به نام خدا

درس: مقدمه‌ای بر یادگیری ماشین

استاد: دکتر صالح کلیبر

گزارش تمرین شماره ۴

سیّدمحمّدامین منصوری طهرانی

۹۴۱۰۵۱۷۴

۲ شبکه عصبی تمام‌متصل

۱. ابتدا به توضیح خلاصه کد می‌پردازیم و سپس نتایج آن را می‌آوریم.

```python
import numpy as np
import pickle
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tqdm import tqdm


def next_batch(num, data, labels):

    """Return a total of `num` random samples and labels."""
    idx = np.arange(0, len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data[i] for i in idx]
    labels_shuffle = [labels[i] for i in idx]

    return np.asarray(data_shuffle), np.asarray(labels_shuffle)


pickle_in = open('train.data', 'rb')
train_data = pickle.load(pickle_in)

pickle_in = open('test.data', 'rb')
test_data = pickle.load(pickle_in)

x_train, x_validation, y_train, y_validation = train_test_split(train_data['data'], train_data['labels'], test_size=0.2)

x_train = np.reshape(x_train, [len(x_train), 784])
x_validation = np.reshape(x_validation, [len(x_validation), 784])

X = tf.placeholder(dtype=tf.float32, shape=[None, 784])
Y = tf.placeholder(dtype=tf.float32, shape=[None, 10])
```

در قسمت فوق یک تابع برای انتخاب رندم batch بعدی تعریف شده است که اعدادی رندم تولید کرده و داده متناظر این اندیس‌ها را از داده‌های ورودی تابع به همراه برچسب آن‌ها برمی‌دارد و در خروجی یک batch به همراه برچسب آن تحویل می‌دهد.

بعد از آن داده لود شده و داده‌های تست و آموزش به نسبت خواسته شده به همراه برچسب‌های آن‌ها جدا می‌شوند. از آن‌جایی که این شبکه تمام‌متصل است، نمی‌توان ورودی‌ها را به همان صورتی که هستند به شبکه داد و باید به صورت یک بردار ۲۸ × ۲۸ یا ۷۸۴ تایی به ورودی شبکه داده شوند. این تغییر شکل برای داده‌های آموزش و اعتبارسنجی انجام شده و در نهایت placeholder ها برای استفاده در Session و feed کردن مقادیر مورد نیاز آن‌ها تعریف شده‌اند. برای ورودی آرگومان اول شکل آن‌ها نامعین است تا بسته به اندازه batch مشخص شود و آرگومان دوم همان تعداد نورون‌های ورودی یا ۷۸۴ است. برای خروجی نیز وضعیت مشابه است با این تفاوت که خروجی ۱۰ عدد مربوط به ۱۰ حرف مشخص شده است.

```python
n_hidden_layer = 50
learning_rate = 0.01
batch_size = 64
std = 0.1
training_iterations = 60

with tf.name_scope(name="Hidden_layer"):
    w1 = tf.Variable(tf.random_normal(shape=(784, n_hidden_layer), mean=0, stddev=std, seed=1), name="W")
    b1 = tf.Variable(tf.zeros([n_hidden_layer]), name="B")

    hidden_layer_in = tf.matmul(X, w1) + b1
    hidden_layer_output = tf.sigmoid(hidden_layer_in)

with tf.name_scope("output_layer"):
    w2 = tf.Variable(tf.random_normal(shape=(n_hidden_layer, 10), mean=0, stddev=std, seed=100), name="W")
    b2 = tf.Variable(tf.zeros([10]), name="B")  # 10, is the number of classes since we are classifying the data into 10
    #  different classes as is mentioned in the pdf file.

    output_layer_in = tf.matmul(hidden_layer_output, w2) + b2
    output = tf.nn.softmax(output_layer_in)


cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y, logits=output))

optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(output, 1), tf.argmax(Y, 1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, dtype=tf.float32), name="accuracy")

confusion_matrix = tf.confusion_matrix(tf.argmax(Y, 1), tf.argmax(output, 1), num_classes=10)
```

در ابتدا پارامترهای مدل تعیین شده‌اند و عدد std برای مقداردهی اولیه وزن‌ها از توزیع گاوسی استفاده شده است و بدون تعیین مناسب آن مدل نتیجه خوبی بدست نمی‌دهد. در ادامه لایه مخفی تعریف شده و وزن‌های آن را به همراه بایاس جمع‌شونده تعریف می‌کنیم. عملیات ضرب و عبور از تابع فعالیت سیگموید انجام شده و برای لایه خروجی مشابه همین اتفاق را خواهیم داشت به جز این‌که در خروجی از تابع فعالیت softmax استفاده می‌شود تا احتمال طبقه‌بندی هر حرف مشخص شود.

در ادامه تابع هزینه و بهینه‌ساز تعریف می‌شوند و نتیجه پیش‌بینی‌ها با مقایسه خروجی softmax و برچسب‌ها صورت گرفته و با آن دقت و نهایتاً ماتریس درهم‌ریختگی محاسبه می‌شوند.

```
confusion_matrix_summary = tf.summary.tensor_summary('Confusion Matrix', confusion_matrix)

train_loss_summary = tf.summary.scalar('loss_train', cross_entropy)
train_accuracy_summary = tf.summary.scalar('train_accuracy', accuracy)

validation_loss_summary = tf.summary.scalar('loss_validation', cross_entropy)
validation_accuracy_summary = tf.summary.scalar('validation_accuracy', accuracy)

merge = tf.summary.merge_all()

init = tf.global_variables_initializer()

accuracy_test_acc = 0

with tf.Session() as sess:

    sess.run(init)

    summary_writer = tf.summary.FileWriter('sigmoid_loss_event')
    # ############################ Training ############################
    for i in tqdm(range(training_iterations)):

        counter = 0

        while counter <= len(x_train)//batch_size:

            counter += 1

            x_batch, y_batch = next_batch(batch_size, x_train, y_train)
            sess.run(optimizer,  feed_dict={X: x_batch, Y: y_batch})

            summary_writer.add_summary((sess.run(train_loss_summary, feed_dict={X: x_batch, Y: y_batch})), counter)
```

در ابتدای این قسمت summary ها برای استفاده در تنسوربرد تعریف می‌شوند. برای آموزش و اعتبارسنجی به صورت جداگانه summary هزینه و دقت گرفته شده و ماتریس درهم‌ریختگی نیز به صورت تنسور به summary داده می‌شود. در نهایت همه این summary ها ادغام می‌شوند. یک session باز شده و همه رویدادهای این کد در پوشه‌ای نوشته می‌شوند. در فرآیند آموزش به تعداد epoch ها، هر بار تا زمانی که به تعداد کل داده‌ها آموزش ندیده باشد، هر بار یک batch جدید گرفته می‌شود و بهینه‌ساز یک مرحله جلو می‌رود. Summary متغیر تابع هزینه هم برای استفاده در تنسوربرد ذخیره می‌شود.

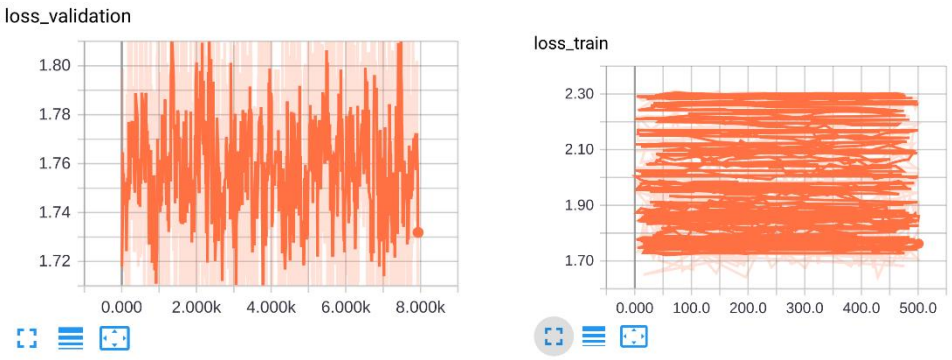
```
    # ############################ Testing ############################

    counter = 0

    while counter <= len(x_validation):
        counter += 1

        x_batch, y_batch = next_batch(batch_size, x_validation, y_validation)
        # sess.run(optimizer, feed_dict={X: x_batch, Y: y_batch})

        summary_writer.add_summary((sess.run(validation_loss_summary, feed_dict={X: x_batch, Y: y_batch})), counter)

    x_batch, y_batch = x_validation, y_validation
    accuracy_test, loss_test = sess.run((accuracy, cross_entropy), feed_dict={X: x_batch, Y: y_batch})
    # summary_writer.add_summary((sess.run(confusion_matrix_summary, feed_dict={X: x_batch, Y: y_batch})))
    cm = sess.run(confusion_matrix, feed_dict={X: x_batch, Y: y_batch})
    print(cm)
    print("The test accuracy is:" + str(accuracy_test))
```

برای اعتبارسنجی نیز به تعداد کل داده‌های آن هر بار یک batch گرفته و برای آن تابع هزینه را محاسبه می‌کنیم و به summary اضافه می‌کنیم. برای محاسبه دقت نیز کل داده‌های اعتبارسنجی را با هم داده و دقت را بدست می‌آوریم. ماتریس درهم‌ریختگی نیز چاپ می‌شود.

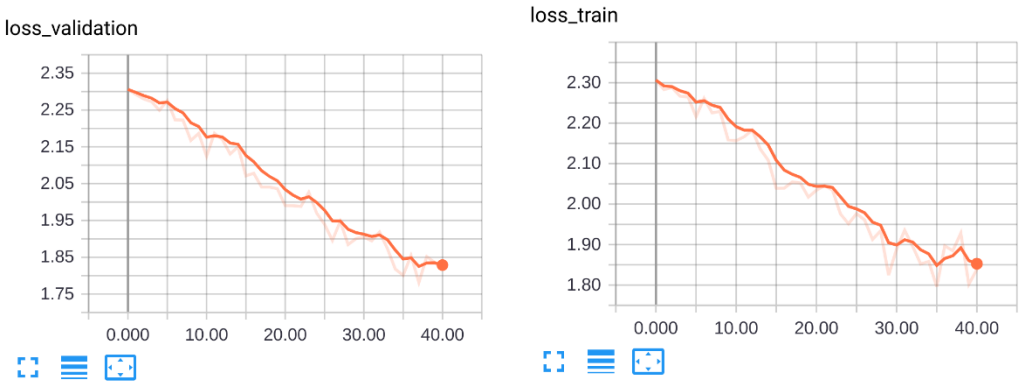
نتایج کد زیر در ادامه آورده شده‌است.

```
 48%|     | 29/60 [04:54<04:25,  8.57s/it]
 50%|     | 30/60 [05:02<04:15,  8.50s/it]
 52%|     | 31/60 [05:10<04:02,  8.37s/it]
 53%|     | 32/60 [05:18<03:49,  8.21s/it]
 55%|     | 33/60 [05:25<03:30,  7.81s/it]
 57%|     | 34/60 [05:31<03:09,  7.31s/it]
 58%|     | 35/60 [05:37<02:52,  6.92s/it]
 60%|     | 36/60 [05:43<02:40,  6.69s/it]
 62%|     | 37/60 [05:49<02:29,  6.52s/it]
 63%|     | 38/60 [05:55<02:19,  6.36s/it]
 65%|     | 39/60 [06:02<02:14,  6.38s/it]
 67%|     | 40/60 [06:08<02:09,  6.46s/it]
 68%|     | 41/60 [06:15<02:00,  6.36s/it]
 70%|     | 42/60 [06:21<01:52,  6.25s/it]
 72%|     | 43/60 [06:27<01:45,  6.23s/it]
 73%|     | 44/60 [06:33<01:38,  6.17s/it]
 75%|     | 45/60 [06:39<01:35,  6.35s/it]
 77%|     | 46/60 [06:47<01:32,  6.59s/it]
 78%|     | 47/60 [06:53<01:23,  6.40s/it]
 80%|     | 48/60 [06:59<01:15,  6.33s/it]
 82%|     | 49/60 [07:05<01:10,  6.37s/it]
 83%|     | 50/60 [07:12<01:04,  6.50s/it]
 85%|     | 51/60 [07:18<00:57,  6.41s/it]
 87%|     | 52/60 [07:24<00:50,  6.36s/it]
 88%|     | 53/60 [07:31<00:43,  6.27s/it]
 90%|     | 54/60 [07:37<00:37,  6.20s/it]
 92%|     | 55/60 [07:43<00:30,  6.14s/it]
 93%|     | 56/60 [07:49<00:24,  6.11s/it]
 95%|     | 57/60 [07:55<00:18,  6.18s/it]
 97%|     | 58/60 [08:01<00:12,  6.23s/it]
 98%|     | 59/60 [08:08<00:06,  6.28s/it]
100%|     | 60/60 [08:14<00:00,  6.35s/it]
[[673  13   0  15   7   7  12  21  16  12]
 [ 14 665   0  37  10   9  22  29  23  10]
 [  6   8   0  42 405  11 302   4  15   9]
 [ 17  18   0 693   2   4  19  22  15  13]
 [ 12  13   0  11 634  14  14  12  21   6]
 [ 10   2   0   9  10 742  19  11  19   8]
 [ 16  10   0  17   7  11 669  14  25  16]
 [ 21  11   0  17   6   9  14 702  26   9]
 [ 15  10   0  17  14  15  23  22 664  46]
 [ 12   7   0  14   5  18   9  12  24 706]]
The test accuracy is:0.7685
```

نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:



در شکل‌های فوق تابع هزینه برای هر batch محاسبه شده و در شکل‌های زیر برای هر epoch رسم شده‌است.

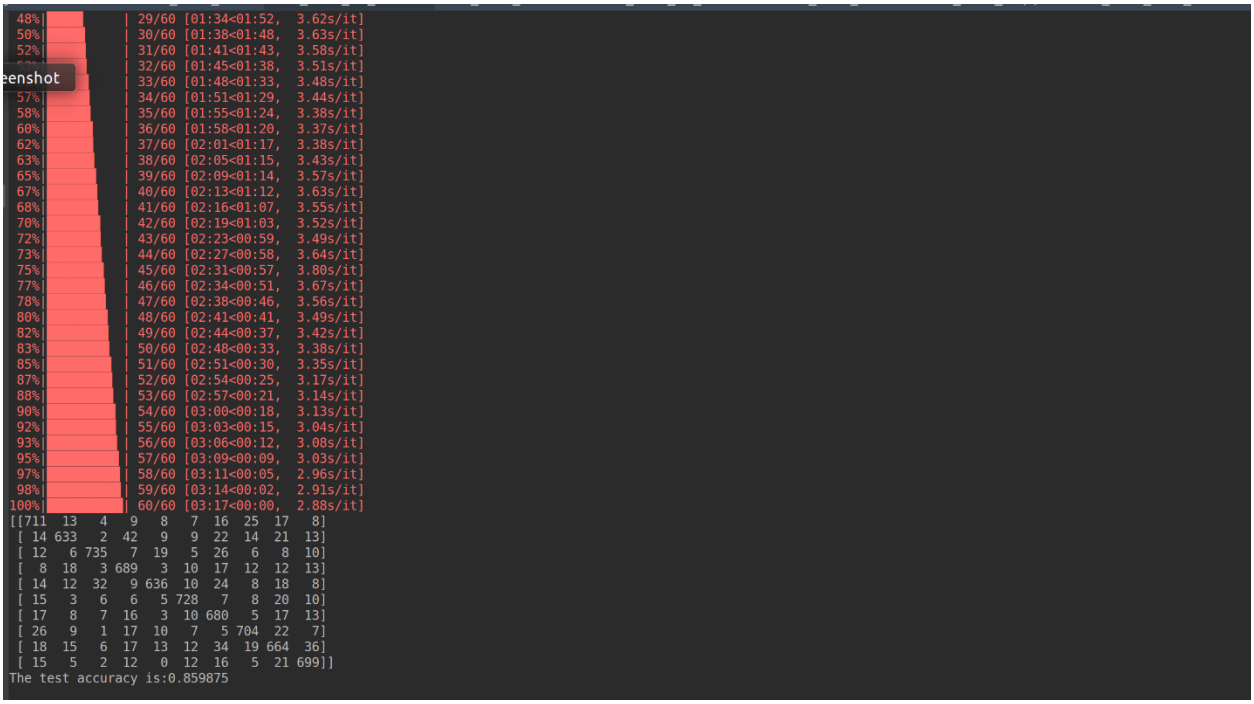


۲. در این قسمت فقط تابع فعالیت به $tanh$ تغییر کرده و کد آن نیاز به توضیح مجدد ندارد. نتایج آن در زیر ملاحظه می‌شود و دقت مقدار قابل توجهی بهبود داشته است.

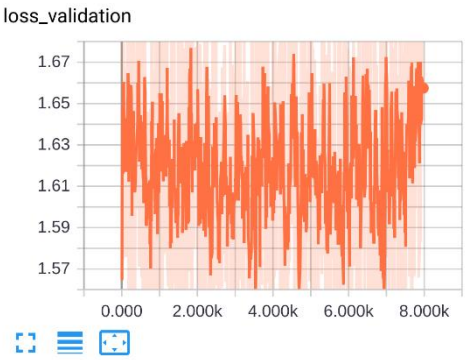
علت بهبود نتیجه با تغییر تابع فعالیت: تابع $tanh$ را می‌توان به صورت زیر برحسب سیگموید نوشت:

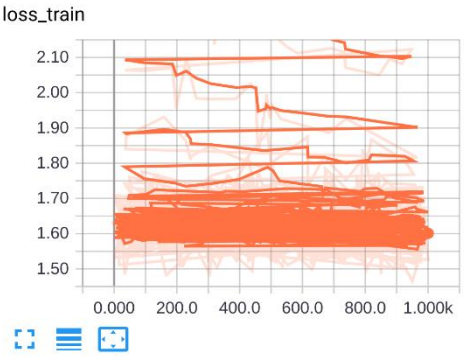
$$tanh(x) = 2 \times sigmoid(2x) - 1$$

و به سادگی مشاهده می‌شود که گرادیان آن در نقطه مشابه از سیگموید بیشتر است و گام‌های بزرگتری دارد و این می‌تواند یک علت برای رسیدن به نقطه کمینه بهتر باشد. برای بهتر کردن دقت می‌توان از تکنیک‌های بهبود روش گرادیان مانند Nadam،Adam و RMSProp نیز استفاده کرد که در سوال این اجازه داده نشده است.

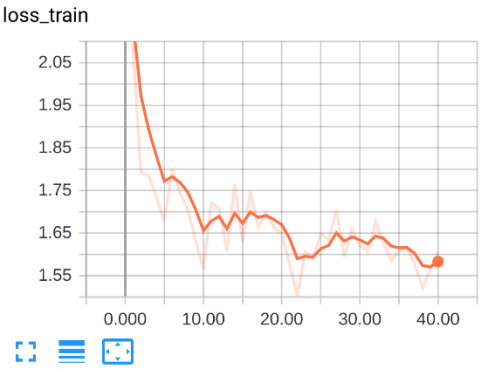


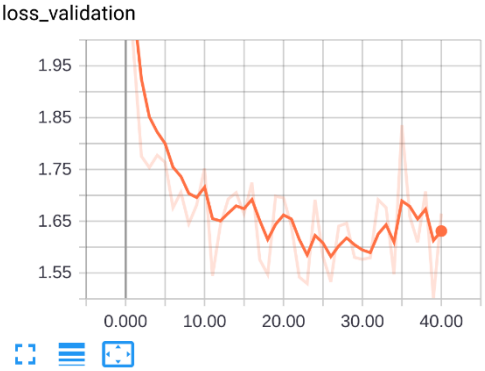
نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:





Epoch by epoch

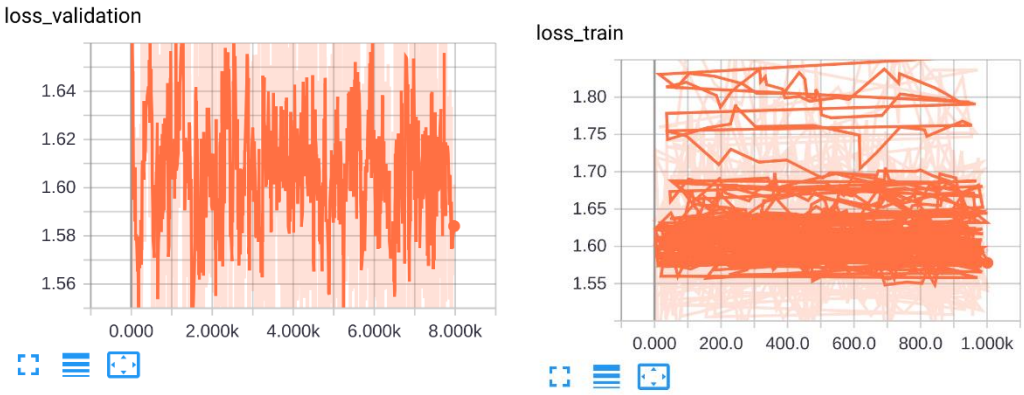




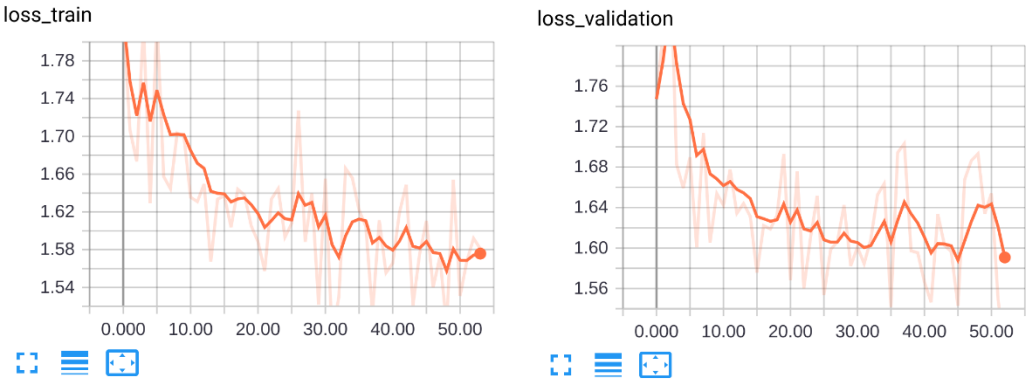
۳. باز هم تنها تغییر تعداد نورون لایه مخفی به ۵۰۰ است و توضیح اضافه‌ای وجود ندارد.

```
 48%|     | 29/60 [02:20<02:34,  4.97s/it]
 50%|     | 30/60 [02:25<02:25,  4.86s/it]
 52%|     | 31/60 [02:29<02:18,  4.77s/it]
 53%|     | 32/60 [02:34<02:11,  4.71s/it]
 55%|     | 33/60 [02:38<02:05,  4.63s/it]
 57%|     | 34/60 [02:42<01:53,  4.38s/it]
 58%|     | 35/60 [02:47<01:49,  4.37s/it]
 60%|     | 36/60 [02:51<01:43,  4.29s/it]
 62%|     | 37/60 [02:55<01:37,  4.24s/it]
 63%|     | 38/60 [02:59<01:31,  4.14s/it]
 65%|     | 39/60 [03:03<01:24,  4.05s/it]
 67%|     | 40/60 [03:07<01:20,  4.01s/it]
 68%|     | 41/60 [03:11<01:16,  4.01s/it]
 70%|     | 42/60 [03:15<01:13,  4.08s/it]
 72%|     | 43/60 [03:18<01:06,  3.90s/it]
 73%|     | 44/60 [03:22<00:59,  3.73s/it]
 75%|     | 45/60 [03:25<00:53,  3.58s/it]
 77%|     | 46/60 [03:28<00:48,  3.48s/it]
 78%|     | 47/60 [03:31<00:44,  3.41s/it]
 80%|     | 48/60 [03:35<00:40,  3.34s/it]
 82%|     | 49/60 [03:38<00:36,  3.30s/it]
 83%|     | 50/60 [03:41<00:32,  3.26s/it]
 85%|     | 51/60 [03:44<00:29,  3.24s/it]
 87%|     | 52/60 [03:47<00:25,  3.23s/it]
 88%|     | 53/60 [03:51<00:22,  3.27s/it]
 90%|     | 54/60 [03:54<00:20,  3.35s/it]
 92%|     | 55/60 [03:57<00:16,  3.31s/it]
 93%|     | 56/60 [04:01<00:13,  3.31s/it]
 95%|     | 57/60 [04:04<00:10,  3.34s/it]
 97%|     | 58/60 [04:07<00:06,  3.31s/it]
 98%|     | 59/60 [04:11<00:03,  3.29s/it]
100%|     | 60/60 [04:14<00:00,  3.25s/it]
[[681  10   3  15   9   4  15  19  13  11]
 [ 11 684   2  44  11  13  15  22   6   7]
 [  8  15 699  13  20  13  33   7  12   7]
 [ 11  12   1 664   2   8  12  13   9  10]
 [ 11  17  31  13 654  22  22   8  29   7]
 [ 13   7   4  26  12 703  11   6  21   9]
 [ 20  23  18  21   7   6 681   6  24  12]
 [ 11  10   1  11   5  12  12 653  18  12]
 [ 20  13   1  21  17  13  22  23 675  29]
 [ 18   9   3  12   6   9  11  10  16 719]]
The test accuracy is:0.851625
```

نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:



Epoch by epoch

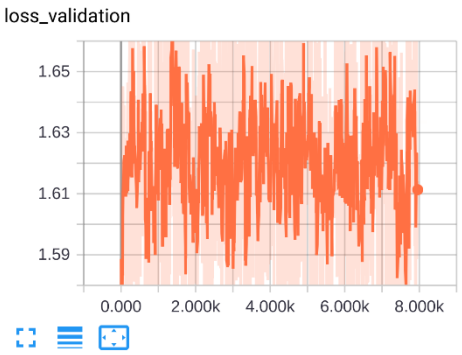


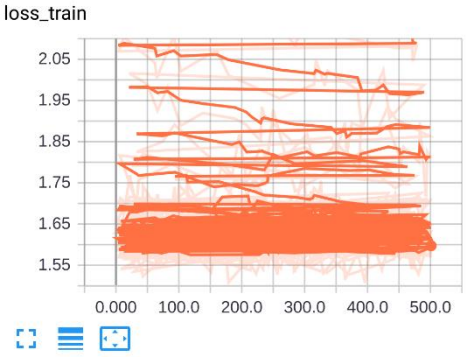
۴. تنها تغییر کد اضافه شدن یک لایه است و اکنون دو لایه داریم که هر کدام ۱۰۰ نورون دارند.

```
with tf.name_scope(name="Hidden_layer_1"):
    w1 = tf.Variable(tf.random_normal(shape=(784, n_hidden_layer), mean=0, stddev=std, seed=1), name="W")
    b1 = tf.Variable(tf.zeros([n_hidden_layer]), name="B")

    hidden_layer_in_1 = tf.matmul(X, w1) + b1
    hidden_layer_output_1 = tf.tanh(hidden_layer_in_1)

with tf.name_scope(name="Hidden_layer_2"):
    w2 = tf.Variable(tf.random_normal(shape=(n_hidden_layer, n_hidden_layer), mean=0, stddev=std, seed=1), name="W")
    b2 = tf.Variable(tf.zeros([n_hidden_layer]), name="B")

    hidden_layer_in_2 = tf.matmul(hidden_layer_output_1, w2) + b2
    hidden_layer_output_2 = tf.tanh(hidden_layer_in_2)

with tf.name_scope("output_layer"):
    w3 = tf.Variable(tf.random_normal(shape=(n_hidden_layer, 10), mean=0, stddev=std), name="W")
    b3 = tf.Variable(tf.zeros([10]), name="B")  # 10, is the number of classes since we are classifying the data into 10
    #  different classes as is mentioned in the pdf file.

    output_layer_in = tf.matmul(hidden_layer_output_2, w3) + b3
    output = tf.nn.softmax(output_layer_in)

```

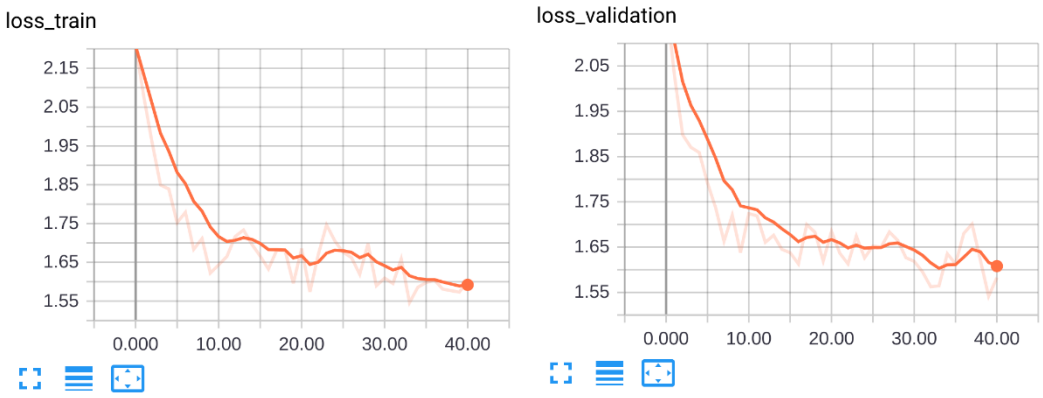
```
 23%|██▎       | 14/60 [00:20<01:05,  1.42s/it]
 25%|██▌       | 15/60 [00:22<01:02,  1.39s/it]
 27%|██▋       | 16/60 [00:23<01:00,  1.38s/it]
 28%|██▊       | 17/60 [00:24<00:58,  1.36s/it]
 30%|███       | 18/60 [00:26<00:56,  1.35s/it]
 32%|███▏      | 19/60 [00:27<00:55,  1.34s/it]
 33%|███▎      | 20/60 [00:28<00:53,  1.35s/it]
 35%|███▌      | 21/60 [00:30<00:55,  1.44s/it]
 37%|███▋      | 22/60 [00:31<00:53,  1.41s/it]
 38%|███▊      | 23/60 [00:33<00:51,  1.40s/it]
 40%|████      | 24/60 [00:34<00:54,  1.51s/it]
 42%|████▏     | 25/60 [00:36<00:50,  1.45s/it]
 43%|████▎     | 26/60 [00:37<00:47,  1.41s/it]
 45%|████▌     | 27/60 [00:38<00:45,  1.38s/it]
 47%|████▋     | 28/60 [00:39<00:42,  1.32s/it]
 48%|████▊     | 29/60 [00:41<00:39,  1.27s/it]
 50%|█████     | 30/60 [00:42<00:37,  1.24s/it]
 52%|█████▏    | 31/60 [00:43<00:35,  1.22s/it]
 53%|█████▎    | 32/60 [00:44<00:33,  1.21s/it]
 55%|█████▌    | 33/60 [00:45<00:32,  1.20s/it]
 57%|█████▋    | 34/60 [00:47<00:31,  1.21s/it]
 58%|█████▊    | 35/60 [00:48<00:30,  1.20s/it]
 60%|██████    | 36/60 [00:49<00:28,  1.20s/it]
 62%|██████▏   | 37/60 [00:50<00:27,  1.20s/it]
 63%|██████▎   | 38/60 [00:52<00:28,  1.28s/it]
 65%|██████▌   | 39/60 [00:53<00:27,  1.30s/it]
 67%|██████▋   | 40/60 [00:54<00:25,  1.30s/it]
 68%|██████▊   | 41/60 [00:56<00:24,  1.29s/it]
 70%|███████   | 42/60 [00:57<00:23,  1.28s/it]
 72%|███████▏  | 43/60 [00:58<00:21,  1.27s/it]
 73%|███████▎  | 44/60 [00:59<00:20,  1.27s/it]
 75%|███████▌  | 45/60 [01:01<00:19,  1.27s/it]
 77%|███████▋  | 46/60 [01:02<00:17,  1.26s/it]
 78%|███████▊  | 47/60 [01:03<00:16,  1.25s/it]
 80%|████████  | 48/60 [01:04<00:15,  1.25s/it]
 82%|████████▏ | 49/60 [01:06<00:14,  1.28s/it]
 83%|████████▎ | 50/60 [01:07<00:13,  1.36s/it]
 85%|████████▌ | 51/60 [01:08<00:11,  1.33s/it]
 87%|████████▋ | 52/60 [01:10<00:10,  1.29s/it]
 88%|████████▊ | 53/60 [01:11<00:08,  1.27s/it]
 90%|█████████ | 54/60 [01:12<00:07,  1.25s/it]
 92%|█████████▏| 55/60 [01:13<00:06,  1.23s/it]
 93%|█████████▎| 56/60 [01:14<00:04,  1.22s/it]
 95%|█████████▌| 57/60 [01:16<00:03,  1.21s/it]
 97%|█████████▋| 58/60 [01:17<00:02,  1.21s/it]
 98%|█████████▊| 59/60 [01:18<00:01,  1.20s/it]
100%|██████████| 60/60 [01:19<00:00,  1.19s/it]
[[656  16   2   9   5   8  22  26   9  16]
 [  8 637   4  36  17  11  25  21  10   2]
 [  7  12 690   8  15  13  30  12   7   7]
 [ 14  19   3 703   5  17  18  20  12  11]
 [ 13  21  31   8 669  17  19   9  24   7]
 [  7   9   9  15  14 685   8  10  13  26]
 [  6  16  27  19   9  17 691   1  15   8]
 [ 31  10   6   6  18  16   8 689  18  12]
 [ 11  11   3  15  20  10  12  19 679  34]
 [ 11   9   7  14   3  15   9  10  20 688]]
The test accuracy is:0.848375
```

نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:





Epoch by epoch



مقایسه مدل دو لایه مخفی با ۲۰۰ نورون با مدل تک لایه مخفی با ۵۰۰ نورون: همان‌طور که در درس نیز داشتیم، طبق قضیه‌ای دسته بسیار وسعی از توابع به کمک یک لایه مخفی قابل تقریب زدن هستند به شرطی که تعداد نورون‌های لایه مخفی به اندازه کافی زیاد باشند. این تعداد کافی ممکن است خیلی زیاد باشد اما اگر شبکه عمیق شود، ترکیب وزن‌های دو لایه قابلیت $representation$ بسیار وسیع‌تری از یک لایه دارد و هم‌چنین تعداد پارامترهای آن نیز کمتر است و می‌تواند تقریب بهتری از تابع طبقه‌بند واقعی بدست دهد. در شبکه با ۵۰۰ نورون در لایه مخفی تعداد پارامترها برابر است با:

$$784 \times 500(layer1weights) + 500(layer1biases) + 500 \times 10(outputlayerweights) + 10(outputlayerbiases) = 397510$$

در شبکه با دو لایه مخفی که هر کدام ۱۰۰ نورون دارند داریم:

$$784 \times 100(layer1weights) + 100(layer1biases) + 100 \times 100(layer2weights) + 100(layer2biases) + 100 \times 10(outputlayerweights) + 10(outputlayerbiases) = 89610$$

بنابراین تعداد پارامترها حدود یک چهارم حالت قبل است و آموزش مطابق عکس نشان داده نیز حدود ۴ برابر سریع‌تر انجام شده و دقت دو شبکه تقریباً برابر شده است. قابل توجه است که شبکه دوم صرفاً به خاطر عمیق‌تر شدن به این دقت رسیده است و تعداد پارامترهای آن کم‌تر است و این قابلیت عمیق کردن شبکه در نمایش و تقریب توابع است.

۳ شبکه عصبی کانولوشنی

در ابتدا به توضیح خلاصه کل کد می‌پردازیم:

در قطعه زیر اندازه ضلع مربع تصویر و تعداد کلاس‌های خروجی (۱۰ عدد) و انحراف معیار مقداردهی اولیه از توزیع گاوسی مشخص شده و در ادامه آن وزن تک تک لایه‌ها به علاوه لایه تمام‌متصل و لایه خروجی تعریف

می‌شوند. لایه تمام‌متصل نیز flatten شده و خروجی آن که ۱۲۸ تصویر ۷ × ۷ است به ۲۵۶ نورون لایه تمام‌متصل وصل شده و این نورون‌ها به ۱۰ کلاس خروجی متصل می‌شوند.

```python
# Python 2.7
# By: Amin Mansouri

# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tqdm import tqdm
import pickle
from sklearn.model_selection import train_test_split

tf.reset_default_graph()

# image data input (img shape: 28*28)
n_input = 28

n_classes = 10
std = 0.1

weights = {
    'weight_conv_layer_1': tf.get_variable('W0', shape=(5, 5, 1, 64),
                                           initializer=tf.initializers.random_normal(stddev=std)
                                           ),
    'weight_conv_layer_2': tf.get_variable('W1', shape=(5, 5, 64, 64),
                                           initializer=tf.initializers.random_normal(stddev=std)),
    'weight_conv_layer_3': tf.get_variable('W2', shape=(5, 5, 64, 128),
                                           initializer=tf.initializers.random_normal(stddev=std)),
    'weight_conv_layer_4': tf.get_variable('W3', shape=(5, 5, 128, 128),
                                           initializer=tf.initializers.random_normal(stddev=std)),

    'w_fc': tf.get_variable('W_fc', shape=(7 * 7 * 128, 256), initializer=tf.initializers.random_normal(stddev=std)),
    'out': tf.get_variable('W_out', shape=(256, n_classes), initializer=tf.initializers.random_normal(stddev=std)),
```

در کد زیر، ابتدا بایاس‌ها تعریف شدند. دو تابع نیز تعریف کردم که اولی وظیفه کانولوشن دو بعدی با پدینگ مناسب و سپس اعمال تابع فعالیت است. تابع $pooling$ هم با گرفتن ورودی و گام $pooling$ خروجی نمونه‌برداری شده را بازمی‌گرداند. تابع $next_{batch}$ نیز مثل قبل است و نهایتاً داده‌ها لود می‌شوند.

```
biases = {
    'bias_conv_layer_1': tf.get_variable('B0', shape=64, initializer=tf.initializers.random_normal(stddev=std)),
    'bias_conv_layer_2': tf.get_variable('B1', shape=64, initializer=tf.initializers.random_normal(stddev=std)),
    'bias_conv_layer_3': tf.get_variable('B2', shape=128, initializer=tf.initializers.random_normal(stddev=std)),
    'bias_conv_layer_4': tf.get_variable('B3', shape=128, initializer=tf.initializers.random_normal(stddev=std)),
    # 'bias_conv_layer_3': tf.get_variable('B2', shape=128, initializer=tf.contrib.layers.xavier_initializer()),
    'b_fc': tf.get_variable('B_fc', shape=256, initializer=tf.initializers.random_normal(stddev=std)),
    'out': tf.get_variable('B_out', shape=n_classes, initializer=tf.initializers.random_normal(stddev=std)),
}


# noinspection PyShadowingNames
def conv2d(x, w, b, strides=1):
    # Conv2D wrapper, with bias and ReLU activation
    x = tf.nn.conv2d(x, w, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)


def maxpool2d(x, k=2):
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')


def next_batch(num, data, labels):

    """Return a total of `num` random samples and labels."""
    idx = np.arange(0, len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data[i] for i in idx]
    labels_shuffle = [labels[i] for i in idx]

    return np.asarray(data_shuffle), np.asarray(labels_shuffle)


pickle_in = open('train.data', 'rb')
train_data = pickle.load(pickle_in)
```

در زیر ابتدا داده‌های آموزش و اعتبارسنجی جدا می‌شوند و لایه‌های کانولوشنی و $pooling$ و تمام‌متصل و طبقه‌بندی تعریف می‌شوند. در انتها پارامترهای مدل تعریف شده‌اند که البته epoch ها بیشتر از مقدار زیر در نظر گرفته شد. متغیر $keep_{prob}$ برای عملیات $dropout$ در نظر گرفته شده و هنگام اجرای عملیات آموزش و تست مقداردهی می‌شود.

```
x_train, x_validation, y_train, y_validation = train_test_split(train_data['data'], train_data['labels'], test_size=0.2)

# both placeholders are of type float
x = tf.placeholder("float", [None, 28, 28, 1])
# x = tf.placeholder("float", [28, 28])
y = tf.placeholder("float", [None, n_classes])

# here we call the conv2d function we had defined above and pass the input image x, weights weight_conv_layer_1
# and bias bias_conv_layer_1.
conv1 = conv2d(x, weights['weight_conv_layer_1'], biases['bias_conv_layer_1'], strides=2)
# Max Pooling (down-sampling), this chooses the max value from a 2*2 matrix window and outputs a 14*14 matrix.
conv1_maxpool = maxpool2d(conv1, k=1)

conv2 = conv2d(conv1_maxpool, weights['weight_conv_layer_2'], biases['bias_conv_layer_2'], strides=2)
conv2_maxpool = maxpool2d(conv2, k=1)

conv3 = conv2d(conv2_maxpool, weights['weight_conv_layer_3'], biases['bias_conv_layer_3'], strides=1)

conv4 = conv2d(conv3, weights['weight_conv_layer_4'], biases['bias_conv_layer_4'], strides=1)

# Fully connected layer
# Reshape conv4 output to fit fully connected layer input
fc1 = tf.reshape(conv4, [-1, weights['w_fc'].get_shape().as_list()[0]])
fc1 = tf.add(tf.matmul(fc1, weights['w_fc']), biases['b_fc'])
fc1 = tf.nn.relu(fc1)
# Output, class prediction
# finally we multiply the fully connected layer with the weights and add a bias term.
out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])


training_iterations = 10
learning_rate = 0.001
batch_size = 128
keep_prob = tf.placeholder("float")
```

در کد زیر، ابتدا شکل داده‌ها به گونه‌ای تغییر داده می‌شود که یک بعد برای اندازه batch داشته باشند. تابع هزینه مثل قبل از نوع $cross_{entropy}$ و بهینه‌ساز مطابق خواسته سؤال از نوع $Adam$ تعریف شده و پیش‌بینی صحیح و دقت نیز مثل قبل محاسبه می‌شوند. در انتها هم $summary$ها برای استفاده در تنسوربرد ذخیره می‌شوند.

```
# Reshape training and testing image
train_X = x_train.reshape(-1, 28, 28, 1)
test_X = x_validation.reshape(-1, 28, 28, 1)

print([train_X.shape, test_X.shape])

train_y = y_train
test_y = y_validation

print(train_y.shape, test_y.shape)

prediction = out

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction, labels=y))

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

# Here you check whether the index of the maximum value of the predicted image is equal to the actual labelled image.
#  and both will be a column vector.
correct_prediction = tf.equal(tf.argmax(prediction, 1), tf.argmax(y, 1))

# calculate accuracy across all the given images and average them out.
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# summary settings:

train_loss_summary = tf.summary.scalar('loss_train', cost)
train_accuracy_summary = tf.summary.scalar('train_accuracy', accuracy)
train_summaries = tf.summary.merge([train_loss_summary, train_accuracy_summary])
# file_writer = tf.summary.FileWriter('./Output', sess.graph)

validation_loss_summary = tf.summary.scalar('loss_validation', cost)
validation_accuracy_summary = tf.summary.scalar('validation_accuracy', accuracy)
validation_summaries = tf.summary.merge([validation_loss_summary, validation_accuracy_summary])
# validation_file_writer = tf.summary.FileWriter(output_folder)

# Initializing the variables
init = tf.global_variables_initializer()

```

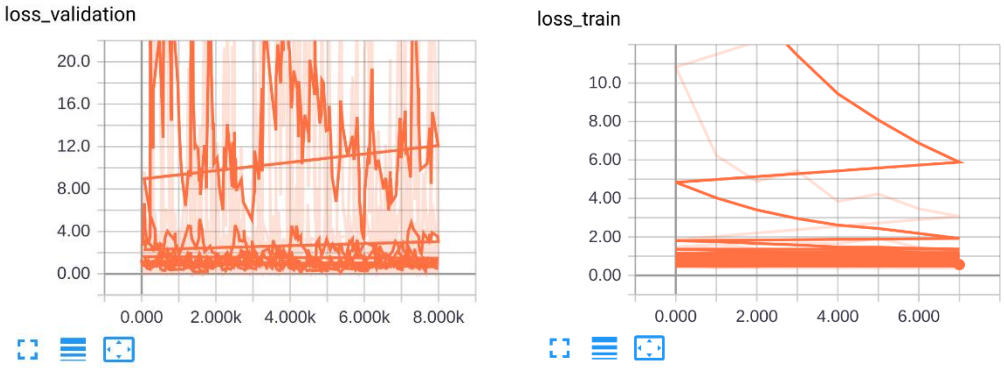
در کد زیر که به قسمت آموزش مربوط است، به تعداد $epoch$ ها هر بار تعدادی داده مدل را آموزش می‌دهند و هزینه در summary ذخیره می‌شود.هر چند وقت یک بار نیز دقت آموزش چاپ بر روی batch داده شده چاپ می‌شود.

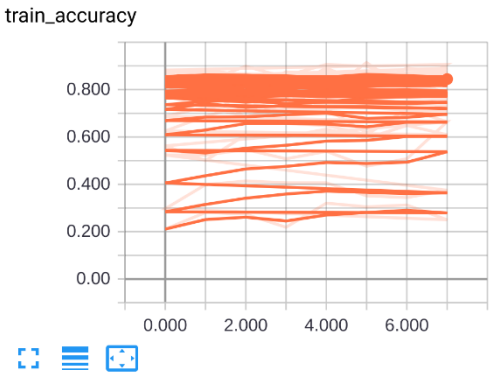
```
with tf.Session() as sess:
    sess.run(init)

    merge = tf.summary.merge_all()
    summary_writer_train = tf.summary.FileWriter('./Output_train', sess.graph)
    summary_writer_validation = tf.summary.FileWriter('./Output_validation', sess.graph)

    # ########################################## Training ######################################### #

    for i in range(training_iterations):

        for batch in tqdm(range(len(train_X) // batch_size // training_iterations)):

            counter += 1
            batch_x = train_X[(batch + i * len(train_X) // batch_size // training_iterations) *
                              batch_size:min((batch + 1 + i * len(train_X) // batch_size // training_iterations) *
                                             batch_size, len(train_X))]
            batch_y = train_y[(batch + i * len(train_X) // batch_size // training_iterations) *
                              batch_size:min((batch + 1 + i * len(train_X) // batch_size // training_iterations) *
                                             batch_size, len(train_y))]

            # Run optimization op (backprop).
            # Calculate batch loss and accuracy
            opt = sess.run(optimizer, feed_dict={x: batch_x,
                                                 y: batch_y})
            loss, acc = sess.run([cost, accuracy], feed_dict={x: batch_x,
                                                              y: batch_y})

            summary_writer_train.add_summary(
                (sess.run(train_summaries, feed_dict={x: batch_x,
                                                      y: batch_y})), batch)

            if counter % disp_counter == 0:

                print("Iter " + str(i) + ", Loss= " + "{:.6f}".format(loss) + ", Training Accuracy= " + "{:.5f}".format(acc)
                      )
                print("Optimization Finished!")
```

در این قسمت به که تست مربوط است، هر عکس به صورت مجزا داده شده و خطای آن ذخیره می‌شود و برای تصاویری که اشتباه دسته‌بندی می‌شوند، اندیس آن‌ها را در لیست mis_idx ذخیره می‌کنیم تا بعداً همان‌طور که خواسته شده چند نمونه از آن‌ها را مشاهده کرده و خودمان دسته‌بندی کنیم. دقت مدل بر روی داده‌های تست نیز با دادن کل داده‌های اعتبارسنجی به شبکه محاسبه و چاپ می‌شود.

```
        # ######################################## Testing ######################################### #

        counter = 0

        while counter <= len(x_validation):
            counter += 1

            x_batch, y_batch = next_batch(1, x_validation, y_validation)
            summary_writer_validation.add_summary((sess.run(validation_loss_summary, feed_dict={x: x_batch, y: y_batch})))

            prediction_result = sess.run(correct_prediction, feed_dict={x: x_batch, y: y_batch})
            if prediction_result != 1:
                mis_classified_idx.append(counter)

        # Calculate accuracy for all test images
        x_batch, y_batch = x_validation, y_validation
        accuracy_test, loss_test = sess.run((accuracy, cost), feed_dict={x: x_batch, y: y_batch})
        print("The test accuracy is:" + str(accuracy_test))

        # summary_writer_conv_layer_1.add_summary(sess.run(filter_summary_conv_1, feed_dict={x: test_X,
        #                                                                                    y: test_y}), i)
        # summary_writer_conv_layer_2.add_summary(sess.run(filter_summary_conv_2, feed_dict={x: test_X,
        #                                                                                    y: test_y}), i)

    summary_writer_train.close()
    summary_writer_validation.close()
```

نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:



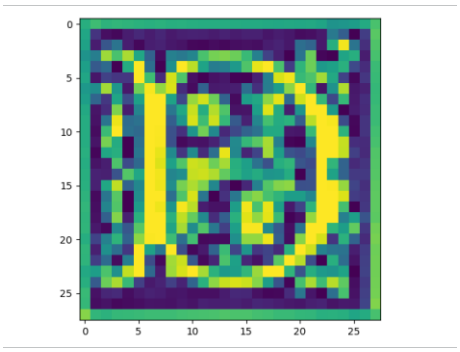


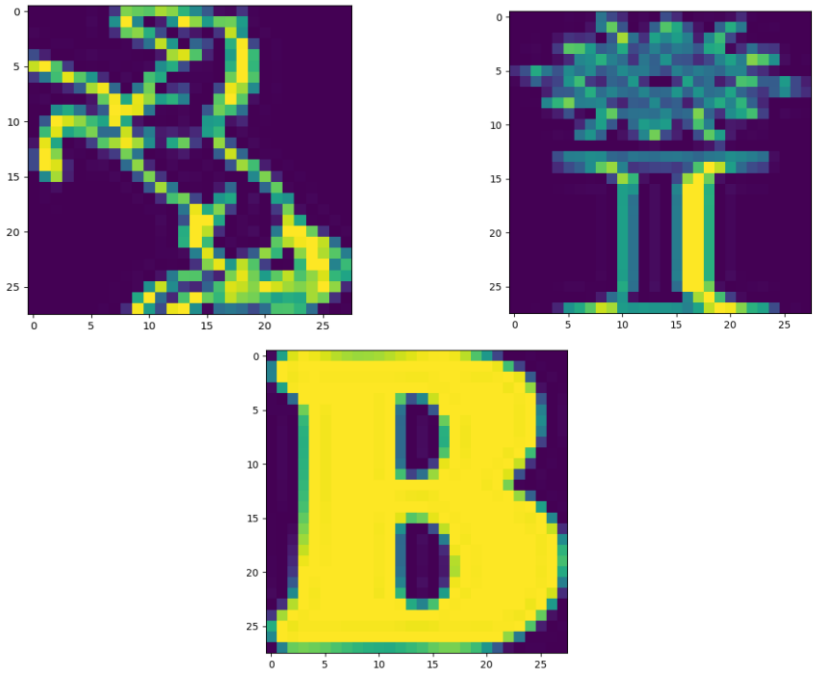
Epoch by epoch

نتیجه مدل (دقت و ماتریس درهم‌ریختگی):

```
 67%|██████▋   | 40/60 [01:26<00:53,  2.69s/it]
 68%|██████▊   | 41/60 [01:29<00:51,  2.72s/it]
 70%|███████   | 42/60 [01:32<00:49,  2.75s/it]
 72%|███████▏  | 43/60 [01:34<00:46,  2.76s/it]
 73%|███████▎  | 44/60 [01:37<00:43,  2.75s/it]
 75%|███████▌  | 45/60 [01:40<00:42,  2.84s/it]
 77%|███████▋  | 46/60 [01:43<00:41,  2.99s/it]
 78%|███████▊  | 47/60 [01:47<00:41,  3.22s/it]
 80%|████████  | 48/60 [01:51<00:39,  3.27s/it]
 82%|████████▏ | 49/60 [01:53<00:34,  3.13s/it]
 83%|████████▎ | 50/60 [01:56<00:30,  3.05s/it]
 85%|████████▌ | 51/60 [01:59<00:26,  2.99s/it]
 87%|████████▋ | 52/60 [02:02<00:23,  2.95s/it]
 88%|████████▊ | 53/60 [02:05<00:20,  2.93s/it]
 90%|█████████ | 54/60 [02:08<00:18,  3.04s/it]
 92%|█████████▏| 55/60 [02:11<00:15,  3.03s/it]
 93%|█████████▎| 56/60 [02:14<00:12,  3.04s/it]
 95%|█████████▌| 57/60 [02:17<00:08,  2.98s/it]
 97%|█████████▋| 58/60 [02:21<00:06,  3.11s/it]
 98%|█████████▊| 59/60 [02:24<00:03,  3.11s/it]
100%|██████████| 60/60 [02:27<00:00,  3.04s/it]
[[700  11   5   7   3   3  21   9   9   6]
 [ 10 674   2  24   4  13  20  13  17   5]
 [  6  21 712   9  13   9  31  12   9   5]
 [ 10  15   3 716   2   6  13  20  12  11]
 [ 15  17  20   8 669   7  25  13  29   6]
 [ 12   5   8  16   7 675  14   4  15  12]
 [  8   8  18  11   3  11 709   6  20  12]
 [ 21   7   4  10   8   5   7 732  19  12]
 [ 16   6   5  21  15  11  22  13 660  29]
 [ 12   8   3  15   1   9  10   8  17 720]]
The test accuracy is:0.870875
```

تعدادی از عکس‌هایی که اشتباه طبقه‌بندی شده‌اند.

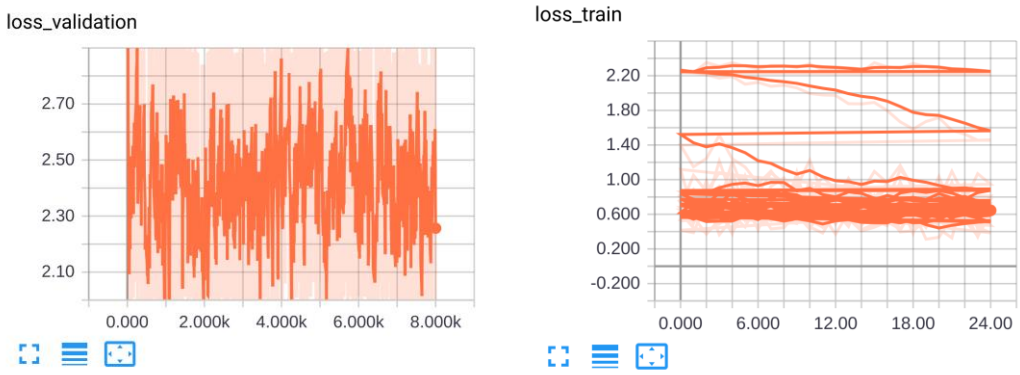




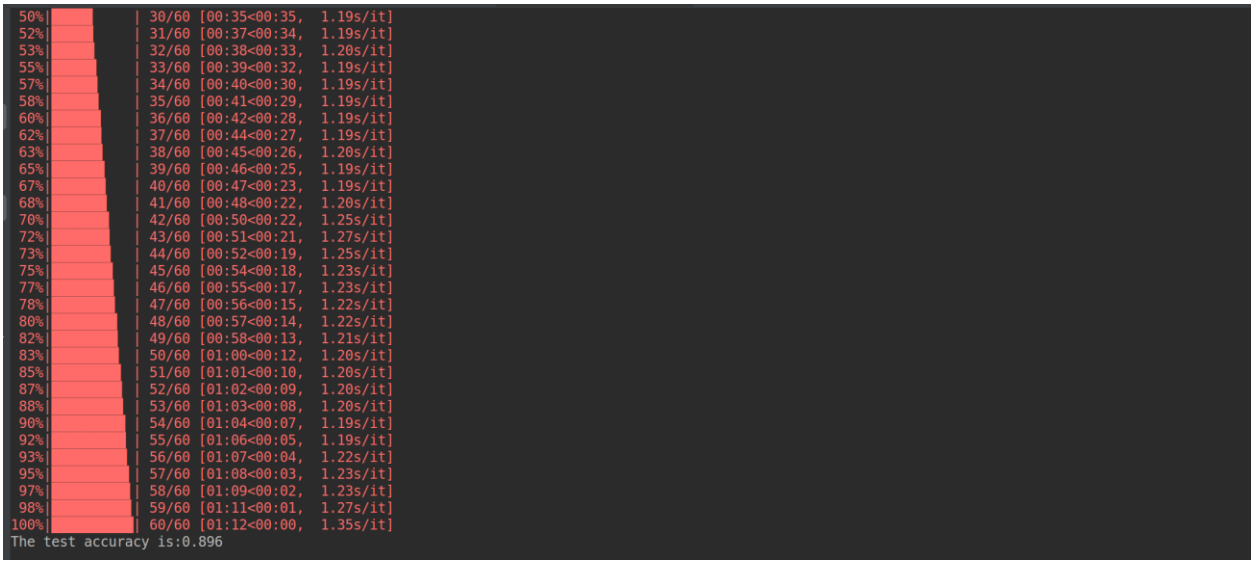
۲. در این قسمت من از تکنیک $batchnormalization$ و $dropout$ به صورت همزمان استفاده کردم و به عنوان نمونه فقط توضیح کد یک لایه را می‌آورم. ورودی با احتمال نگه‌داشته شدن نورون‌های ورودی $dropout$ می‌شود که این مقدار بالاتر مشخص شده‌است و برابر ۰/۸ است. پس از عملیات $pooling$، با داشتن میانگین و واریانس $batch$ و مقدار $scaling$ آن و انتقالش، عملیات $batchnormaliation$ انجام می‌شود. خروجی لایه $pool$ شده با احتمال $keep_{prob}$ که در $session$ داده می‌شود، $dropout$ می‌شود و این روند در انتهای هر لایه پس از $pooling$ انجام می‌شود.

```
    # Dropout
    x = tf.nn.dropout(x, keep_prob=keep_prob_input)

    # here we call the conv2d function we had defined above and pass the input image x, weights weight_conv_layer_1
    # and bias bias_conv_layer_1.
    conv1 = conv2d(x, weights['weight_conv_layer_1'], biases['bias_conv_layer_1'], strides=2)
    # Max Pooling (down-sampling), this chooses the max value from a 2*2 matrix window and outputs a 14*14 matrix.
    conv1_maxpool = maxpool2d(conv1, k=1)
    batch_mean_1, batch_var_1 = tf.nn.moments(conv1_maxpool, [0])
    beta_1 = tf.Variable(tf.zeros(conv1_maxpool.get_shape().as_list()[1:]))
    scale_1 = tf.Variable(tf.zeros(conv1_maxpool.get_shape().as_list()[1:]))
    conv1_maxpool = tf.nn.batch_normalization(conv1_maxpool, batch_mean_1, batch_var_1, offset=beta_1, scale=scale_1, variance_epsilon=epsilon)

    # Dropout
    conv1_maxpool = tf.nn.dropout(conv1_maxpool, keep_prob=keep_prob)
```

نمودار هزینه برای داده‌های آموزش و اعتبارسنجی:



نتیجه مدل (دقت و ماتریس درهم‌ریختگی):

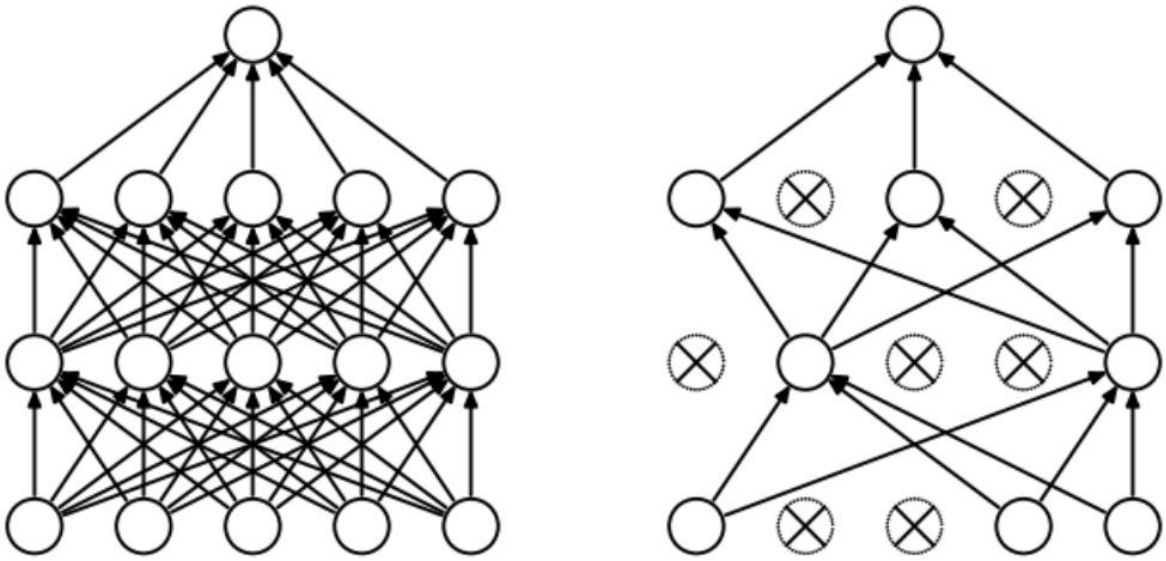


ماتریس درهم‌سازی در ابتدا رسم نشد و در یک اجرای دیگر با Colab بدست آمد:

* نمی‌دانم گفتن این موضوع فایده دارد یا نه! من چند روز قبل از ۳۰ دی کل تمرین را انجام دادم و دقتم از ۹۰ درصد بالاتر نرفت. تنها چیزی که مانده بود همین قسمت بود. همه کارهایی که به ذهنم می‌رسید را برای مدت یک روز کامل انجام دادم. فایل نوت‌بوک پیوست شده آخرین تغییرات را نشان می‌دهد ولی باز هم نتیجه مطلوبی حاصل نشد. من سایز ابعاد فیلترها را از ۲ در ۲ تا ۹ در ۹ تغییر دادم، سایز batch را از ۳۲ تا ۵۱۲ تغییر دادم و بهترین نتیجه به ازای همان ۶۴ بود. ۲ لایه کانولوشنی بدون pooling و ۲ لایه fully connected دیگر اضافه کردم و batch normalization و dropout در تمام لایه‌ها وجود دارد ولی باز هم دقت به ۹۲ درصد نرسید. لطفاً در نمره‌دهی با نرمی برخورد کنید☺.

```
Optimization Finished!
 60%|██████    | 302/500 [00:24<00:15, 12.47it/s]Iter 9, Loss= 0.440883, Training Accuracy= 0.82812
Optimization Finished!
 64%|██████    | 322/500 [00:25<00:14, 12.30it/s]Iter 9, Loss= 0.387967, Training Accuracy= 0.87500
Optimization Finished!
 68%|██████    | 342/500 [00:27<00:12, 12.39it/s]Iter 9, Loss= 0.676533, Training Accuracy= 0.81250
Optimization Finished!
 72%|██████    | 362/500 [00:29<00:11, 12.44it/s]Iter 9, Loss= 0.734839, Training Accuracy= 0.79688
Optimization Finished!
 76%|███████   | 382/500 [00:30<00:09, 12.39it/s]Iter 9, Loss= 0.556269, Training Accuracy= 0.81250
Optimization Finished!
 80%|███████   | 402/500 [00:32<00:07, 12.46it/s]Iter 9, Loss= 0.509597, Training Accuracy= 0.82812
Optimization Finished!
 84%|███████   | 422/500 [00:33<00:06, 12.44it/s]Iter 9, Loss= 0.386842, Training Accuracy= 0.85938
Optimization Finished!
 88%|████████  | 442/500 [00:35<00:04, 12.56it/s]Iter 9, Loss= 0.507009, Training Accuracy= 0.89062
Optimization Finished!
 92%|████████  | 462/500 [00:37<00:03, 12.49it/s]Iter 9, Loss= 0.199573, Training Accuracy= 0.95312
Optimization Finished!
 96%|████████  | 482/500 [00:38<00:01, 12.36it/s]Iter 9, Loss= 0.630633, Training Accuracy= 0.78125
Optimization Finished!
100%|████████| 500/500 [00:40<00:00, 12.39it/s]
Iter 9, Loss= 0.431069, Training Accuracy= 0.85938
Optimization Finished!
[[757  14   9  14  14   9   7  26  16   9]
 [ 15 719   4  29  18  10  25  10  16  16]
 [  2   1 689   2  17   3  22   3   2   2]
 [  8  13   3 725   4   5   6  15  12   8]
 [  1  12  21   1 745   5   8   5  11   1]
 [  8   3   3   4  14 706   7   4   8   2]
 [  5  15  31   8  15  11 685   2  10   5]
 [  9   4   4   5   5   3   3 717   5   2]
 [ 16  10   8  17  21  14  14  19 661  15]
 [  8   7   2  18   4  16  14   3  49 692]]
The test accuracy is:0.887
```

توضیح روش منظم‌سازی $dropout$: ایده این روش با توجه به مقاله اصلی آن این است که برای جلوگیری از $overfit$ شدن مدل، به نوعی نورون‌ها را برای تولید نتیجه به خود و تعدادی از اطرفیان متکی می‌کند! در واقع در هر بار آموزش خروجی تعدادی از نورون‌های ورودی و لایه‌های میانی در صفر ضرب می‌شود و به این ترتیب آن‌ها در آموزش مدل تأثیر نخواهند داشت و یک مدل $sparse$ خواهیم داشت. هر گام آموزش برای تولید نتیجه از یکی از تعداد زیادی شبکه هرس‌شده استفاده می‌کند و به این ترتیب نورون‌ها مجبور می‌شوند نتیجه را بدون کمک نورون‌های همسایه تولید کنند و این باعث می‌شود داده‌های آموزش حفظ نشوند و فقط ویژگی‌های واقعی متمایز کننده در مدل نگه داشته شود. در زمان تست همه نورون‌ها حضور دارند ولی خروجی آن‌ها در احتمال حضور آن‌ها هنگام آموزش ضرب می‌شود. به نوعی میانگین وزن‌دار مدل‌های آموزش دیده شده در خروجی اثر می‌گذارد. برای آموزش و تست ساختار شماتیک زیر روش $dropout$ را بهتر به نمایش می‌گذارد.



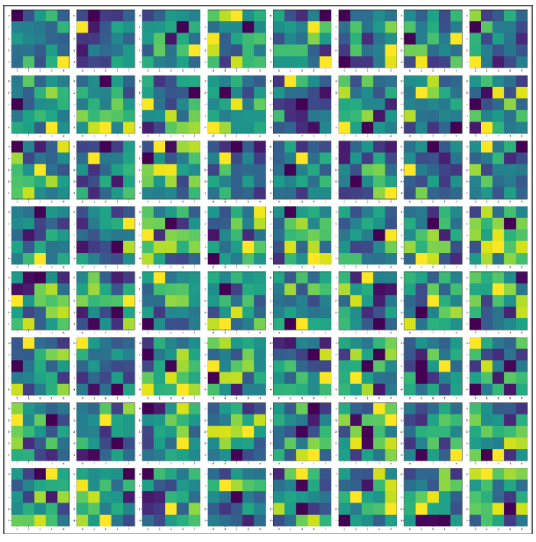
توضیح روش منظم‌سازی $batchnormalization$: یک مشکل در آموزش شبکه‌های عمیق که به $Internal\ Covariate\ shift$ شناخته می‌شود این است که توزیع داده‌ها در لایه‌های بعدی شدیداً به

پارامترهای لایه‌های قبلی وابسته است و این باعث می‌شود تغییرات کوچک اثرات شدیدی در توزیع داده‌ها در لایه‌های عمیق‌تر ایجاد کند و در واقع شبکه باید دائماً به این توزیع متغیر $adapt$ شود که فرآیند آموزش را سخت می‌کند. برای جبران این مشکل در ابتدای هر لایه توزیع ورودی نرمال می‌شود تا با توزیع‌های بسیار متفاوت در لایه‌های متوالی روبرو نشویم.

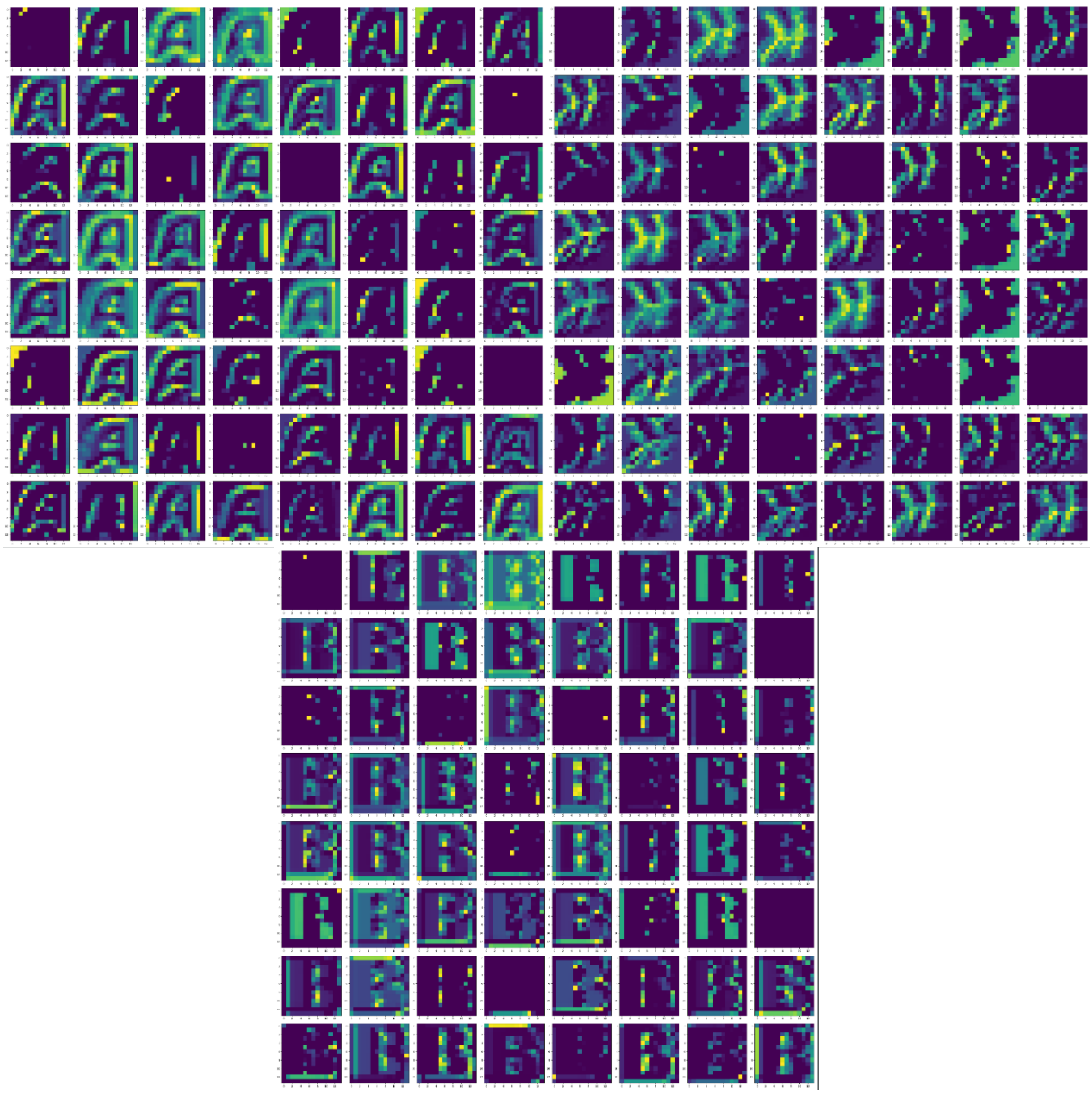
۳. برای مشاهده موارد ذکر شده قطعه کد زیر به کد اصلی قسمت قبل اضافه می‌شود. در این کد وزن‌های لایه اول و خروجی این لایه برای ۳ عکس نشان داده می‌شود. (در زیر کد یک عکس دلخواه آورده شده‌است.) چون این لایه ۶۴ کرنل دارد، در یک subplot همه وزن‌ها را نشان می‌دهیم. خروجی این لایه نیز در یک تصویر ۸ در ۸ برای ۳ عکس از عکس‌های تست نشان داده شده‌است. در ادامه به ارائه آن‌ها و توضیحشان می‌پردازیم.

```
# Plotting first layer's weights

weights_image_conv_1 = np.reshape(sess.run(weights['weight_conv_layer_1']).T, newshape=(64, 5, 5))

fig_1 = plt.figure()
plt.figure(num=None, figsize=(30, 30), dpi=100)
plt.title('First Convolution Layer Weights')

for j in np.arange(0, 64):
    plt.subplot(8, 8, j + 1)
    plt.imshow((weights_image_conv_1[j][:][:]).T)

plt.show()

# Plotting first cnn layer's output for arbitrary images

feed_x = test_X[15]
feed_x = feed_x.reshape(-1, 28, 28, 1)
first_layer_output = np.reshape(sess.run(conv1, feed_dict={x: feed_x}).T, newshape=(64, 14, 14))

fig_2 = plt.figure()
plt.figure(num=None, figsize=(30, 30), dpi=100)
plt.title('First Convolution Layer Outputs')

for j in np.arange(0, 64):
    plt.subplot(8, 8, j + 1)
    plt.imshow((first_layer_output[j][:][:]).T)

plt.show()
```

وزن‌های لایه کانولوشن اول:



خروجی ۳ تصویر ورودی پس از گذر کردن از لایه اول:



معمولاً لایه‌های اولیه شبکه‌های عمیق کانولوشنی مانند لایه‌های اولیه سیستم بینایی (که از آن نیز ایده گرفته‌اند) ویژگی‌های سطح پایین تصویر را استخراج می‌کنند و تفسیرهای عمیق‌تر آن‌ها به عهده لایه‌های بعدی شبکه است. از این ویژگی‌های سطح پایین می‌توان به تشخیص لبه، میانگین‌گیری و $smoothing$ و مشتق‌گیری در جهت‌های مختلف اشاره کرد. در شکل وزن‌های لایه کانولوشن که آورده شده است نیز این نوع فیلترها مشاهده می‌شوند. اگر یک نوار در تصویر روشن و نوار کناری آن تاریک‌تر باشد به معنی مشتق درجهت عمود بر جهت نوارهاست و این الگو در تعدادی از تصویر صفحه قبل مشاهده می‌شود. هم‌چنین در تعدادی فیلترها مرکز عکس وزن بیشتری در خروجی داشته و نوعی $smoothing$ به شمار می‌رود.

در ۳ تصویر فوق اثر اعمال این کرنل‌ها بر تصویر ورودی نشان داده شده و ویژگی‌های اولیه‌ای که به آن‌ها اشاره شد کاملاً در این تصاویر مشهودند. به عنوان مثال در حرف A و B، در تعدادی تصویر لبه‌های افقی روشن‌اند و در بعضی دیگر فقط لبه‌های عمودی روشن‌اند که نشان‌دهنده مشتق‌گیری در این جهت‌هاست. در بعضی دیگر نیز کل تصویر تاریک یا روشن و تار شده که نتیجه اعمال کرنل‌های $smooth$ کننده هستند.