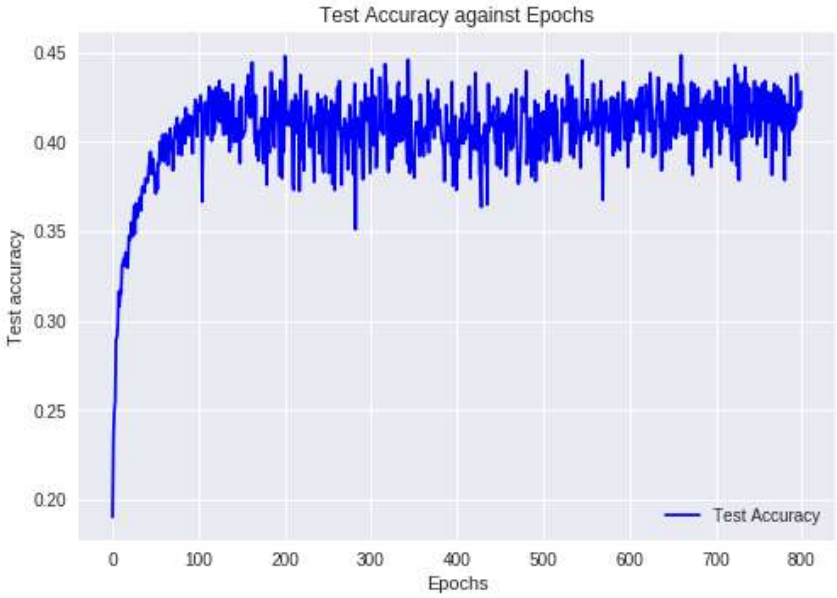
iter 740: test accuracy 0.3925
iter 750: test accuracy 0.4155
iter 760: test accuracy 0.411
iter 770: test accuracy 0.3835
iter 780: test accuracy 0.42
iter 790: test accuracy 0.412



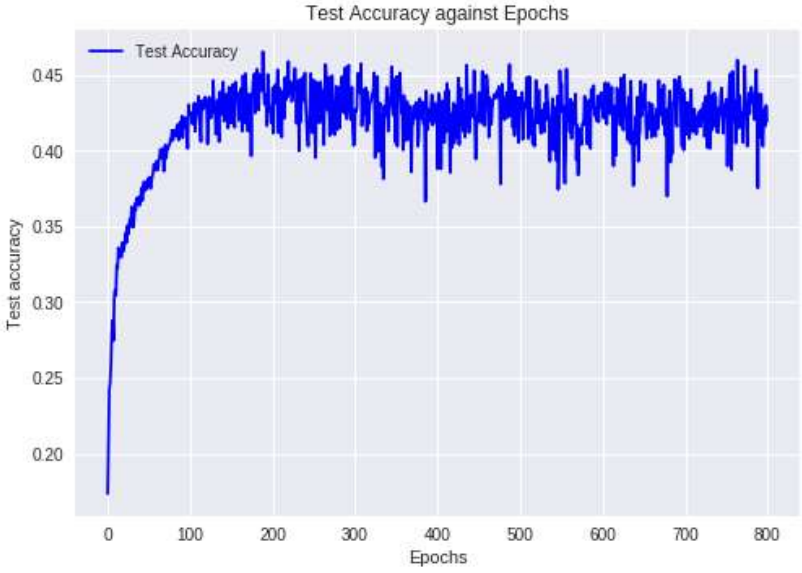
iter 700: test accuracy 0.424
iter 710: test accuracy 0.4215
iter 720: test accuracy 0.4225
iter 730: test accuracy 0.411
iter 740: test accuracy 0.39
iter 750: test accuracy 0.4205
iter 760: test accuracy 0.439
iter 770: test accuracy 0.4145
iter 780: test accuracy 0.419
iter 790: test accuracy 0.428



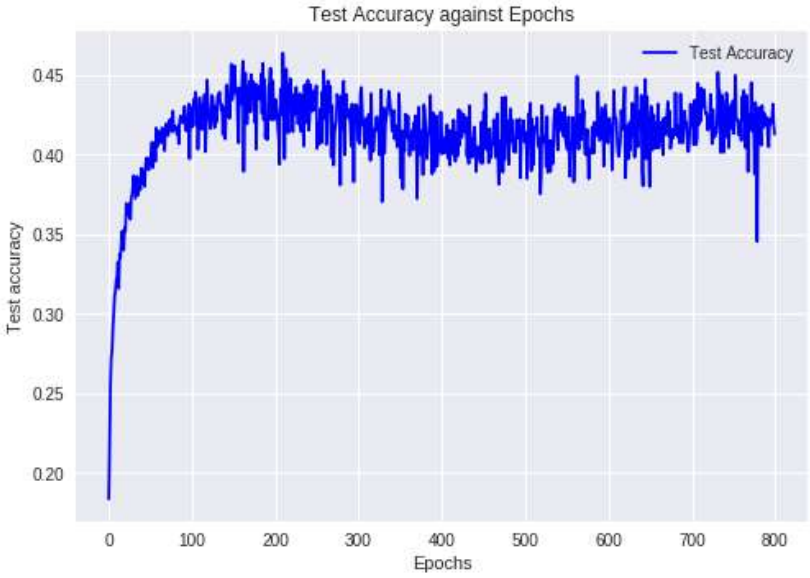
iter 700: test accuracy 0.434
iter 710: test accuracy 0.4305
iter 720: test accuracy 0.409
iter 730: test accuracy 0.4185
iter 740: test accuracy 0.407
iter 750: test accuracy 0.425
iter 760: test accuracy 0.428
iter 770: test accuracy 0.3955
iter 780: test accuracy 0.3785
iter 790: test accuracy 0.408



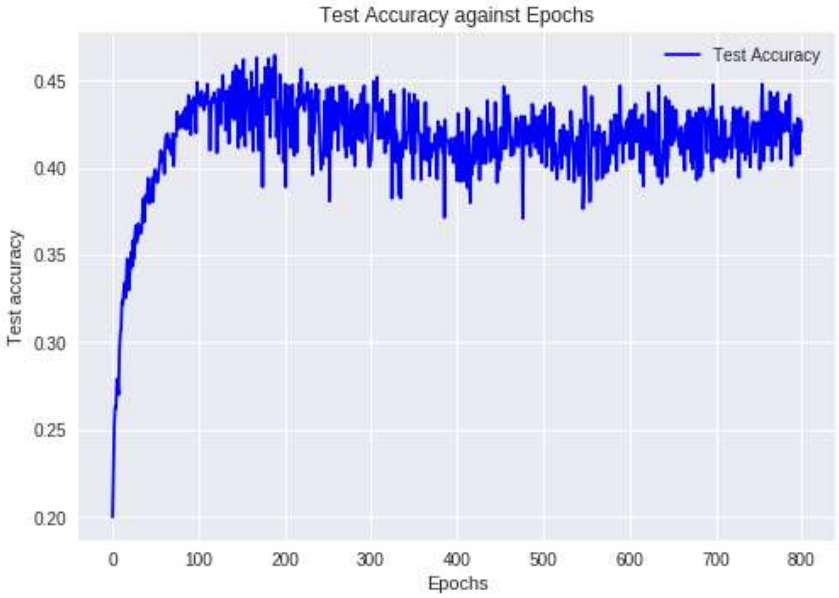
iter 700: test accuracy 0.42
iter 710: test accuracy 0.416
iter 720: test accuracy 0.42
iter 730: test accuracy 0.4015
iter 740: test accuracy 0.412
iter 750: test accuracy 0.434
iter 760: test accuracy 0.4325
iter 770: test accuracy 0.4055
iter 780: test accuracy 0.442
iter 790: test accuracy 0.409



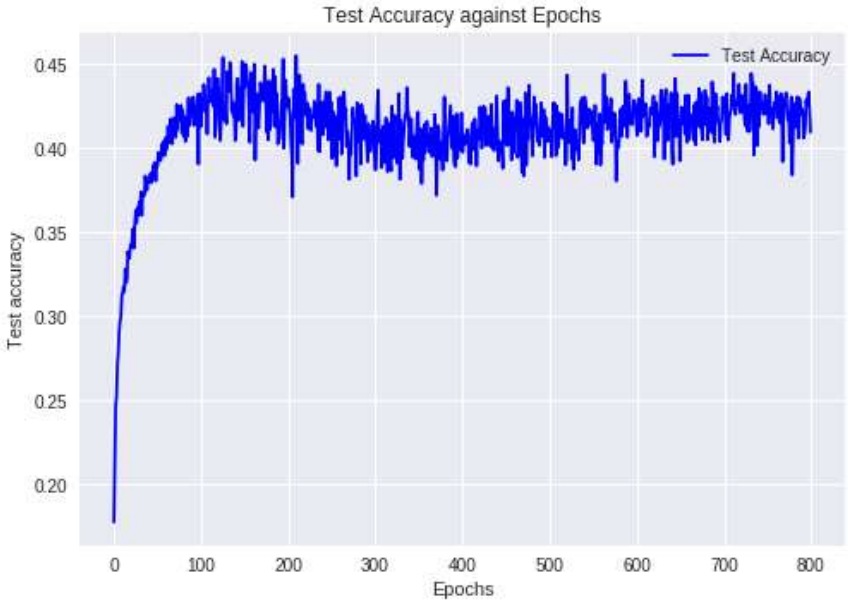
iter 700: test accuracy 0.427
iter 710: test accuracy 0.421
iter 720: test accuracy 0.4235
iter 730: test accuracy 0.419
iter 740: test accuracy 0.411
iter 750: test accuracy 0.4135
iter 760: test accuracy 0.4365
iter 770: test accuracy 0.4285
iter 780: test accuracy 0.4295
iter 790: test accuracy 0.4235



iter 700: test accuracy 0.406
iter 710: test accuracy 0.4035
iter 720: test accuracy 0.425
iter 730: test accuracy 0.4095
iter 740: test accuracy 0.4005
iter 750: test accuracy 0.422
iter 760: test accuracy 0.432
iter 770: test accuracy 0.4125
iter 780: test accuracy 0.4245
iter 790: test accuracy 0.4115



iter 700: test accuracy 0.43
iter 710: test accuracy 0.426
iter 720: test accuracy 0.4315
iter 730: test accuracy 0.4185
iter 740: test accuracy 0.413
iter 750: test accuracy 0.428
iter 760: test accuracy 0.4335
iter 770: test accuracy 0.43
iter 780: test accuracy 0.4265
iter 790: test accuracy 0.4225

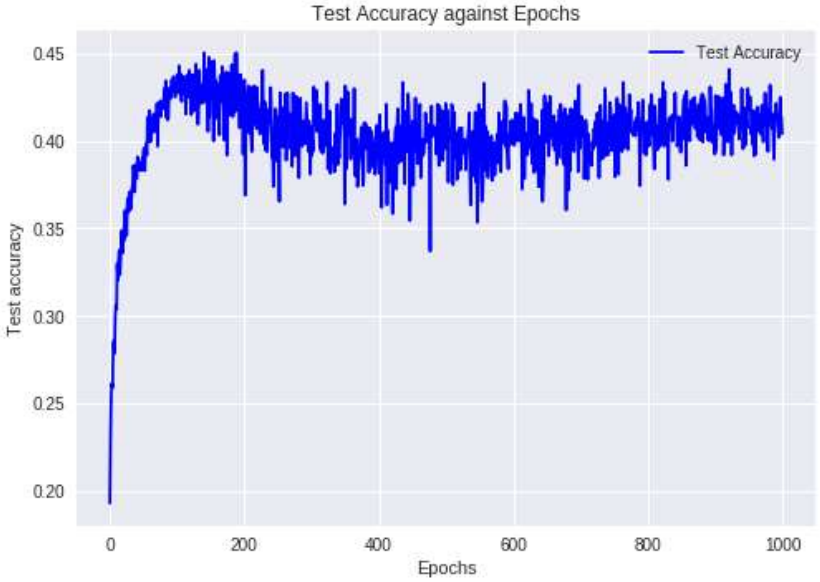


epochs = 1000

NUM_FEATURE_MAPS1 = [100]

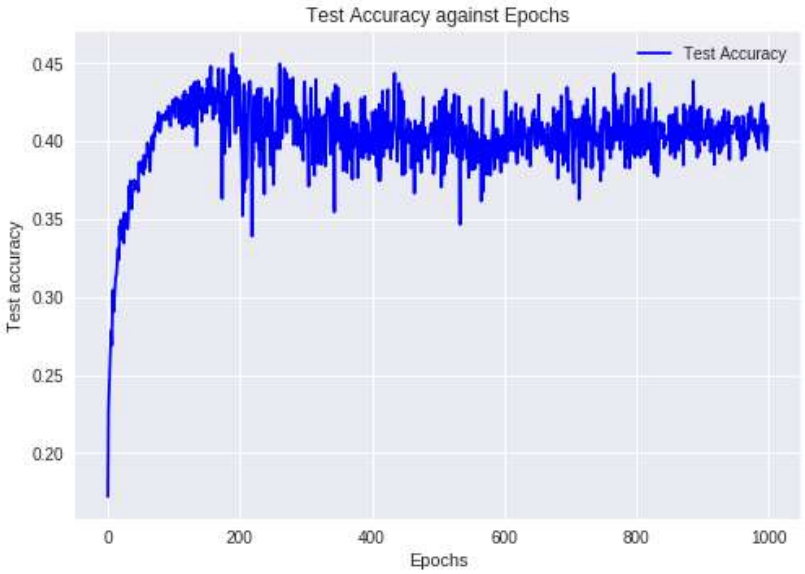
NUM_FEATURE_MAPS2 = [100,120,140,160,180,200]

iter 900: test accuracy 0.426
iter 910: test accuracy 0.421
iter 920: test accuracy 0.441
iter 930: test accuracy 0.417
iter 940: test accuracy 0.417
iter 950: test accuracy 0.418
iter 960: test accuracy 0.4065
iter 970: test accuracy 0.406
iter 980: test accuracy 0.4215
iter 990: test accuracy 0.4215

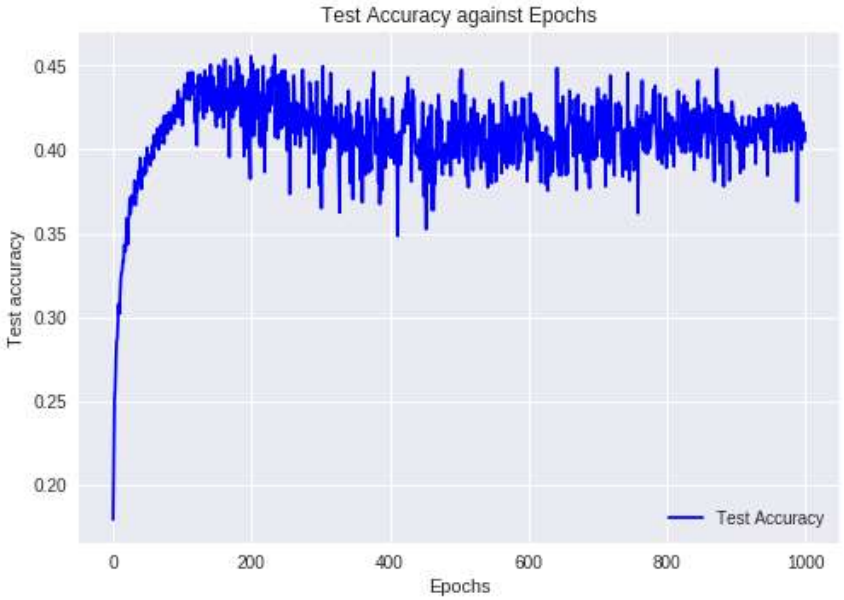


iter 900: test accuracy 0.402
iter 910: test accuracy 0.407
iter 920: test accuracy 0.4095
iter 930: test accuracy 0.4155

iter 940: test accuracy 0.4085
iter 950: test accuracy 0.4105
iter 960: test accuracy 0.3955
iter 970: test accuracy 0.402
iter 980: test accuracy 0.403
iter 990: test accuracy 0.4065
<matplotlib.figure.Figure at 0x7f7a7a26ba90>

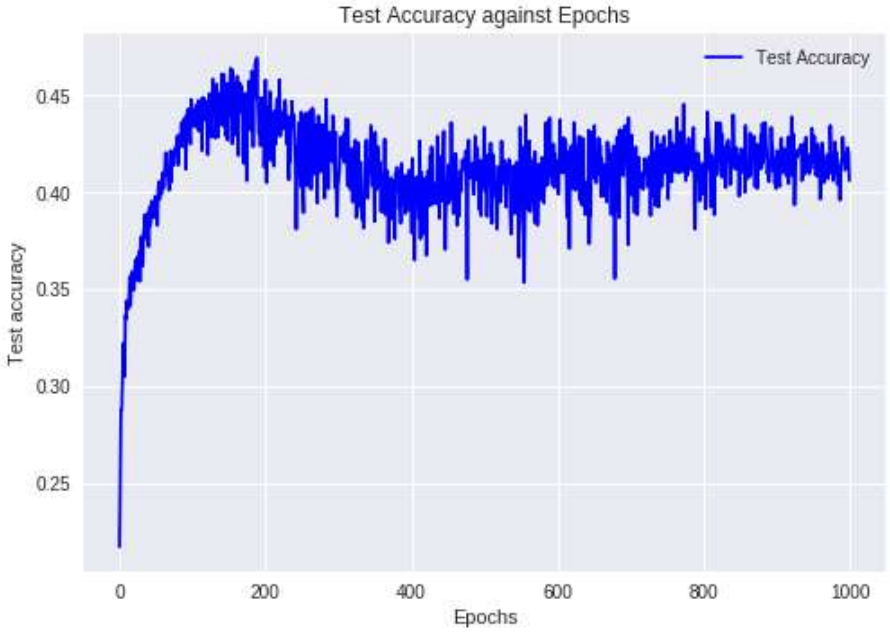


iter 900: test accuracy 0.414
iter 910: test accuracy 0.3885
iter 920: test accuracy 0.4045
iter 930: test accuracy 0.418
iter 940: test accuracy 0.4
iter 950: test accuracy 0.426
iter 960: test accuracy 0.427
iter 970: test accuracy 0.4125
iter 980: test accuracy 0.3995
iter 990: test accuracy 0.406
<matplotlib.figure.Figure at 0x7f7a7a2509b0>

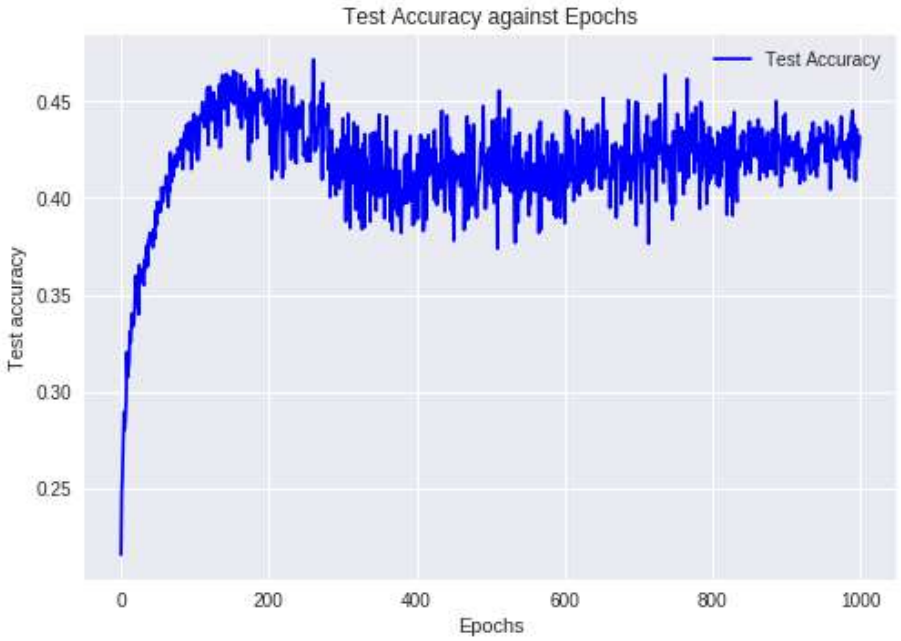


iter 900: test accuracy 0.4215
iter 910: test accuracy 0.4225
iter 920: test accuracy 0.439
iter 930: test accuracy 0.421
iter 940: test accuracy 0.4285

iter 950: test accuracy 0.4215
iter 960: test accuracy 0.4045
iter 970: test accuracy 0.407
iter 980: test accuracy 0.42
iter 990: test accuracy 0.4285

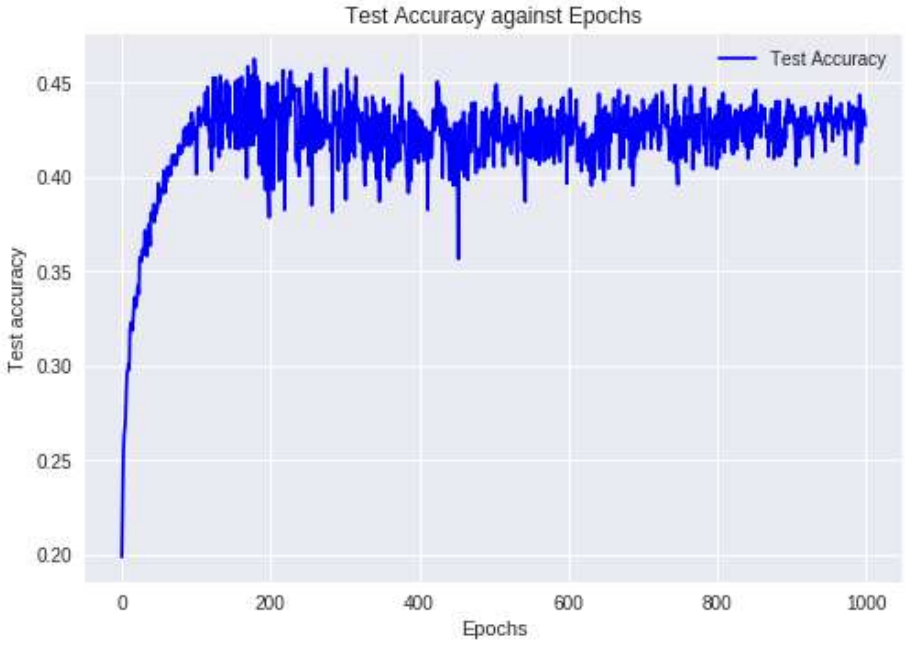


iter 900: test accuracy 0.419
iter 910: test accuracy 0.429
iter 920: test accuracy 0.4255
iter 930: test accuracy 0.4285
iter 940: test accuracy 0.431
iter 950: test accuracy 0.4305
iter 960: test accuracy 0.4185
iter 970: test accuracy 0.4245
iter 980: test accuracy 0.4155
iter 990: test accuracy 0.425

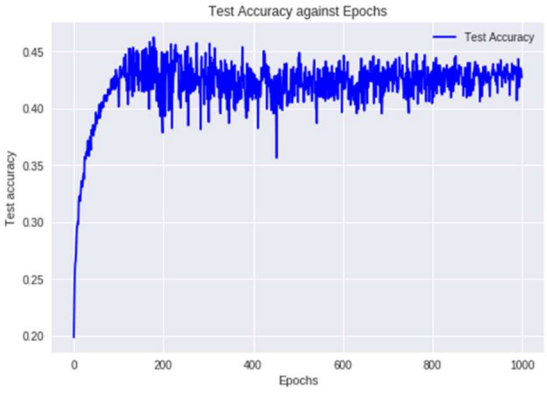


iter 900: test accuracy 0.4355
iter 910: test accuracy 0.4105
iter 920: test accuracy 0.4225

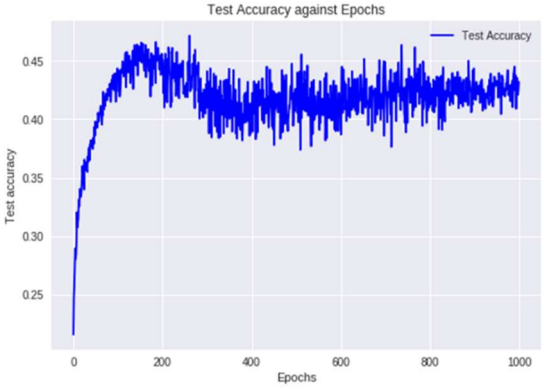
iter 930: test accuracy 0.4275
iter 940: test accuracy 0.4255
iter 950: test accuracy 0.43
iter 960: test accuracy 0.4365
iter 970: test accuracy 0.4285
iter 980: test accuracy 0.425
iter 990: test accuracy 0.4275



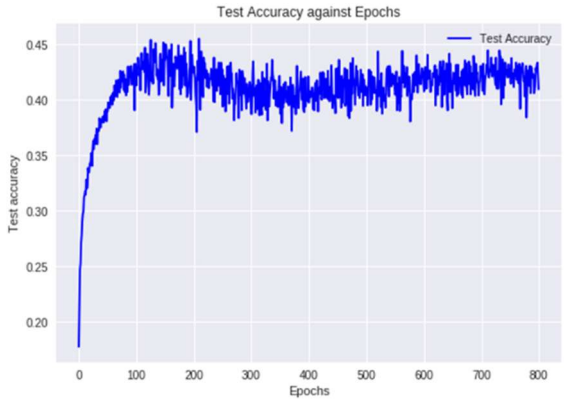
After getting these results, here are some of the **<u>top few results</u>**:



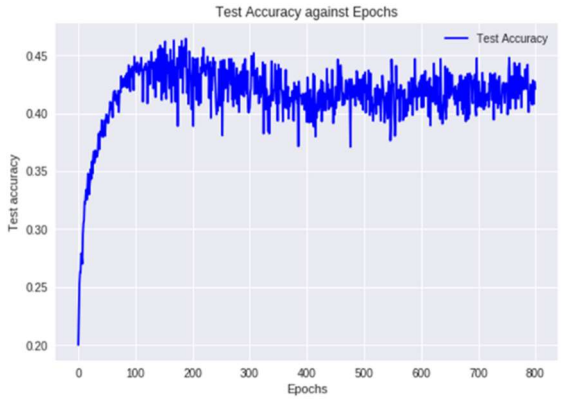
(C1=100, C2=200)



(C1=100, C2=180)



(C1=80, C2=200)



(C1=80, C2=180)

C1=100 filters, C2=200 filters is selected as the most optimal number of feature maps at the convolution layers.

3. Using the optimal number of filters found in part (2), train the network by:

a. Adding the momentum term with momentum γ = 0.1.
b. Using RMSProp algorithm for learning
c. Using Adam optimizer for learning
d. Adding dropout to the layers

Plot the training costs and test accuracies against epochs for each case.

3a. Adding the momentum term with momentum γ = 0.1.

C1=100 filters

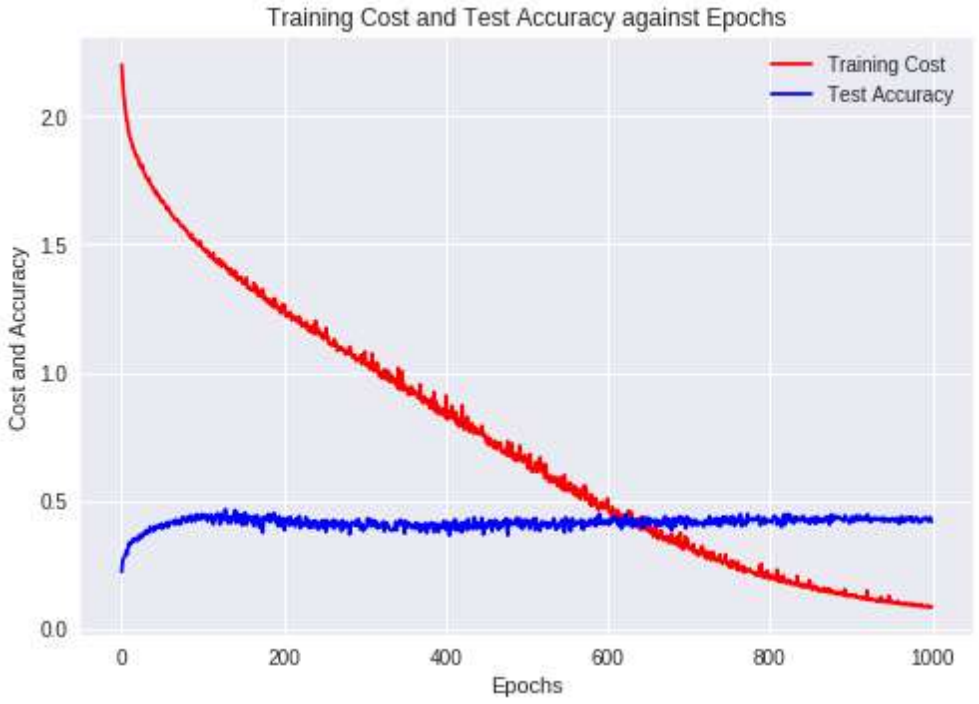
C2=200 filters

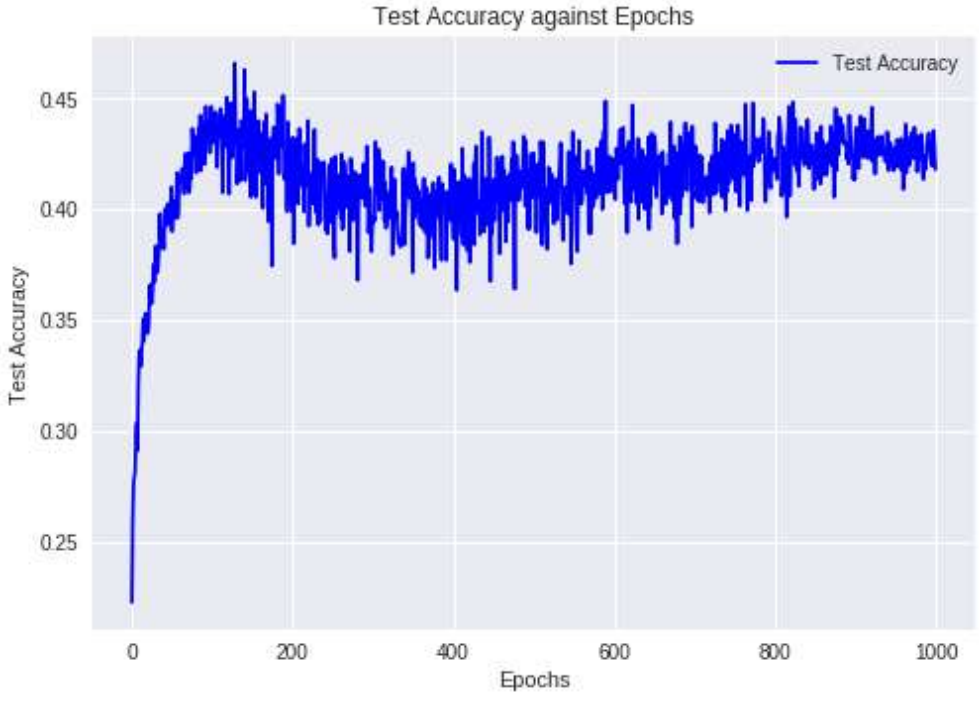
Epochs = 1000

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but

consistent gradients, or noisy gradients. When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture. Momentum is one method of speeding the convergence along a narrow ravine. The momentum parameter # ∈ 0,1 indicates how many iterations the previous gradients are incorporated into the current update. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
In tensorflow:

train_step = tf.train.MomentumOptimizer(learning_rate, 0.1).minimize(loss)





3b. Using RMSProp algorithm for learning.

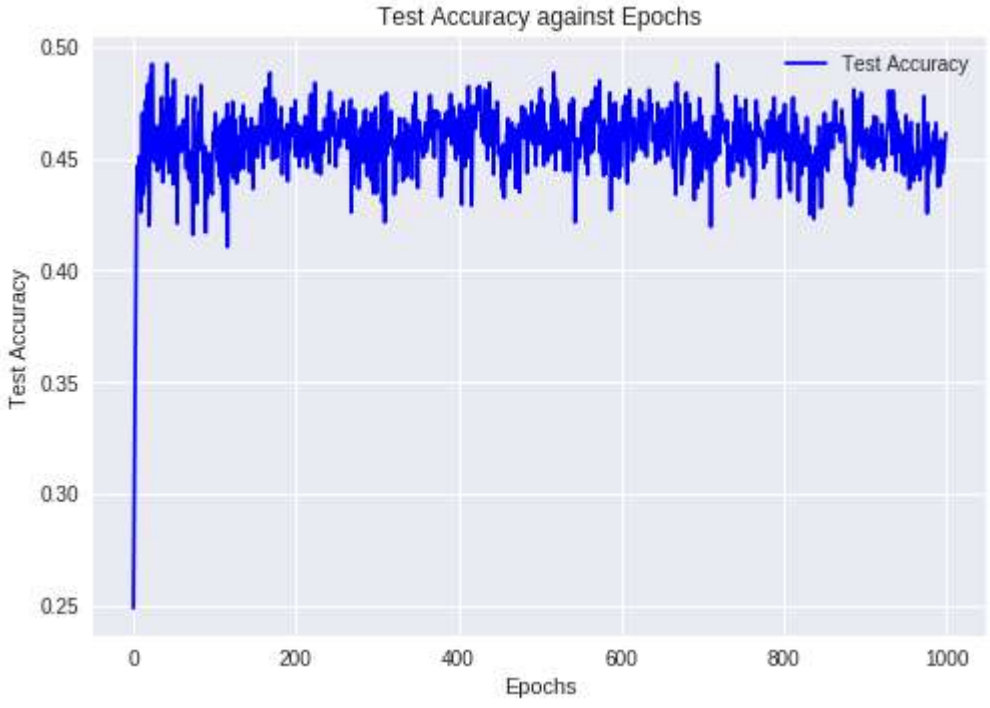
train_step = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

RMSProp improves upon AdaGrad algorithms uses and exponentially decaying average to decay from the extreme past gradient. RMSProp uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)(\nabla_{\boldsymbol{W}} J)^2$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\sqrt{\varepsilon + \boldsymbol{r}}} \cdot (\nabla_{\boldsymbol{W}} J)$$

The parameter ρ controls the length of the moving average of gradients

RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.



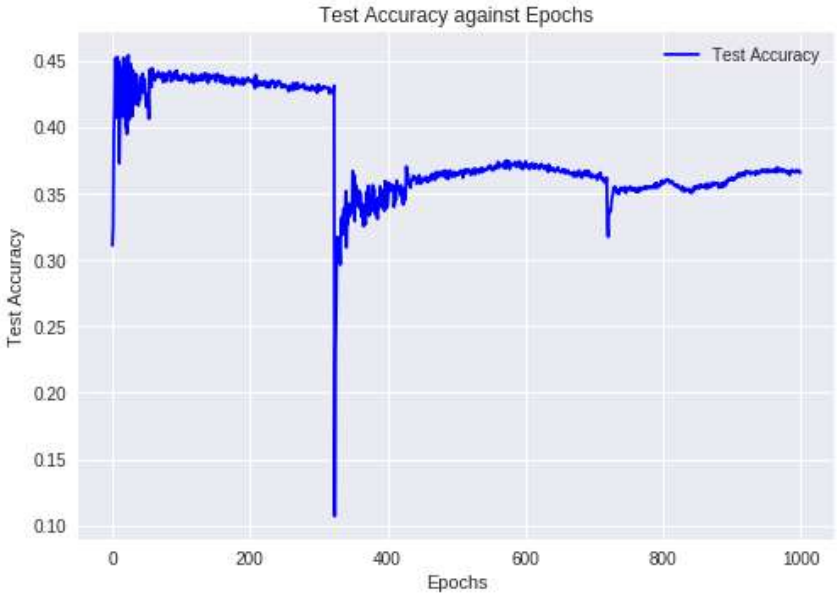
3c. Using Adam optimizer for learning

train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)

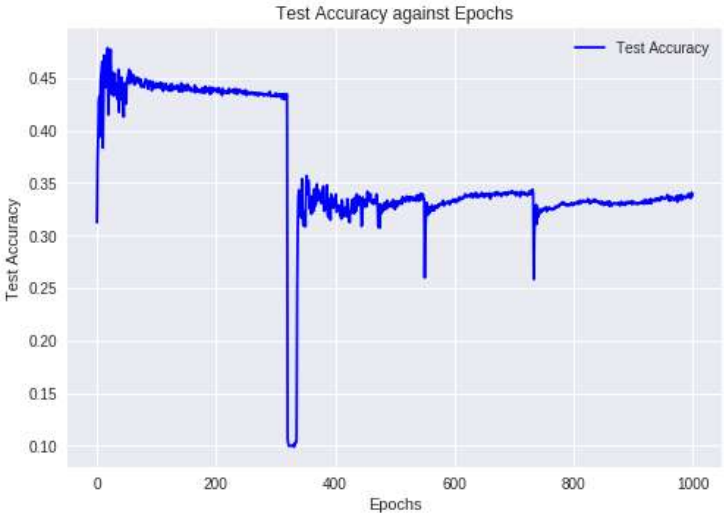
Adams optimizer combines RMSProp and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\nabla_W J$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)(\nabla_W J)^2$$
$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1}$$
$$\boldsymbol{r} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \frac{\alpha}{\varepsilon + \sqrt{\boldsymbol{r}}} \cdot \boldsymbol{s}$$

S adds the momentum and r contributes to the adaptive learning rate.

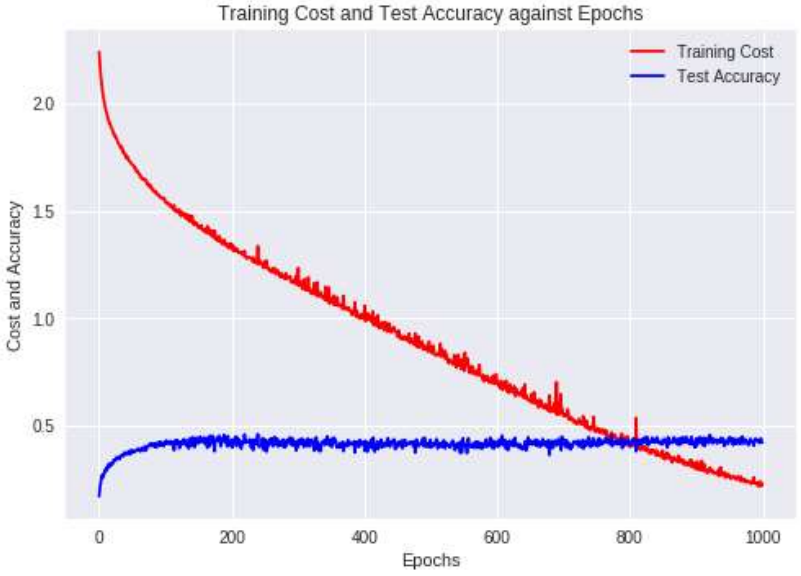


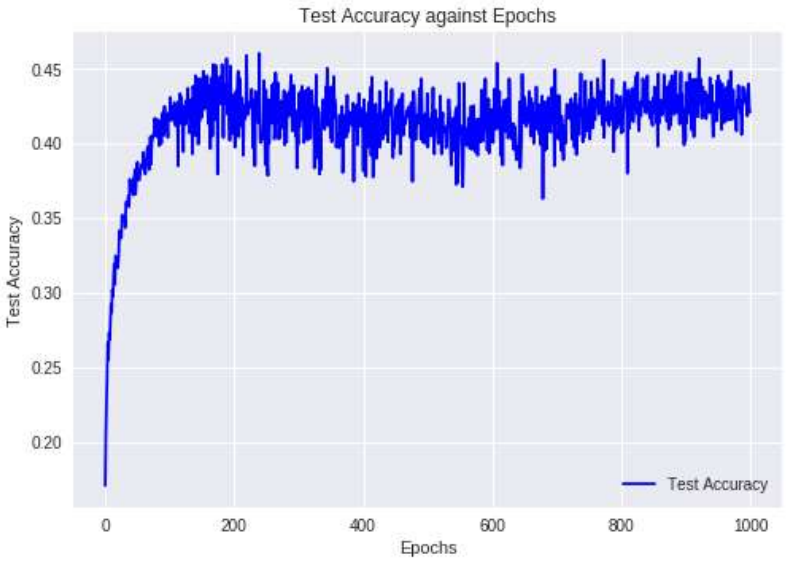
Second run:



d. Adding dropout to the layers

Keep prob of 0.9 with tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)



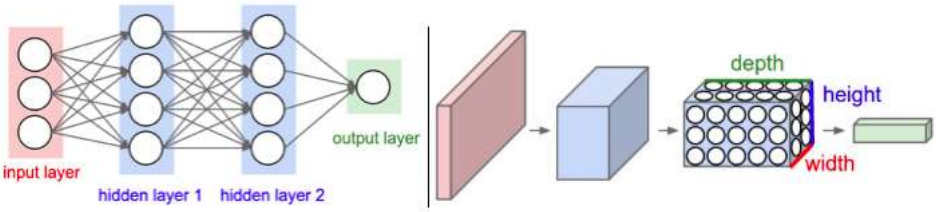


Keep prob of 0.9 with tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

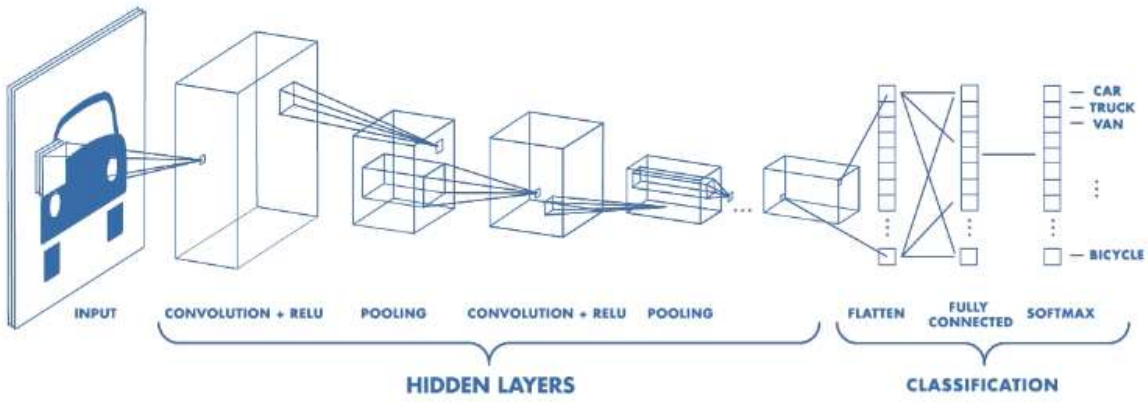
4. Compare the accuracies of all the models from parts (1) - (3) and discuss their performances.

The comparisons for parts (1) – (3) was explained in each section of (1) – (3) for better organization. Here are the more generalized conclusions:

Part (1): The test accuracy converges at around 0.4. Even with more epochs, the value did not increase. This is most likely because of the 32 by 32 low resolution of the individual images.



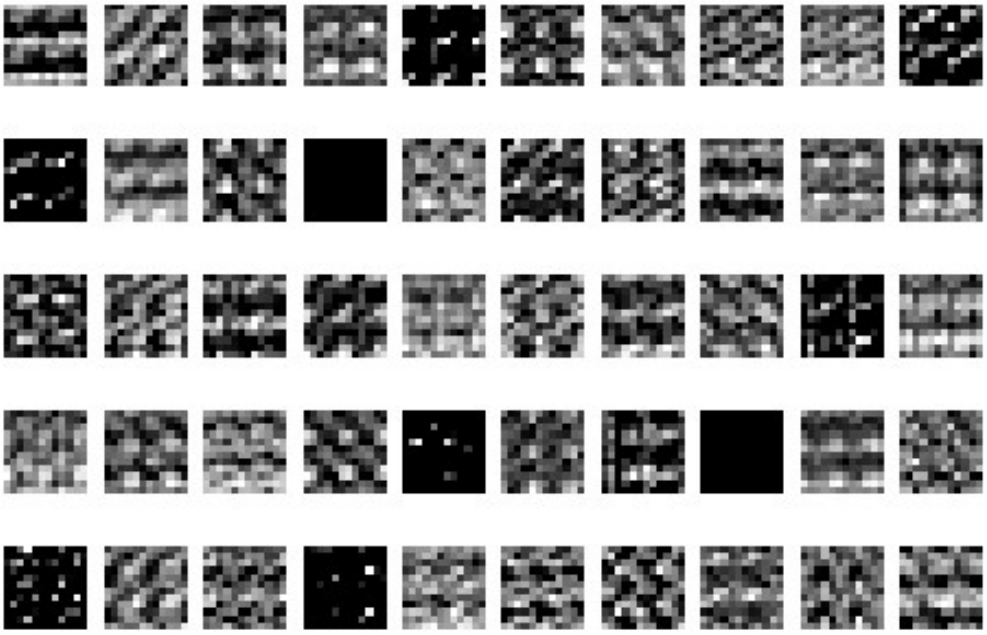
Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).



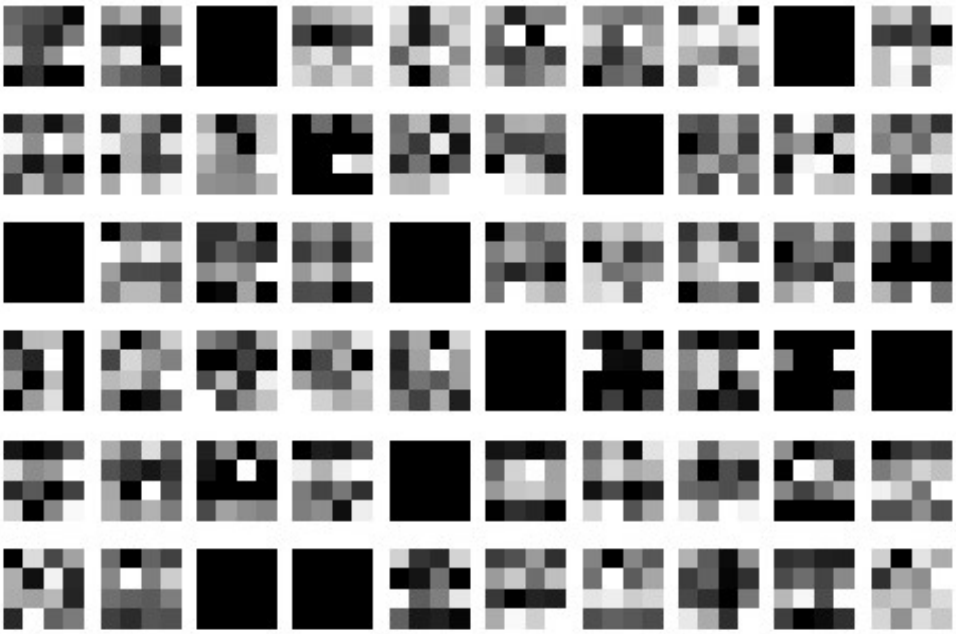
Pooling layer:

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

Max pooling layer S1:



Max pooling layer S2:



From these diagrams, it is observable that the spatial representation of the image is progressively reduced. This helps to reduce the amount of parameters and computation in the network, and hence to also control overfitting.

An alternative to pooling:

Many people dislike the pooling operation and think that we can get away without it. There has been proposals to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

Part (2): C1=100 filters, C2=200 filters is the most optimal. For this question, I first tried doing a grid search across a larger search space and then did another grid search with a smaller search space after observing the results for the first search. Usually the initial layers represent generic features, while the deeper layers represent more detailed features (of the specific dataset that is used for training the model). Hence, for this grid search, I will be using less filters for C1 and more layers for C2.

The richness of possible representations increases because a given layer feeds directly from a layer below to form new kernels by combinations of the features in the layer below. For example, the first hidden layer forms it's kernels by combinations of pixel values from the input, the combination of pixels is much richer than the pixels themselves, while the combination of pixel combinations (second hidden-layer) is even richer. And so on for the consecutive layers.

So the number of filters are incremented so as to be able to properly encode the increasingly richer and richer representations as the signal moves up the representational hierarchy in order to avoid the bottleneck effect.

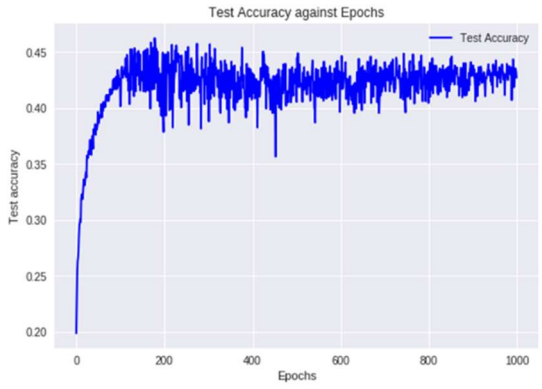
For the grid search, I ran:

- C1 with 40 filters and C2 with [60,80,100,120,140] filters
- C1 with 60 filters and C2 with [60,80,100,120,140] filters
- C1 with 80 filters and C2 with [80,100,120,140,160,180,200] filters
- C1 with 100 filters and C2 with [100,120,140,160,180,200] filters

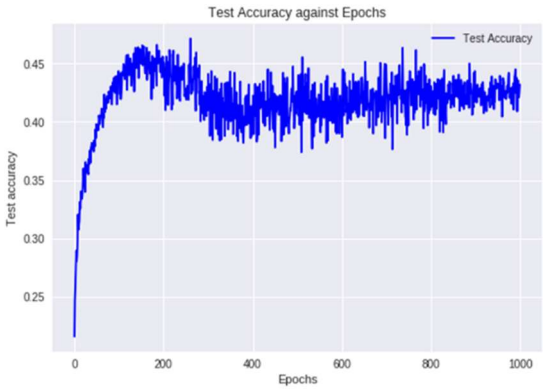
There are most tests for C1 with 80 filters and C2 with 100 filters because they seem to perform better.

Finally, the top few results are:

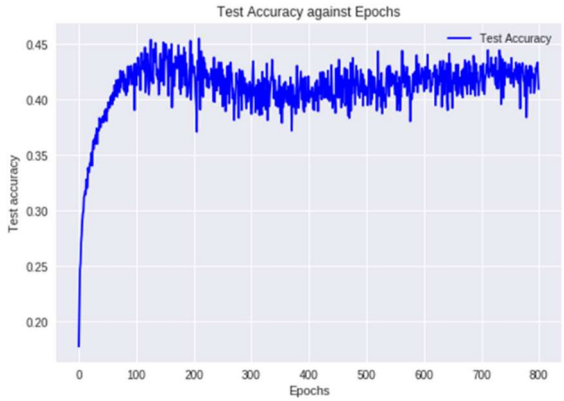
(C1=100, C2=200)



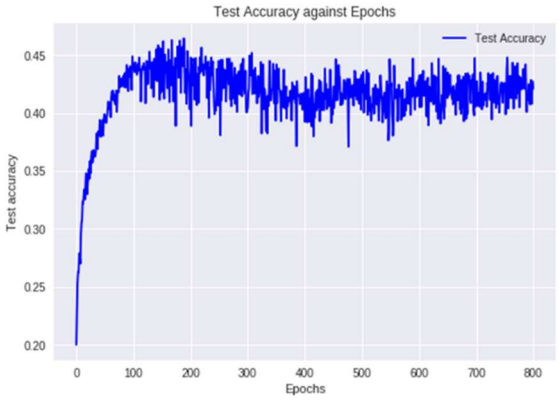
(C1=100, C2=180)



(C1=80, C2=200)



(C1=80, C2=180)

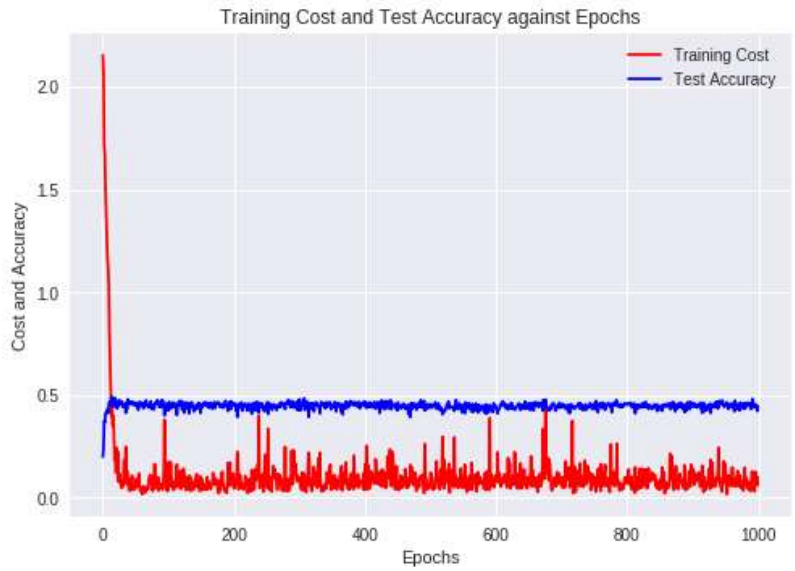


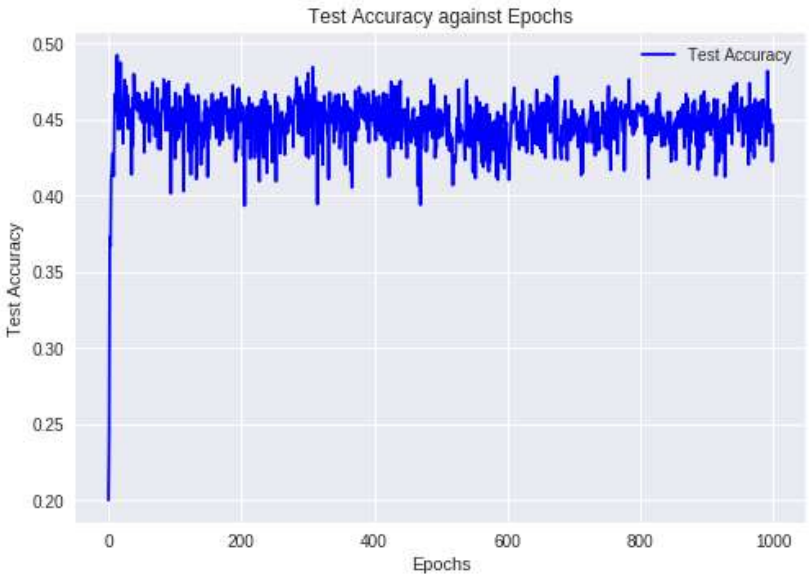
C1=100 filters, C2=200 filters is selected as the most optimal number of feature maps at the convolution layers.

(3a-c): RMSProp algorithm for learning seems to work best – it produces better test accuracies than the standard GradientDescentOptimizer. The test accuracy does not fluctuate too much across the epochs and the test accuracy is pretty consistent at the value of slightly higher than 0.45. Out of all the alternative training methods, the AdamOptimizer seems to produce worst results – it does not seem to be converging properly and fluctuates around too much. This is despite running it for a few times. Explanations of each algorithm and the diagrams generated are provided in Section 3 above for better organization.

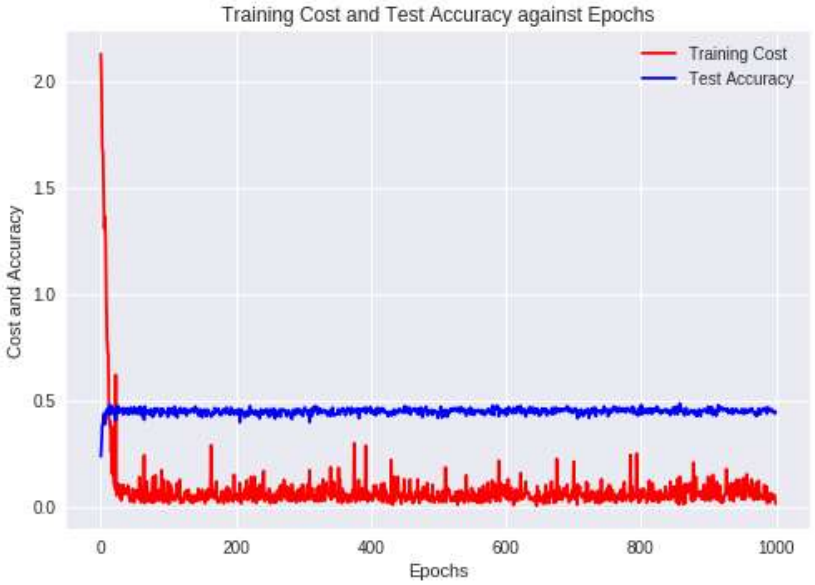
(3d): Keep prob of 0.9

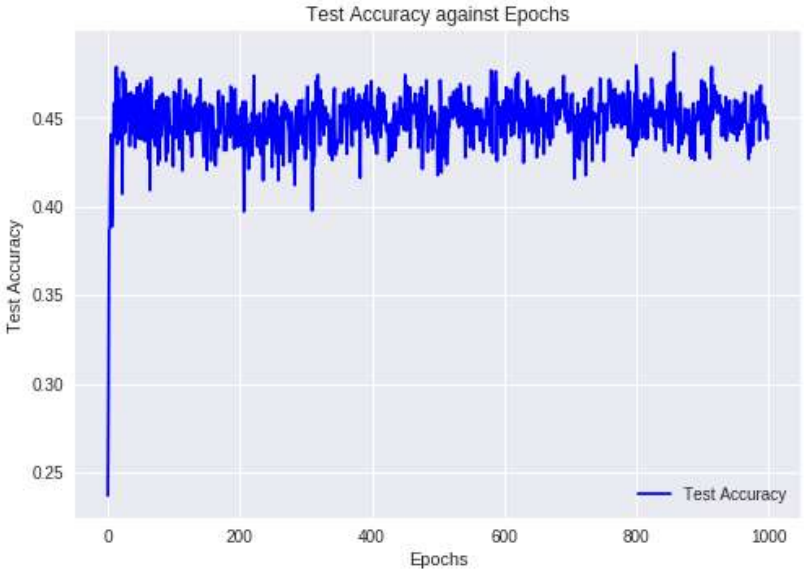
Testing with dropping out at different layers and not dropping out at different layers.
Using the RMSProp algorithm:
Dropping at C1, C2 and full layer. Drop out after each relu activation function:



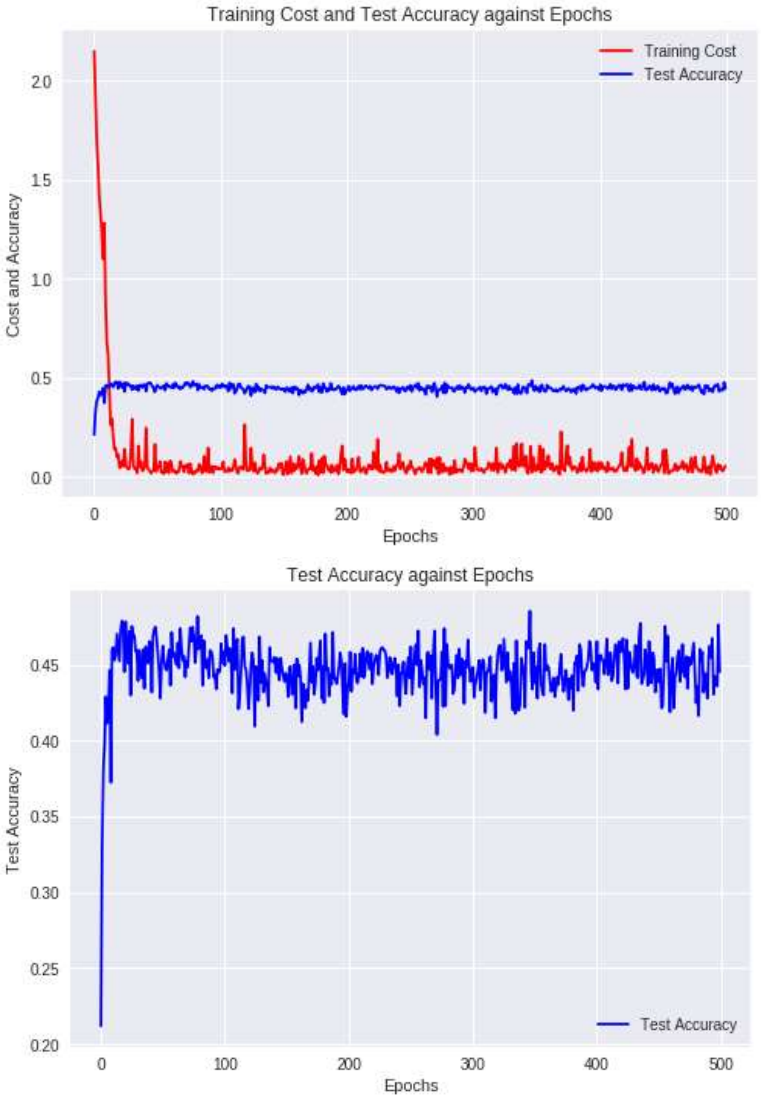


Dropping at C1 and C2 only.





Dropping at the fully connected layer before softmax layer.



From the results here, we can see that dropouts with keep_prob of 0.9 does not significantly improve results.

Dropout is a regularization technique, and is most effective at preventing overfitting. However, there are several places when dropout can hurt performance.

1. Right before the last layer. This is generally a bad place to apply dropout, because the network has no ability to "correct" errors induced by dropout before the classification happens.
2. When the network is small relative to the dataset, regularization is usually unnecessary. If the model capacity is already low, lowering it further by adding regularization may hurt performance.

Another technique that can replace drop outs is by using batch normalization. Batch normalisation is a technique for improving the performance and stability of neural networks, and also makes more sophisticated deep learning architectures work in practice (like DCGANs). The idea is to normalise the inputs of each layer in such a way that they have a mean output activation of zero and standard deviation of one. This is analogous to how the inputs to networks are standardised.