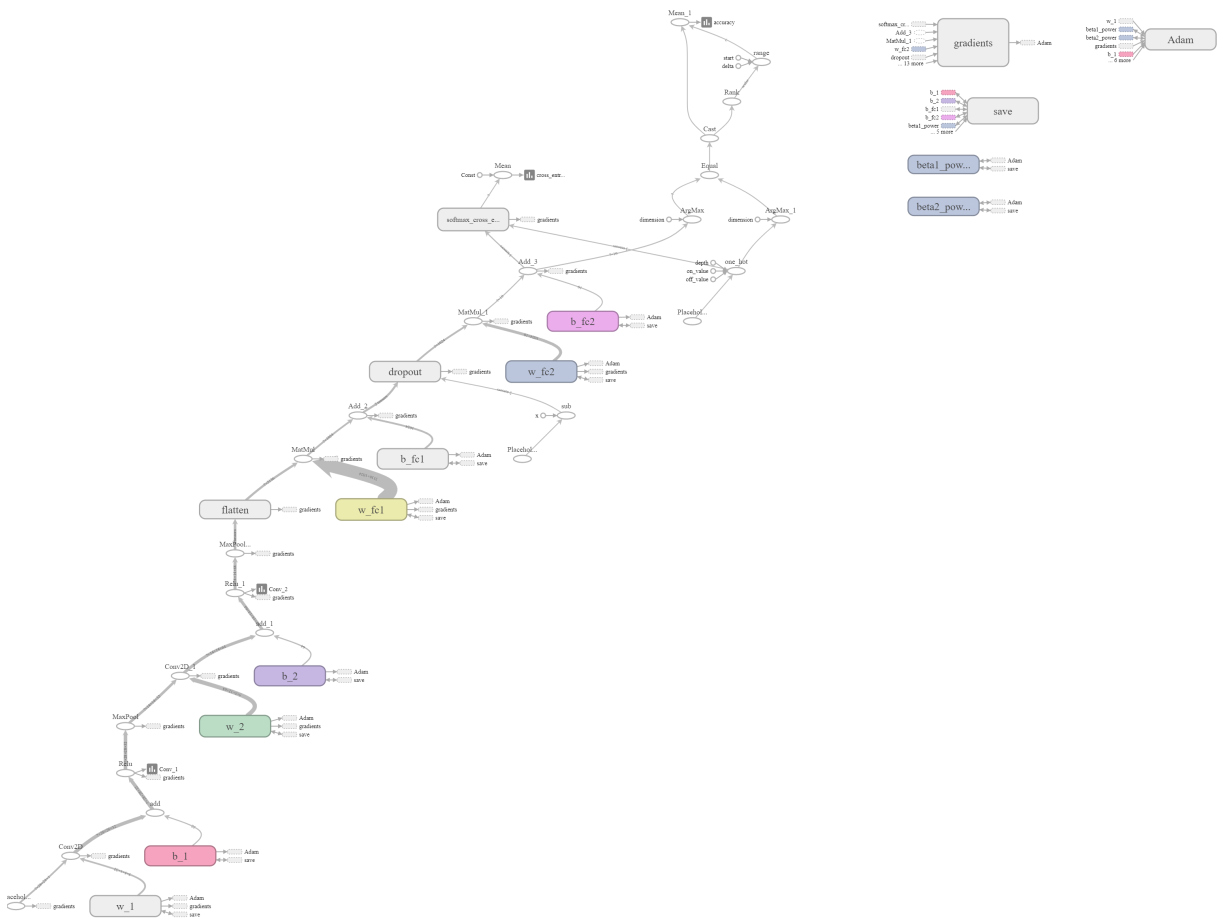
ASSIGNMENT 2

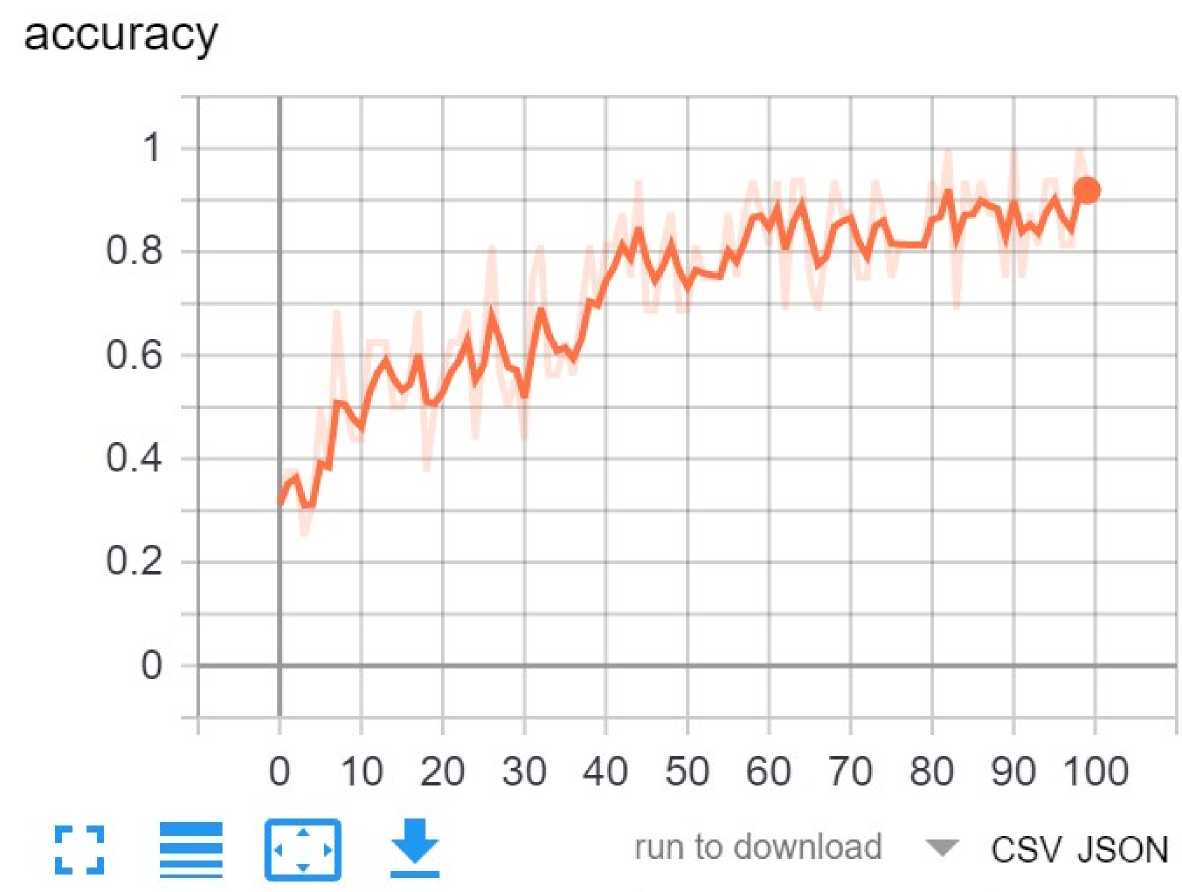
Part 1 (Visualizing a CNN with CIFAR10)

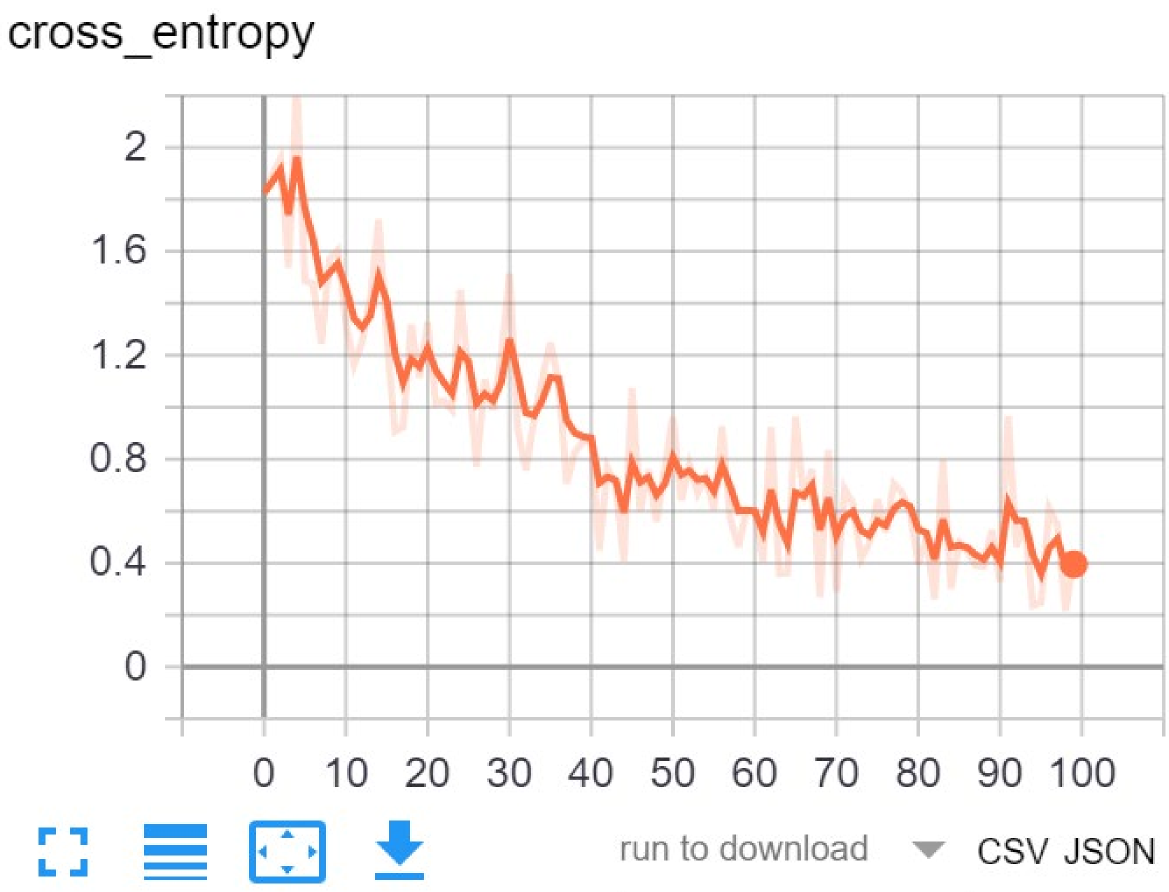
Train LeNet5 on CIFAR10

After training the CIFAR10 Dataset, below is the architecture framework of the network using tensorboard.



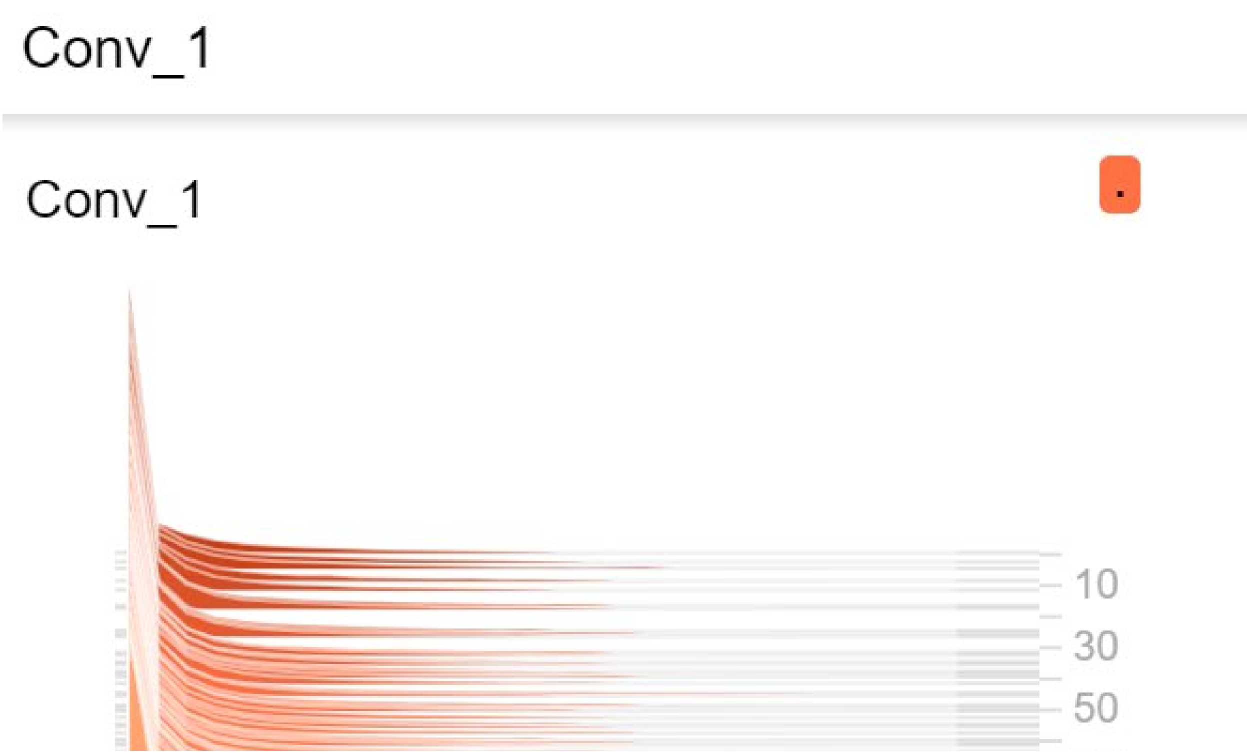
Also, using AdamOptimizer with learning_rate = 0.0001, the final training accuracy was 93.75 percent (= 0.9375) and training loss was 41.46 percent (= 0.4145758) at 100 epochs. Also the test accuracy was 56.9 percent which is not very high.

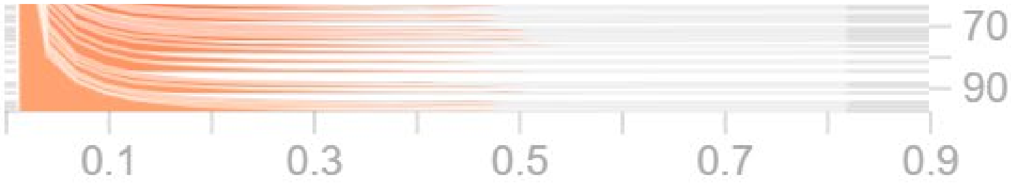

accuracy
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
50
60
70
80
90
100
run to download
CSV JSON



However, the highest training accuracy and training loss for all iteration was found to be epoch = 98 iteration = 78 batch loss = 0.21675661 epoch = 98 iteration = 78 accuracy = 1.0

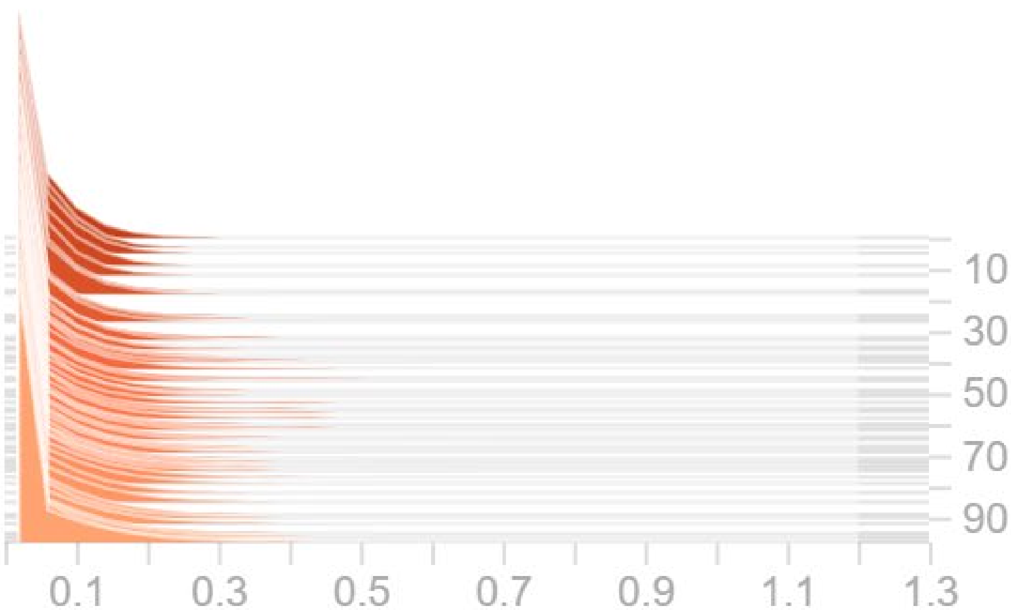
Using tensorboard like the previous homework, the plot of histogram distribution of the weight for the first fully connected layer every iterations can be seen below:



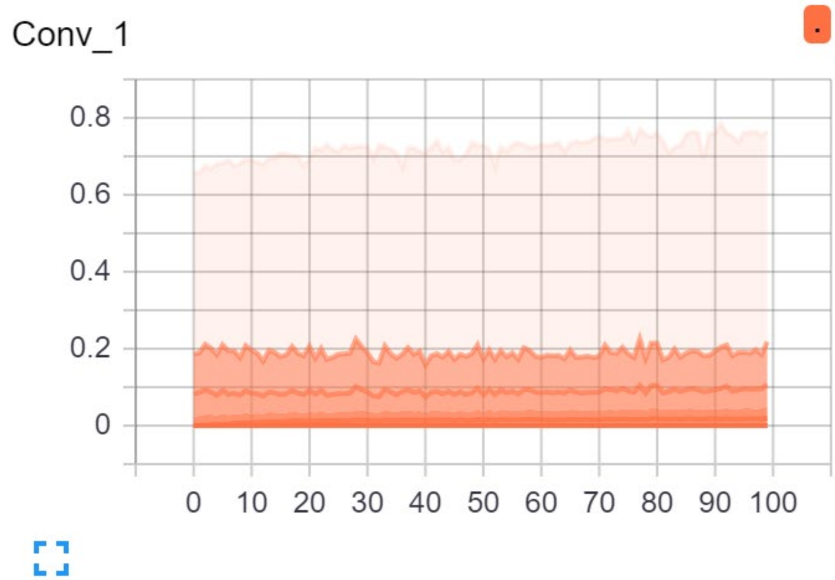


Conv_2

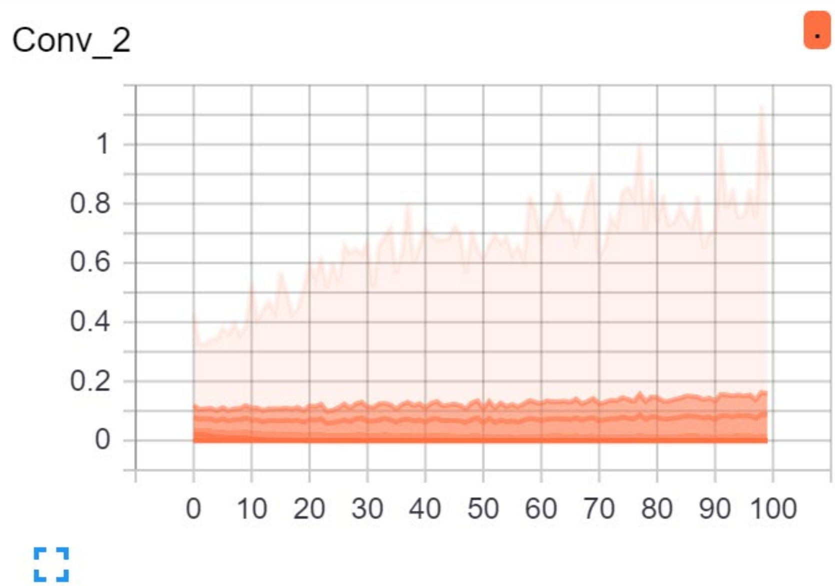
Conv_2



Conv_1 1



Conv_2 1



Also below are the test means and test standard deviations of the weighs in each layers:

Means for first layer: [0.016132174, 0.029338859, 0.01393891, 0.08973887, 0.031540327, 0.01676252, 0.22206718, 0.019344307, 0.013705693, 0.012988208, 0.037079323, 0.023153018, 0.01855602, 0.13712077, 0.041094296, 0.03676815, 0.098006554, 0.016511103, 0.025684064, 0.027391933, 0.05253628, 0.034088288, 0.03209313, 0.040605433, 0.029922921, 0.035356913, 0.3311147, 0.022650024, 0.02995678, 0.034102045, 0.020353775, 0.029478675]

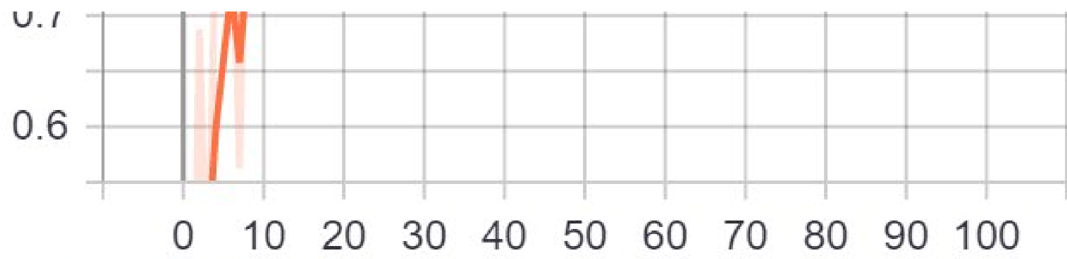
Standard Deviation for first layer: [0.02915756, 0.038086936, 0.019988138, 0.082939364, 0.037936207, 0.027575852, 0.10899919, 0.024041682, 0.023784598, 0.019124055, 0.055252455, 0.030977795, 0.023042217, 0.086668625, 0.035565533, 0.057067554, 0.09598874, 0.030323485, 0.051125195, 0.031572353, 0.06088244, 0.04654906, 0.043900803, 0.063050024, 0.042631447, 0.046884242, 0.16524898, 0.028067151, 0.05684843, 0.04443655, 0.027996512, 0.043884825]

Means for second layer: [0.027298879, 0.035685092, 0.039673265, 0.033722837, 0.023302646, 0.036286287, 0.024474269, 0.0217095, 0.03454492, 0.025159061, 0.039617248, 0.031754762, 0.045082152, 0.027849874, 0.03100977, 0.025041549, 0.019566825, 0.023994189, 0.043361876, 0.04382928, 0.019488882, 0.032915432, 0.022974387, 0.040203024, 0.031487584, 0.03400421, 0.036882248, 0.027963633, 0.041905727, 0.039056685, 0.04042995, 0.030967383, 0.021435905, 0.045604263, 0.03226313, 0.032411434, 0.035416115, 0.03084651, 0.040136755, 0.048507094, 0.040806767, 0.032124583, 0.06491389, 0.044385165, 0.035522502, 0.045388885, 0.032178864, 0.027475057, 0.031259637, 0.040684305, 0.026635565, 0.018128056, 0.04145645, 0.03401644, 0.035258997, 0.037756484, 0.023261363, 0.039127298, 0.044950772, 0.024582645, 0.025254158, 0.016750393, 0.03599538, 0.04138647]

Standard Deviations for second layer: [0.061776217, 0.061587073, 0.06404622, 0.060734514, 0.051323842, 0.054099385, 0.06820811, 0.045092247, 0.059013527, 0.044423725, 0.07956225, 0.061532382, 0.06505881, 0.05579189, 0.05334235, 0.04876248, 0.045079656, 0.05396011, 0.07569376, 0.078639425, 0.035681777, 0.0620447, 0.04612172, 0.056052435, 0.062345088, 0.056784045, 0.07562206, 0.060737666, 0.073725335, 0.073596686, 0.07487188, 0.055628233, 0.053808376, 0.07811466, 0.06707132, 0.06298042, 0.054162603, 0.066521585, 0.063783616, 0.082595944, 0.081171766, 0.06784395, 0.12339037, 0.0820339, 0.058915436, 0.07345317, 0.066693954, 0.051331073, 0.042380318, 0.065567166, 0.061258085, 0.03704379, 0.08340636, 0.059124853, 0.074299484, 0.07128134, 0.044010922, 0.073321484, 0.09325731, 0.047742385, 0.056028005, 0.036077257, 0.066954054, 0.07184373]

We noticed by trying different learning rate we discover several thigs and below are some of these discoveries:

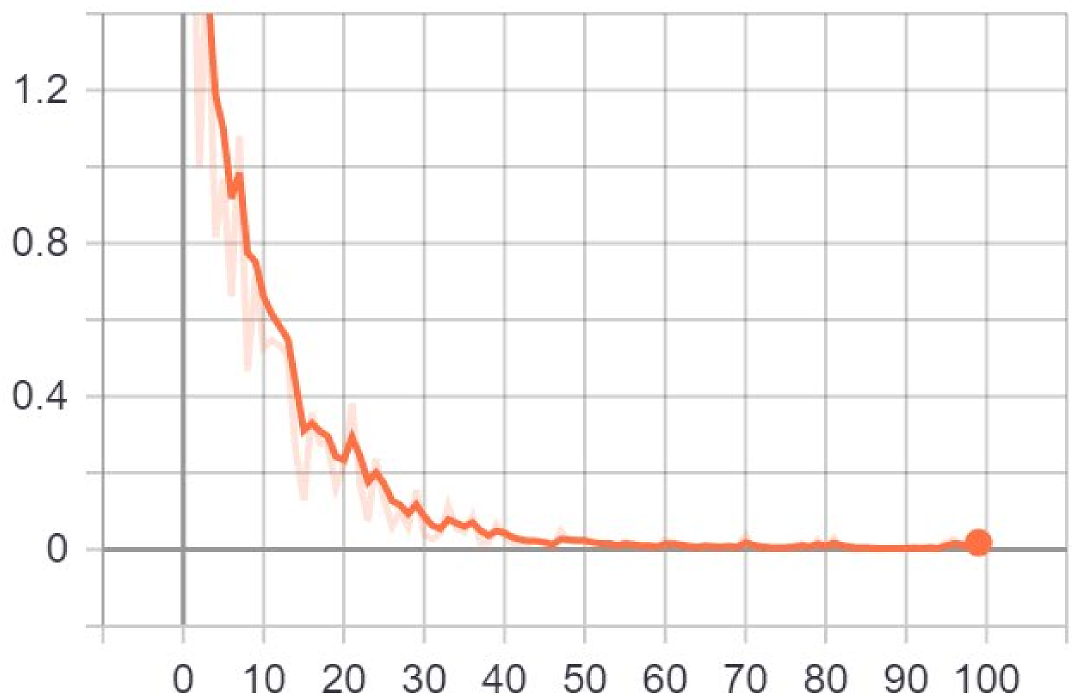
Using learning rate = 0.001 an increase from the 0.0001 used in this report. We have acheived epoch = 22 iteration = 78 batch loss = 0.17137559 accuracy = 1.0 and the test accuracy = 0.501 The training accuracy = 1 did not changed however the loss continues to reduce unitl 0.030311339 at the end of al epochs.



run to download CSV JSON

cross_entropy

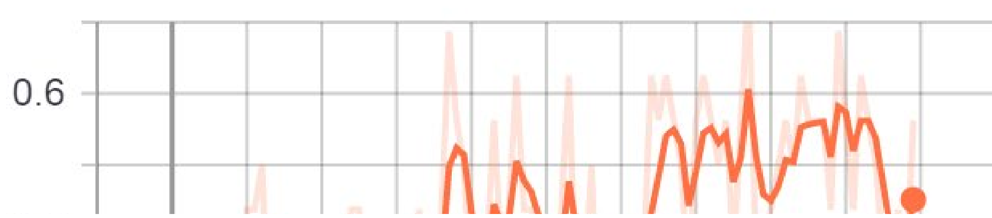
cross_entropy

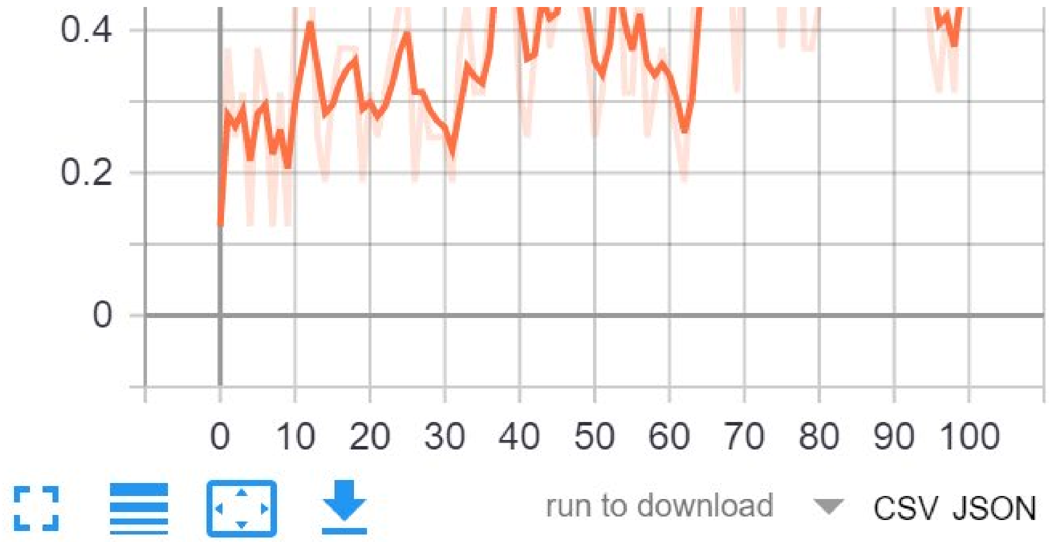


run to download CSV JSON

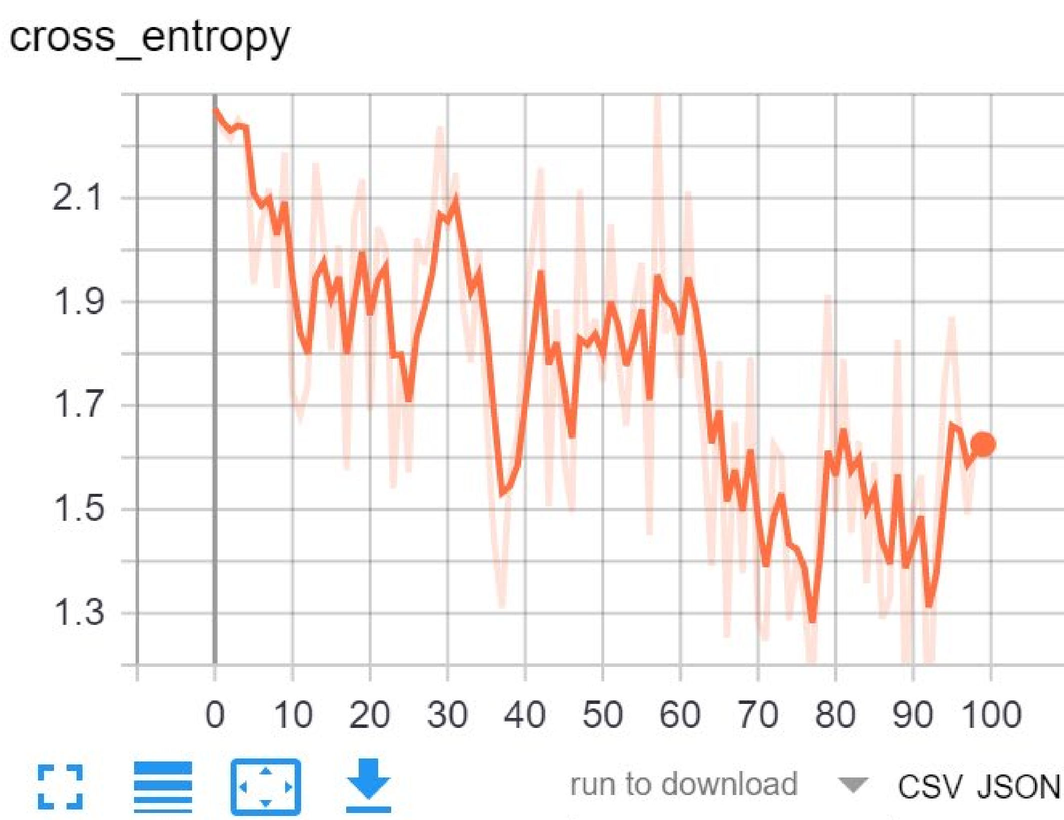
However, using learning rate = 0.00001 a decrease from the 0.0001 used in this report. We have loss = 1.6505405 and accuracy = 0.5625 at the end of training and the test accuracy = 0.451. This is not a good choice.

accuracy





cross_entropy



In [ ]:

In [1]:
```
from scipy import misc
import numpy as np
import tensorflow as tf
import random
```

```
import pylab as pl
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import matplotlib as mp
from visualize_weights import nice_imshow, make_mosaic

# --------------------------------------------------
# setup
result_dir = 'result_summary/'

def weight_variable(shape):
    '''
    Initialize weights
    :param shape: shape of weights, e.g. [w, h ,Cin, Cout] where
    w: width of the filters
    h: height of the filters
    Cin: the number of the channels of the filters
    Cout: the number of filters
    :return: a tensor variable for weights with initial values
    '''

    # IMPLEMENT YOUR WEIGHT_VARIABLE HERE
    initial = tf.truncated_normal(shape)
    W = tf.Variable(initial)

    return W

def weight_(shape, name, init):
    w = tf.get_variable(name=name, shape=shape, initializer=init, dtype=tf
.float32)

    return w

def bias_variable(shape):
    '''
    Initialize biases
    :param shape: shape of biases, e.g. [Cout] where
    Cout: the number of filters    :return: a tensor variable for biases w
ith initial values
    '''

    # IMPLEMENT YOUR BIAS_VARIABLE HERE
    initial = tf.constant(0.1, shape=shape)
    b = tf.Variable(initial)

    return b

def conv2d(x, W):
    '''
    Perform 2-D convolution
    :param x: input tensor of size [N, W, H, Cin] where
    N: the number of images
    W: width of images
    H: height of images
    Cin: the number of channels of images
    :param W: weight tensor [w, h, Cin, Cout]
    w: width of the filters
```

```
    h: height of the filters
    Cin: the number of the channels of the filters = the number of channel
s of images
    Cout: the number of filters
    :return: a tensor of features extracted by the filters, a.k.a. the res
ults after convolution
    '''

    # IMPLEMENT YOUR CONV2D HERE
    h_conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

    return h_conv

def max_pool_2x2(x):
    '''
    Perform non-overlapping 2-D maxpooling on 2x2 regions in the input dat
a
    :param x: input data
    :return: the results of maxpooling (max-marginalized + downsampling)
    '''

    # IMPLEMENT YOUR MAX_POOL_2X2 HERE
    h_max = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], pa
dding='SAME')

    return h_max


ntrain = 1000 # per class
ntest = 100  # per class
nclass = 10  # number of classes
imgwidth = 28
imgheight = 28
imsize = imgwidth * imgheight
nchannels = 1
batchsize = 128
max_step = 100

Train = np.zeros((ntrain*nclass,imgwidth,imgheight,nchannels))
Test = np.zeros((ntest*nclass,imgwidth,imgheight,nchannels))
LTrain = np.zeros((ntrain*nclass,nclass))
LTest = np.zeros((ntest*nclass,nclass))

itrain = -1
itest = -1
for iclass in range(0, nclass):
    for isample in range(0, ntrain):
        path = 'CIFAR10/Train/%d/Image%05d.png' % (iclass,isample)
        im = misc.imread(path);  # 28 by 28
        im = im.astype(float) / 255
        itrain += 1
        Train[itrain,:,:,0] = im
        LTrain[itrain,iclass] = 1 # 1-hot lable
    for isample in range(0, ntest):
        path = 'CIFAR10/Test/%d/Image%05d.png' % (iclass,isample)
        im = misc.imread(path);  # 28 by 28
        im = im.astype(float) / 255
```

```
        itest += 1
        Test[itest,:,:,0] = im
        LTest[itest,iclass] = 1 # 1-hot lable

sess = tf.InteractiveSession()

tf_data = tf.placeholder(tf.float32, [None, 28, 28, 1]) #tf variable for t
he data, remember shape is [None, width, height, numberOfChannels]
tf_labels = tf.one_hot(tf.placeholder(tf.uint8), 10) #tf variable for labe
ls
keep_prob = tf.placeholder(tf.float32)

# --------------------------------------------------
# model
#create your model
W_conv1 = weight_(shape = [5, 5, 1, 32],name='w_1', init= tf.contrib.layer
s.xavier_initializer())
b_conv1 = weight_(shape = [32], name='b_1', init= tf.constant_initializer(
value=0))
h_conv1 = tf.nn.relu(conv2d(tf_data, W_conv1) + b_conv1)
tf.summary.histogram(('Conv_1'),h_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# second convolutional layer
W_conv2 = weight_(name='w_2', shape=[5, 5, 32, 64], init=tf.contrib.layers
.xavier_initializer())
b_conv2 = weight_(name='b_2', shape =[64], init= tf.constant_initializer(v
alue=0))
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
tf.summary.histogram('Conv_2', h_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_(shape=[7 * 7 * 64, 1024], name='w_fc1', init=tf.contrib.la
yers.xavier_initializer())
b_fc1 = weight_(shape=[1024], name='b_fc1', init=tf.constant_initializer(v
alue=0))
h_pool2_flat = tf.layers.flatten(h_pool2)
h_fc1 = tf.add(tf.matmul(h_pool2_flat, W_fc1), b_fc1)

# dropout
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# softmax
W_fc2 = weight_(name='w_fc2', shape=[1024, 10], init=tf.contrib.layers.xav
ier_initializer())
b_fc2 = weight_(name='b_fc2', shape=[10], init=tf.constant_initializer(val
ue=0))
y_conv = tf.add(tf.matmul(h_fc1_drop, W_fc2), b_fc2)


def data_iterator(train_images, train_labels, batch_size):
    """ A simple data iterator """
    n = train_images.shape[0]
    # batch_idx = 0
    while True:

        shuf_idxs = np.random.permutation(n).reshape((1, n))[0]
```

```
        shuf_images = train_images[shuf_idxs]
        shuf_labels = train_labels[shuf_idxs]

        for batch_idx in range(0, n, batch_size):
            # print(shuf_idxs[batch_idx: batch_idx + batch_size])
            batch_images = shuf_images[batch_idx: batch_idx + batch_size]
            batch_labels = shuf_labels[batch_idx: batch_idx + batch_size]
            yield batch_images, batch_labels


# --------------------------------------------------
# loss
#set up the loss, optimization, evaluation, and accuracy
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
logits=y_conv, labels=tf_labels))
optimizer = tf.train.AdamOptimizer(learning_rate=0.0001).minimize(cross_en
tropy)
pred_idx = tf.argmax(y_conv, 1)
y_idx = tf.argmax(tf_labels, 1)
correct_prediction = tf.equal(pred_idx, y_idx)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float32'))

tf.summary.scalar("cross_entropy", cross_entropy)
tf.summary.scalar("accuracy", accuracy)

summary_op = tf.summary.merge_all()

saver = tf.train.Saver()
summary_writer = tf.summary.FileWriter(result_dir, sess.graph)


# --------------------------------------------------
# optimization

sess.run(tf.global_variables_initializer())
#batch_xs = np.zeros([batchsize, imwidth, imheight, nchannels]) #setup as
[batchsize, width, height, numberOfChannels] and use np.zeros()
#batch_ys = np.zeros([batchsize, nclass]) #setup as [batchsize, the how ma
ny classes]

iter_ = data_iterator(Train, LTrain, batchsize)
n_batch = len(LTrain) // batchsize + 1

for epoch in range(max_step):
    for i in range(n_batch): # try a small iteration size once it works th
en continue
        batch_x, batch_y = next(iter_)
        sess.run(optimizer, feed_dict={tf_data: batch_x, tf_labels: batch_
y, keep_prob: 0.5})


         # perm = np.arange(nsamples)
         # np.random.shuffle(perm)
        # for j in range(batchsize):
            # batch_xs[j,:,:,:] = Train[perm[j],:,:,:]
            # batch_ys[j,:] = LTrain[perm[j],:]
        # if i%10 == 0:
```

```
        if i == n_batch-1:
            batch_loss, summ, accu = sess.run([cross_entropy, summary_op,
accuracy],
                                    feed_dict={tf_data: batch_x, tf_labels: ba
tch_y, keep_prob: 1.0})
            summary_writer.add_summary(summ, epoch)
            print("epoch = ", epoch, "iteration = ", i, "batch loss = ", b
atch_loss)
            print("epoch = ", epoch, "iteration = ", i, "accuracy = ", acc
u)
            #calculate train accuracy and print it
     #   optimizer.run(feed_dict={}) # dropout only during training


# visualize weights
W_c = sess.run(W_conv1)
W_c2 = W_c[:, :, 0, :]
print(W_c.shape)
print('W_c[0]:', W_c[0])
W = np.transpose(W_c2, (2, 0, 1))
plt.figure(figsize=(15, 15))
plt.title('conv1 weights')
nice_imshow(plt.gca(), make_mosaic(W, 6, 6), cmap=cm.binary)
plt.close()

# --------------------------------------------------
# test
test_stat = sess.run(h_conv1, feed_dict={tf_data: Test, tf_labels: LTest,
keep_prob: 1.0})
test_stat2 = sess.run(h_conv2, feed_dict={tf_data: Test, tf_labels: LTest,
 keep_prob: 1.0})


test_mean = []
test_sd = []
for i in range(32):
    ax = plt.subplot(8, 4, i+1)

    data_stat = test_stat[:, :, :, i].flatten()
    plt.hist(data_stat)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    test_mean.append(np.mean(data_stat))
    test_sd.append(np.std(data_stat))


print("Means for first layer:", test_mean)
print("Standard Deviation for first layer:", test_sd)

plt.close()

test_mean = []
test_sd = []
for i in range(32):
    ax = plt.subplot(8, 4, i + 1)

    data_stat = test_stat2[:, :, :, i].flatten()
```

```
    plt.hist(data_stat)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    test_mean.append(np.mean(data_stat))
    test_sd.append(np.std(data_stat))


#plt.savefig('Images/hist_conv21_all.png')
#plt.close()

for i in range(32,64):
    ax = plt.subplot(8, 4, i - 31)

    data_stat = test_stat2[:, :, :, i].flatten()
    plt.hist(data_stat)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    test_mean.append(np.mean(data_stat))
    test_sd.append(np.std(data_stat))


#plt.savefig('Images/hist_conv22_all.png')
#plt.close()
print("Means for second layer:", test_mean)
print("Standard Deviations for second layer:", test_sd)

print("test accuracy %g"%accuracy.eval(feed_dict={tf_data: Test, tf_labels
: LTest, keep_prob: 1.0}))


sess.close()
```

```
C:\Users\oyeoy\Anaconda3\lib\site-packages\ipykernel_launcher.py:104: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
C:\Users\oyeoy\Anaconda3\lib\site-packages\ipykernel_launcher.py:111: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
WARNING: Logging before flag parsing goes to stderr.
W1105 03:53:39.478218  3556 lazy_loader.py:50]
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-cont
rib-sunset.md
  * https://github.com/tensorflow/addons
  * https://github.com/tensorflow/io (for I/O related ops)
If you depend on functionality not listed there, please file an issue.

W1105 03:53:39.538094  3556 deprecation.py:323] From <ipython-input-1-852a
5042b36a>:141: flatten (from tensorflow.python.layers.core) is deprecated
and will be removed in a future version.
Instructions for updating:
Use keras.layers.flatten instead.
W1105 03:53:39.711630  3556 deprecation.py:506] From <ipython-input-1-852a
5042b36a>:145: calling dropout (from tensorflow.python.ops.nn_ops) with ke
```

```
ep_prob is deprecated and will be removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1
- keep_prob`.
```

```
epoch =  0 iteration =  78 batch loss =  1.8244528
epoch =  0 iteration =  78 accuracy =  0.3125
epoch =  1 iteration =  78 batch loss =  1.8932481
epoch =  1 iteration =  78 accuracy =  0.375
epoch =  2 iteration =  78 batch loss =  1.9573536
epoch =  2 iteration =  78 accuracy =  0.375
epoch =  3 iteration =  78 batch loss =  1.5367157
epoch =  3 iteration =  78 accuracy =  0.25
epoch =  4 iteration =  78 batch loss =  2.2576995
epoch =  4 iteration =  78 accuracy =  0.3125
epoch =  5 iteration =  78 batch loss =  1.4862906
epoch =  5 iteration =  78 accuracy =  0.5
epoch =  6 iteration =  78 batch loss =  1.4745551
epoch =  6 iteration =  78 accuracy =  0.375
epoch =  7 iteration =  78 batch loss =  1.2446125
epoch =  7 iteration =  78 accuracy =  0.6875
epoch =  8 iteration =  78 batch loss =  1.5704011
epoch =  8 iteration =  78 accuracy =  0.5
epoch =  9 iteration =  78 batch loss =  1.6041176
epoch =  9 iteration =  78 accuracy =  0.4375
epoch =  10 iteration =  78 batch loss =  1.3221052
epoch =  10 iteration =  78 accuracy =  0.4375
epoch =  11 iteration =  78 batch loss =  1.1640774
epoch =  11 iteration =  78 accuracy =  0.625
epoch =  12 iteration =  78 batch loss =  1.2542493
epoch =  12 iteration =  78 accuracy =  0.625
epoch =  13 iteration =  78 batch loss =  1.4203117
epoch =  13 iteration =  78 accuracy =  0.625
epoch =  14 iteration =  78 batch loss =  1.7228533
epoch =  14 iteration =  78 accuracy =  0.5
epoch =  15 iteration =  78 batch loss =  1.2735868
epoch =  15 iteration =  78 accuracy =  0.5
epoch =  16 iteration =  78 batch loss =  0.90640306
epoch =  16 iteration =  78 accuracy =  0.5625
epoch =  17 iteration =  78 batch loss =  0.9233516
epoch =  17 iteration =  78 accuracy =  0.6875
epoch =  18 iteration =  78 batch loss =  1.3187904
epoch =  18 iteration =  78 accuracy =  0.375
epoch =  19 iteration =  78 batch loss =  1.1125401
epoch =  19 iteration =  78 accuracy =  0.5
epoch =  20 iteration =  78 batch loss =  1.3293891
epoch =  20 iteration =  78 accuracy =  0.5625
epoch =  21 iteration =  78 batch loss =  1.0163113
epoch =  21 iteration =  78 accuracy =  0.625
epoch =  22 iteration =  78 batch loss =  1.0243292
epoch =  22 iteration =  78 accuracy =  0.625
epoch =  23 iteration =  78 batch loss =  0.9885441
epoch =  23 iteration =  78 accuracy =  0.6875
epoch =  24 iteration =  78 batch loss =  1.4522048
epoch =  24 iteration =  78 accuracy =  0.4375
epoch =  25 iteration =  78 batch loss =  1.1201768
epoch =  25 iteration =  78 accuracy =  0.625
```

```
epoch =  26 iteration =  78 batch loss =  0.7719278
epoch =  26 iteration =  78 accuracy =  0.8125
epoch =  27 iteration =  78 batch loss =  1.1057065
epoch =  27 iteration =  78 accuracy =  0.5625
epoch =  28 iteration =  78 batch loss =  0.9888904
epoch =  28 iteration =  78 accuracy =  0.5
epoch =  29 iteration =  78 batch loss =  1.2022023
epoch =  29 iteration =  78 accuracy =  0.5625
epoch =  30 iteration =  78 batch loss =  1.5153153
epoch =  30 iteration =  78 accuracy =  0.4375
epoch =  31 iteration =  78 batch loss =  0.9170447
epoch =  31 iteration =  78 accuracy =  0.75
epoch =  32 iteration =  78 batch loss =  0.7581726
epoch =  32 iteration =  78 accuracy =  0.8125
epoch =  33 iteration =  78 batch loss =  0.9522314
epoch =  33 iteration =  78 accuracy =  0.5625
epoch =  34 iteration =  78 batch loss =  1.1102428
epoch =  34 iteration =  78 accuracy =  0.5625
epoch =  35 iteration =  78 batch loss =  1.2476431
epoch =  35 iteration =  78 accuracy =  0.625
epoch =  36 iteration =  78 batch loss =  1.1037424
epoch =  36 iteration =  78 accuracy =  0.5625
epoch =  37 iteration =  78 batch loss =  0.7070851
epoch =  37 iteration =  78 accuracy =  0.6875
epoch =  38 iteration =  78 batch loss =  0.82768214
epoch =  38 iteration =  78 accuracy =  0.8125
epoch =  39 iteration =  78 batch loss =  0.8650338
epoch =  39 iteration =  78 accuracy =  0.6875
epoch =  40 iteration =  78 batch loss =  0.87373304
epoch =  40 iteration =  78 accuracy =  0.8125
epoch =  41 iteration =  78 batch loss =  0.45015824
epoch =  41 iteration =  78 accuracy =  0.8125
epoch =  42 iteration =  78 batch loss =  0.766863
epoch =  42 iteration =  78 accuracy =  0.875
epoch =  43 iteration =  78 batch loss =  0.70051116
epoch =  43 iteration =  78 accuracy =  0.75
epoch =  44 iteration =  78 batch loss =  0.40852866
epoch =  44 iteration =  78 accuracy =  0.9375
epoch =  45 iteration =  78 batch loss =  1.0739012
epoch =  45 iteration =  78 accuracy =  0.6875
epoch =  46 iteration =  78 batch loss =  0.6007951
epoch =  46 iteration =  78 accuracy =  0.6875
epoch =  47 iteration =  78 batch loss =  0.7600157
epoch =  47 iteration =  78 accuracy =  0.8125
epoch =  48 iteration =  78 batch loss =  0.559161
epoch =  48 iteration =  78 accuracy =  0.875
epoch =  49 iteration =  78 batch loss =  0.76560664
epoch =  49 iteration =  78 accuracy =  0.6875
epoch =  50 iteration =  78 batch loss =  0.9605765
epoch =  50 iteration =  78 accuracy =  0.6875
epoch =  51 iteration =  78 batch loss =  0.64245546
epoch =  51 iteration =  78 accuracy =  0.8125
epoch =  52 iteration =  78 batch loss =  0.77759624
epoch =  52 iteration =  78 accuracy =  0.75
epoch =  53 iteration =  78 batch loss =  0.6687389
epoch =  53 iteration =  78 accuracy =  0.75
epoch =  54 iteration =  78 batch loss =  0.73263156
```

```
epoch =  54 iteration =  78 accuracy =  0.75
epoch =  55 iteration =  78 batch loss =  0.60453266
epoch =  55 iteration =  78 accuracy =  0.875
epoch =  56 iteration =  78 batch loss =  0.9267757
epoch =  56 iteration =  78 accuracy =  0.75
epoch =  57 iteration =  78 batch loss =  0.57138175
epoch =  57 iteration =  78 accuracy =  0.875
epoch =  58 iteration =  78 batch loss =  0.4603182
epoch =  58 iteration =  78 accuracy =  0.9375
epoch =  59 iteration =  78 batch loss =  0.60585
epoch =  59 iteration =  78 accuracy =  0.875
epoch =  60 iteration =  78 batch loss =  0.597418
epoch =  60 iteration =  78 accuracy =  0.8125
epoch =  61 iteration =  78 batch loss =  0.40400442
epoch =  61 iteration =  78 accuracy =  0.9375
epoch =  62 iteration =  78 batch loss =  0.92292666
epoch =  62 iteration =  78 accuracy =  0.6875
epoch =  63 iteration =  78 batch loss =  0.35870585
epoch =  63 iteration =  78 accuracy =  0.9375
epoch =  64 iteration =  78 batch loss =  0.3611383
epoch =  64 iteration =  78 accuracy =  0.9375
epoch =  65 iteration =  78 batch loss =  0.96354544
epoch =  65 iteration =  78 accuracy =  0.75
epoch =  66 iteration =  78 batch loss =  0.6400733
epoch =  66 iteration =  78 accuracy =  0.6875
epoch =  67 iteration =  78 batch loss =  0.76186305
epoch =  67 iteration =  78 accuracy =  0.8125
epoch =  68 iteration =  78 batch loss =  0.26888296
epoch =  68 iteration =  78 accuracy =  0.9375
epoch =  69 iteration =  78 batch loss =  0.8355322
epoch =  69 iteration =  78 accuracy =  0.875
epoch =  70 iteration =  78 batch loss =  0.28954083
epoch =  70 iteration =  78 accuracy =  0.875
epoch =  71 iteration =  78 batch loss =  0.68584883
epoch =  71 iteration =  78 accuracy =  0.75
epoch =  72 iteration =  78 batch loss =  0.6282463
epoch =  72 iteration =  78 accuracy =  0.75
epoch =  73 iteration =  78 batch loss =  0.41726923
epoch =  73 iteration =  78 accuracy =  0.9375
epoch =  74 iteration =  78 batch loss =  0.47691518
epoch =  74 iteration =  78 accuracy =  0.875
epoch =  75 iteration =  78 batch loss =  0.6457237
epoch =  75 iteration =  78 accuracy =  0.75
epoch =  76 iteration =  78 batch loss =  0.518641
epoch =  76 iteration =  78 accuracy =  0.8125
epoch =  77 iteration =  78 batch loss =  0.7065298
epoch =  77 iteration =  78 accuracy =  0.8125
epoch =  78 iteration =  78 batch loss =  0.67224526
epoch =  78 iteration =  78 accuracy =  0.8125
epoch =  79 iteration =  78 batch loss =  0.5928065
epoch =  79 iteration =  78 accuracy =  0.8125
epoch =  80 iteration =  78 batch loss =  0.3976013
epoch =  80 iteration =  78 accuracy =  0.9375
epoch =  81 iteration =  78 batch loss =  0.49525172
epoch =  81 iteration =  78 accuracy =  0.875
epoch =  82 iteration =  78 batch loss =  0.25990456
epoch =  82 iteration =  78 accuracy =  1.0
```

```
epoch =  83 iteration =  78 batch loss =  0.8000408
epoch =  83 iteration =  78 accuracy =  0.6875
epoch =  84 iteration =  78 batch loss =  0.30215442
epoch =  84 iteration =  78 accuracy =  0.9375
epoch =  85 iteration =  78 batch loss =  0.47866255
epoch =  85 iteration =  78 accuracy =  0.875
epoch =  86 iteration =  78 batch loss =  0.44681364
epoch =  86 iteration =  78 accuracy =  0.9375
epoch =  87 iteration =  78 batch loss =  0.3899202
epoch =  87 iteration =  78 accuracy =  0.875
epoch =  88 iteration =  78 batch loss =  0.3874218
epoch =  88 iteration =  78 accuracy =  0.875
epoch =  89 iteration =  78 batch loss =  0.5285255
epoch =  89 iteration =  78 accuracy =  0.75
epoch =  90 iteration =  78 batch loss =  0.328458
epoch =  90 iteration =  78 accuracy =  1.0
epoch =  91 iteration =  78 batch loss =  0.9653034
epoch =  91 iteration =  78 accuracy =  0.75
epoch =  92 iteration =  78 batch loss =  0.46339768
epoch =  92 iteration =  78 accuracy =  0.875
epoch =  93 iteration =  78 batch loss =  0.5583911
epoch =  93 iteration =  78 accuracy =  0.8125
epoch =  94 iteration =  78 batch loss =  0.23614243
epoch =  94 iteration =  78 accuracy =  0.9375
epoch =  95 iteration =  78 batch loss =  0.24790463
epoch =  95 iteration =  78 accuracy =  0.9375
epoch =  96 iteration =  78 batch loss =  0.60999167
epoch =  96 iteration =  78 accuracy =  0.8125
epoch =  97 iteration =  78 batch loss =  0.5446727
epoch =  97 iteration =  78 accuracy =  0.8125
epoch =  98 iteration =  78 batch loss =  0.21675661
epoch =  98 iteration =  78 accuracy =  1.0
epoch =  99 iteration =  78 batch loss =  0.4145758
epoch =  99 iteration =  78 accuracy =  0.9375
(5, 5, 1, 32)
W_c[0]: [[[ 0.0035654  -0.01668287 -0.00635505 -0.04971165  0.07778414
   -0.06227698 -0.01140072  0.02950826 -0.01826498  0.02485241
   -0.04311982  0.07689416  0.09510606  0.03057814  0.03537707
    0.00782259 -0.06537435  0.0740059  -0.09380892 -0.03976038
   -0.09891572 -0.02091868  0.0769275   0.00291841  0.05262141
   -0.05285369  0.04699641  0.04883552 -0.00849376 -0.08096495
    0.06848155  0.05373408]]

 [[-0.06130788 -0.05090711 -0.05848607 -0.01086143  0.00933647
    0.0468406   0.01712093  0.03243702  0.0935392   0.06498015
   -0.00564435  0.0475246  -0.06331791  0.0693498  -0.01316477
    0.0713117   0.02914519 -0.00519885 -0.07564902 -0.06460368
    0.01604402  0.05323795 -0.09011706 -0.06342962 -0.04561734
   -0.08183501  0.0369459  -0.03501617  0.038773    0.04224622
    0.07616794 -0.08210459]]

 [[ 0.03689568 -0.04071863  0.06087146 -0.02640369 -0.03774048
    0.04878145 -0.00801135  0.0120005  -0.02021366  0.04803716
    0.00981667 -0.00821517  0.0172624   0.03304565 -0.06380174
    0.10306112 -0.07392209 -0.10348194  0.0954367   0.02147155
   -0.00248508  0.08130141  0.01708622 -0.08389644 -0.05222376
   -0.03247355  0.04379009  0.01808399  0.03626723 -0.03313493
```

```
   0.09326153 -0.02281579]]

 [[ 0.09663246  0.0890661   0.07982831  0.09197827  0.07419552
  -0.05468158 -0.01268473  0.05183769  0.01989216  0.04611775
  -0.04557339 -0.02034082 -0.03642407 -0.09528469 -0.00428883
   0.09414041 -0.1069051   0.00351894  0.07518518 -0.03286436
   0.00874519 -0.04354093 -0.0308302  -0.04936728 -0.08213985
  -0.02749265  0.07022414  0.04155891  0.09230305 -0.06010405
   0.0596684  -0.03270367]]

 [[ 0.00040298 -0.03848898  0.06526959  0.06848157 -0.02838744
   0.0894205   0.06035215 -0.05601089 -0.06823834  0.03026815
   0.07997902  0.0062329   0.06391221 -0.08901974  0.04378831
  -0.04344945 -0.11332552  0.0677335  -0.0320106   0.06690422
  -0.04250936 -0.1164019  -0.0159106  -0.07457875 -0.05408761
   0.05584051 -0.05084041 -0.02092315  0.02647152  0.0104445
  -0.00676585 -0.0956274 ]]]
Means for first layer: [0.016132174, 0.029338859, 0.01393891, 0.08973887,
0.031540327, 0.01676252, 0.22206718, 0.019344307, 0.013705693, 0.012988208
, 0.037079323, 0.023153018, 0.01855602, 0.13712077, 0.041094296, 0.0367681
5, 0.098006554, 0.016511103, 0.025684064, 0.027391933, 0.05253628, 0.03408
8288, 0.03209313, 0.040605433, 0.029922921, 0.035356913, 0.3311147, 0.0226
50024, 0.02995678, 0.034102045, 0.020353775, 0.029478675]
Standard Deviation for first layer: [0.02915756, 0.038086936, 0.019988138,
 0.082939364, 0.037936207, 0.027575852, 0.10899919, 0.024041682, 0.0237845
98, 0.019124055, 0.055252455, 0.030977795, 0.023042217, 0.086668625, 0.035
565533, 0.057067554, 0.09598874, 0.030323485, 0.051125195, 0.031572353, 0.
06088244, 0.04654906, 0.043900803, 0.063050024, 0.042631447, 0.046884242,
0.16524898, 0.028067151, 0.05684843, 0.04443655, 0.027996512, 0.043884825]
```

```
C:\Users\oyeoy\Anaconda3\lib\site-packages\ipykernel_launcher.py:273: Matp
lotlibDeprecationWarning: Adding an axes using the same arguments as a pre
vious axes currently reuses the earlier instance.  In a future version, a
new instance will always be created and returned.  Meanwhile, this warning
 can be suppressed, and the future behavior ensured, by passing a unique l
abel to each axes instance.
```

```
Means for second layer: [0.027298879, 0.035685092, 0.039673265, 0.03372283
7, 0.023302646, 0.036286287, 0.024474269, 0.0217095, 0.03454492, 0.0251590
61, 0.039617248, 0.031754762, 0.045082152, 0.027849874, 0.03100977, 0.0250
41549, 0.019566825, 0.023994189, 0.043361876, 0.04382928, 0.019488882, 0.0
32915432, 0.022974387, 0.040203024, 0.031487584, 0.03400421, 0.036882248,
0.027963633, 0.041905727, 0.039056685, 0.04042995, 0.030967383, 0.02143590
5, 0.045604263, 0.03226313, 0.032411434, 0.035416115, 0.03084651, 0.040136
755, 0.048507094, 0.040806767, 0.032124583, 0.06491389, 0.044385165, 0.035
522502, 0.045388885, 0.032178864, 0.027475057, 0.031259637, 0.040684305, 0
.026635565, 0.018128056, 0.04145645, 0.03401644, 0.035258997, 0.037756484,
 0.023261363, 0.039127298, 0.044950772, 0.024582645, 0.025254158, 0.016750
393, 0.03599538, 0.04138647]
Standard Deviations for second layer: [0.061776217, 0.061587073, 0.0640462
2, 0.060734514, 0.051323842, 0.054099385, 0.06820811, 0.045092247, 0.05901
3527, 0.044423725, 0.07956225, 0.061532382, 0.06505881, 0.05579189, 0.0533
4235, 0.04876248, 0.045079656, 0.05396011, 0.07569376, 0.078639425, 0.0356
81777, 0.0620447, 0.04612172, 0.056052435, 0.062345088, 0.056784045, 0.075
62206, 0.060737666, 0.073725335, 0.073596686, 0.07487188, 0.055628233, 0.0
53808376, 0.07811466, 0.06707132, 0.06298042, 0.054162603, 0.066521585, 0.
063783616, 0.082595944, 0.081171766, 0.06784395, 0.12339037, 0.0820339, 0.
```

```
058915436, 0.07345317, 0.066693954, 0.051331073, 0.042380318, 0.065567166,
 0.061258085, 0.03704379, 0.08340636, 0.059124853, 0.074299484, 0.07128134
, 0.044010922, 0.073321484, 0.09325731, 0.047742385, 0.056028005, 0.036077
257, 0.066954054, 0.07184373]
test accuracy 0.569
```

In [ ]: