Problem 4 (1)

October 22, 2019

1 Problem 4: Batch Normalization, Dropout, MNIST

1.1 4.1: Co-adaptation and Co-Variaate Shift

Co-adaptation: In a dense neural network, neurons must learn to identify and map different features either alone or in a small group. This will ensure that each neuron is learning something different about the feature space, and thus will result in superior performance. However, in a densely connected network, while trainining,it may happen that a cetrain set of neurons get activated every time, since a particular feature (or a slight variation) is present in all the input vectors. This will lead to a form of clustering- the neurons work as a group to predict the same feature. This can be termed as co-adaptation.

Co-Variate Shift: The inherent change in distribution of network activations between different layers is called Internal Covariance Shift. As the input progresses down the layers, based on the behaviour of the activation function and weights in the precious layers, the distribution changes drastically and may result in the activation of the lower layers getting saturated in non-linear spaces in higher dimensions. This inturn increases the time it takes for the network to converge. Reducing Co-Variance shift, in esscence speeds up the convergence.

1.1.1 The following sections contain the code for each of the sub-problems. The final section consolidates all the metrics, and has a table containing The Loss On Test Set, and Accuracy of the different models

[1]:

```
from keras.optimizers import RMSprop,Adagrad,Adadelta,Adam,Nadam
from keras.models import Sequential
from keras.layers import Conv2D,AveragePooling2D,Dense,Dropout,Activation,BatchNormalization
import keras.layers as layers
from keras import regularizers

from keras.datasets import mnist
from keras.utils import np_utils
```

```
Using TensorFlow backend.
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
```

```
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:517: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:518: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorflow/python/framework/dtypes.py:525: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning:
```

```
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.5/dist-
packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning:
Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

[2]:

```
# Data prep
# Load dataset as train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Set numeric type to float32 from uint8
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")

# Transform lables to one-hot encoding
y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)

# Reshape the dataset into 4D array
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)

metrics = []
```

1.2 4.2 With BatchNorm for hidden layers, and standard normalization for input layer

[4]:

```
model = Sequential()

# C1 Convolutional Layer
model.add(Conv2D(6, kernel_size=(5,5), strides=(1,1), use_bias=False,␣
 ↪input_shape=(28,28,1), padding="same"))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(1, 1), padding='valid'))

# C3 Convolutional Layer
model.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), use_bias=False,␣
 ↪padding='valid'))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S4 Pooling Layer
```

```
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

# C5 Fully Connected Convolutional Layer
model.add(Conv2D(120, kernel_size=(5, 5), strides=(1, 1), padding='valid'))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Flatten the CNN output so that we can connect it with fully connected layers
model.add(layers.Flatten())

# FC6 Fully Connected Layer
model.add(Dense(84))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Output Layer with softmax activation
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='SGD',␣
 ↪metrics=['accuracy'])
```

[5]:

```
#standard normalize input
x_train /= 255
x_test /= 255

history = model.fit(x_train,y_train,epochs=10,batch_size=256)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.5/dist-
packages/keras/backend/tensorflow_backend.py:422: The name tf.global_variables
is deprecated. Please use tf.compat.v1.global_variables instead.

Epoch 1/10
60000/60000 [==============================] - 58s 970us/step - loss: 0.3338 -
accuracy: 0.9182
Epoch 2/10
60000/60000 [==============================] - 75s 1ms/step - loss: 0.1483 -
accuracy: 0.9659
Epoch 3/10
60000/60000 [==============================] - 67s 1ms/step - loss: 0.1099 -
accuracy: 0.9752
Epoch 4/10
60000/60000 [==============================] - 73s 1ms/step - loss: 0.0907 -
accuracy: 0.9793
Epoch 5/10
60000/60000 [==============================] - 72s 1ms/step - loss: 0.0788 -
accuracy: 0.9822
Epoch 6/10
```

```
60000/60000 [==============================] - 65s 1ms/step - loss: 0.0704 -
accuracy: 0.9841
Epoch 7/10
60000/60000 [==============================] - 70s 1ms/step - loss: 0.0637 -
accuracy: 0.9859
Epoch 8/10
60000/60000 [==============================] - 63s 1ms/step - loss: 0.0581 -
accuracy: 0.9870
Epoch 9/10
60000/60000 [==============================] - 58s 961us/step - loss: 0.0540 -
accuracy: 0.9878
Epoch 10/10
60000/60000 [==============================] - 66s 1ms/step - loss: 0.0502 -
accuracy: 0.9887
```

```
      ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-5-f2c79fc02fc5> in <module>
          4
          5 history = model.fit(x_train,y_train,epochs=10,batch_size=256)
    ----> 6 histories.append(history)


        NameError: name 'histories' is not defined
```

1.2.1 Batch Normalization layer Parameters:

[6]:

```
print("Learned batch parameters: \nLayer 1 (After first convolutional layer) :␣
 ↪",model.layers[1].get_weights(),
      "\nLayer 2 (After second convolutional layer): ",model.layers[5].
 ↪get_weights(),
      "\nLayer 3 (After first dense layer): ",model.layers[9].get_weights(),
      "\nLayer 4 (After second dense layer): ",model.layers[13].get_weights())
```

```
Learned batch parameters:
Layer 1 (After first convolutional layer) :  [array([1.0227486 , 0.9933009 ,
0.95239097, 1.0617564 , 1.0096271 ,
       1.0478406 ], dtype=float32), array([-0.0511883 , -0.00876471,
-0.00128828, -0.08093079, -0.05750775,
       -0.04072163], dtype=float32), array([-0.09701837, -0.04794088, -0.0585382
, -0.16585663, -0.07888251,
       -0.16325763], dtype=float32), array([0.08274772, 0.0272827 , 0.01749997,
0.09768781, 0.03728754,
       0.08225607], dtype=float32)]
Layer 2 (After second convolutional layer):  [array([1.0038083 , 1.0049825 ,
1.0063536 , 0.9980975 , 0.99966276,
       1.0016065 , 1.0001917 , 1.0058019 , 1.0002332 , 1.0133479 ,
       0.999777  , 0.9991526 , 0.9976403 , 1.0245603 , 1.0088576 ,
       0.99244666], dtype=float32), array([-0.01195861,  0.01223815, -0.0022188
,  0.00485479,  0.01419998,
       -0.02188285,  0.00901312,  0.00899166,  0.00261207, -0.01462903,
        0.01454773, -0.01018127,  0.00911689,  0.03315625, -0.01661647,
       -0.00487536], dtype=float32), array([-0.00276375,  0.00479202,  0.0109833
, -0.01949189,  0.03752831,
       -0.02599387, -0.00458565,  0.00746924,  0.00125884, -0.02023948,
        0.01687014, -0.00269349,  0.03349387,  0.0134683 , -0.02399436,
       -0.0161607 ], dtype=float32), array([0.14606214, 0.19912885, 0.1652699 ,
0.20899837, 0.33509785,
       0.65799546, 0.07136545, 0.15779907, 0.17674299, 0.1122109 ,
       0.09138543, 0.14180051, 0.42296848, 0.07588281, 0.07959792,
       0.14786203], dtype=float32)]
Layer 3 (After first dense layer):  [array([1.0005733 , 1.0014514 , 1.0035948 ,
1.0021538 , 1.0055928 ,
       1.0016832 , 1.0031233 , 1.0018834 , 1.0054564 , 1.0059551 ,
       1.0034274 , 1.00065   , 1.004923  , 1.0009328 , 1.0016038 ,
       1.000361  , 1.0021045 , 0.99956405, 1.0047302 , 0.99974185,
       1.0041264 , 0.9989059 , 1.0000215 , 0.99995303, 1.005834  ,
       1.0005907 , 1.0017    , 1.0009811 , 1.0027105 , 1.0013143 ,
       1.0052094 , 1.0037159 , 1.0013502 , 1.0016869 , 0.99946666,
       1.0028617 , 0.9994873 , 0.9975435 , 1.0003728 , 1.0003443 ,
       1.0018057 , 0.9979511 , 0.99769664, 0.99897707, 1.0064453 ,
       1.0034422 , 1.0008949 , 1.0024923 , 1.0026659 , 1.002865  ,
       0.99834085, 1.0005854 , 1.0002403 , 1.0014703 , 1.0023531 ,
       1.0043554 , 0.9995161 , 1.0009936 , 1.0090805 , 0.99881566,
       1.0009359 , 1.0008423 , 1.002945  , 1.0045079 , 1.0042466 ,
       1.0023439 , 1.0005193 , 1.0014554 , 1.0025408 , 0.9998623 ,
       1.0014228 , 1.0039268 , 1.0030086 , 1.0015423 , 1.0025098 ,
       1.0004336 , 1.0016387 , 1.0064819 , 1.0023078 , 1.0023501 ,
       1.00147   , 1.0026412 , 0.99980336, 1.0048072 , 0.99756664,
       0.9977275 , 1.0017371 , 1.001306  , 1.0022146 , 0.997058  ,
       1.0010754 , 1.0018501 , 1.0031835 , 0.99905604, 1.0003238 ,
       0.99859524, 1.0007555 , 1.0014371 , 1.0004492 , 1.0005474 ,
       1.002633  , 1.0010931 , 1.0035088 , 1.0009891 , 1.0000696 ,
       1.0010895 , 1.0015525 , 1.000105  , 0.99980026, 1.0001111 ,
       1.0007921 , 1.0008959 , 0.9992291 , 0.99766546, 1.0018909 ,
       1.0033878 , 1.0032974 , 0.9998828 , 1.0012654 , 0.9990677 ],
      dtype=float32), array([-1.7015125e-03, -2.2923390e-03, -3.3415474e-03,
-1.7897750e-04,
       -6.6333167e-05,  2.3627535e-03,  5.7235576e-04, -1.1031628e-03,
       -2.4293126e-03, -5.7627079e-03, -1.8017006e-03, -1.2580529e-03,
        3.2891724e-03,  1.2746957e-03,  4.1956076e-04,  2.5497752e-03,
        8.1563980e-04, -9.0557348e-04,  2.9660896e-03,  8.0101314e-04,
       -3.6483957e-03,  2.1435313e-03, -2.6833164e-05,  2.2027465e-03,
       -6.1101788e-03,  1.4259717e-03, -4.0312801e-04, -7.0515074e-05,
        1.0991743e-03, -1.0572388e-03,  3.4511341e-03,  5.5862069e-03,
       -3.2832203e-03,  1.7576815e-03, -6.5203942e-04,  5.6189938e-06,
        1.0770315e-03,  2.4525481e-03, -4.7163907e-03, -9.5576956e-04,
       -4.4975206e-03, -3.1377669e-04, -3.0034578e-03,  4.8104592e-04,
        3.9206157e-03, -6.0246750e-03, -3.0687246e-03,  2.2074503e-03,
       -1.7870952e-03,  1.1700044e-04, -1.2266053e-03,  2.1800916e-03,
       -1.2343973e-03,  2.1098864e-03,  3.6139793e-03, -4.3526953e-03,
       -2.4649545e-03,  2.0454435e-03, -5.3749583e-04, -1.9947072e-03,
        1.8657198e-03, -2.8842401e-03,  2.2509329e-03, -4.1728034e-03,
        1.6012858e-03, -3.7000611e-04,  8.4295250e-05, -1.2313918e-03,
        8.7050878e-04,  1.7177262e-03, -1.0301872e-03,  9.8523963e-04,
        9.9251408e-04,  2.4021633e-03,  1.4741030e-03, -2.1904919e-03,
       -1.0732913e-07, -5.3884317e-03,  8.6966937e-04, -2.3649232e-03,
       -2.2141146e-03, -2.5322274e-03, -3.1704465e-03, -3.4450172e-03,
       -2.1504650e-03, -4.0305839e-03,  1.2644174e-04, -7.3498275e-05,
        3.8503204e-03, -5.9170416e-04, -2.9570500e-03,  3.2434477e-03,
       -3.9699492e-03, -1.0976405e-03, -2.0928462e-03,  1.0877523e-03,
        5.9120171e-03, -6.8283908e-04,  1.4311953e-04,  1.5850946e-03,
        7.2913076e-04,  8.7262085e-04,  1.5254642e-03, -5.8338820e-04,
        1.5656067e-03,  3.0237870e-04,  1.0398632e-03,  4.1082310e-03,
        2.4524098e-04, -1.0017380e-03, -2.7493280e-03,  1.4733501e-04,
       -1.8920270e-03, -1.0378627e-03, -1.2232408e-04,  2.8081550e-03,
       -2.5485291e-03, -3.6738423e-04,  1.8417303e-03,  1.7519383e-03],
      dtype=float32), array([ 0.14525461,  0.07013167, -0.02730178, -0.04544163,
0.00190735,
        0.04150978,  0.05542617,  0.13786624,  0.02362221,  0.02651046,
       -0.00910547,  0.00641203,  0.14519177,  0.06172212,  0.03665346,
        0.08367157, -0.04211572,  0.06069427,  0.08213181, -0.03546967,
       -0.10804989, -0.11769501, -0.06876633, -0.11495119,  0.07122132,
       -0.00518842, -0.1843107 , -0.02781058,  0.25886047, -0.02980049,
       -0.00360388, -0.06596375, -0.02559104,  0.02621117,  0.20494056,
       -0.02004141, -0.02071655,  0.18353741, -0.07438858, -0.08976404,
       -0.03967005,  0.0123605 , -0.15949173, -0.06691488, -0.05139984,
       -0.05055316, -0.0168283 ,  0.0560364 , -0.11454628,  0.05901556,
       -0.22737283, -0.09542293,  0.06968255, -0.0532479 , -0.05780787,
        0.01660315,  0.07548505,  0.11276964, -0.01300354, -0.03688841,
       -0.07673144, -0.10000931, -0.05049716,  0.03253292, -0.08787797,
        0.07653227, -0.014417  , -0.04437534, -0.12100431,  0.01410634,
       -0.15499018, -0.0346689 , -0.14609347,  0.14136371,  0.02875607,
        0.13166891,  0.004981  ,  0.05321118,  0.12966408,  0.10544759,
       -0.00399393,  0.00432087,  0.02691093, -0.1033589 ,  0.05814778,
        0.16063082,  0.05566478, -0.02516657, -0.08277831, -0.08720028,
       -0.10223821, -0.12197803, -0.02562243, -0.07567093, -0.21900228,
       -0.03318826,  0.07937697,  0.12584509,  0.01752287,  0.06904717,
        0.0214392 ,  0.00957611,  0.00202798,  0.36408037,  0.05966674,
       -0.14174996, -0.11824527, -0.13307647,  0.0297594 , -0.04752627,
        0.03480808, -0.08412981,  0.21985036,  0.20309483,  0.13524929,
       -0.07592916, -0.00849732, -0.01707737, -0.02586056,  0.00097429],
      dtype=float32), array([0.12150812, 0.16748595, 0.06112478, 0.05484471,
0.04188222,
       0.04743984, 0.07583353, 0.13092105, 0.04463758, 0.09820483,
       0.05022379, 0.10448895, 0.08414842, 0.08936278, 0.1087015 ,
       0.16255215, 0.05323662, 0.06226823, 0.04635723, 0.08570489,
       0.04068493, 0.23550701, 0.08615264, 0.04493684, 0.02765558,
       0.08001579, 0.12083919, 0.08455579, 0.1563778 , 0.08643075,
       0.11242815, 0.05761655, 0.06121784, 0.05179197, 0.12288208,
       0.10953879, 0.18530947, 0.14209145, 0.0926223 , 0.08717394,
       0.04601697, 0.05470276, 0.09853765, 0.14044221, 0.0563033 ,
       0.06110966, 0.16743746, 0.17499527, 0.09564474, 0.12244314,
       0.14815012, 0.05511298, 0.08251189, 0.12148449, 0.10822891,
       0.07444322, 0.10048138, 0.1715871 , 0.08736493, 0.06634975,
       0.05943802, 0.14600709, 0.11061359, 0.04829621, 0.06098269,
       0.08440673, 0.07229514, 0.04637694, 0.08772618, 0.17713976,
       0.10814586, 0.0536167 , 0.18646835, 0.09723603, 0.09607005,
       0.05263199, 0.07514501, 0.06381091, 0.09577116, 0.09039526,
       0.05306574, 0.12098725, 0.15470181, 0.05990868, 0.14458969,
       0.11299943, 0.10311396, 0.10919401, 0.08945891, 0.05863415,
       0.11238606, 0.08277764, 0.08113095, 0.10072866, 0.16480048,
       0.0878296 , 0.123778  , 0.1132776 , 0.04762676, 0.10121648,
       0.05327417, 0.11354055, 0.05309352, 0.25703335, 0.08720883,
       0.09692472, 0.06932184, 0.0841019 , 0.0645803 , 0.22087039,
       0.06932102, 0.1229241 , 0.1638188 , 0.18920986, 0.08486985,
       0.06442834, 0.06600814, 0.08209579, 0.09553234, 0.06927423],
      dtype=float32)]
Layer 4 (After second dense layer):  [array([1.015709 , 1.0203916, 1.0162107,
1.0208316, 1.0232376, 1.02299  ,
       1.010647 , 1.0124627, 1.02396  , 1.0159271, 1.0200316, 1.0251206,
       1.0279607, 1.0200933, 1.0199417, 1.0182372, 1.0242162, 1.0238992,
       1.0195678, 1.0212224, 1.0273153, 1.0081038, 1.0200411, 1.013433 ,
       1.0106845, 1.0171854, 1.0137854, 1.0057262, 1.0124403, 1.0172858,
       1.0301833, 1.0125828, 1.0143591, 1.0172498, 1.019988 , 1.0139835,
       1.0265086, 1.021325 , 1.0186955, 1.0268428, 1.0184878, 1.0232449,
       1.0247233, 1.030357 , 1.021585 , 1.020346 , 1.0144656, 1.0200461,
       1.0223137, 1.0185448, 1.0169013, 1.0244031, 1.0246983, 1.0177377,
       1.016408 , 1.0212418, 1.012529 , 1.0161399, 1.021564 , 1.016091 ,
       1.0150918, 1.0107527, 1.0207438, 1.0275384, 1.0217617, 1.0210139,
       1.0173041, 1.022399 , 1.0183634, 1.0136116, 1.0232247, 1.0250285,
       1.0151812, 1.0129783, 1.0188608, 1.0095383, 1.0113499, 1.0187374,
       1.0136864, 1.0187438, 1.0123662, 1.0158604, 1.0229259, 1.0123378],
      dtype=float32), array([ 3.9086845e-03,  2.5710997e-03,  5.7354583e-03,
5.1022368e-03,
        2.6665160e-03, -1.0022460e-02, -2.3386916e-03, -1.6426131e-03,
       -7.4315756e-03, -2.6631244e-03, -7.1310357e-04,  8.4447283e-03,
       -5.2924193e-03, -3.7179934e-03, -6.9218394e-03,  3.7126823e-03,
       -1.1466362e-02, -1.0095377e-02,  1.3944890e-03, -6.4515596e-05,
        9.1041561e-04,  4.4450082e-04,  9.6995989e-03, -2.6876074e-03,
       -2.1021001e-03, -6.2785223e-03,  4.5494842e-03,  2.5552625e-03,
       -3.9907061e-03,  3.4817224e-04,  1.0672494e-02,  5.9259403e-03,
        2.3059151e-03, -5.8106789e-03,  1.1028710e-03, -3.2144464e-03,
        3.5530236e-03,  4.8656529e-03,  7.1341679e-03,  2.0170573e-03,
        3.2441334e-03, -9.4329873e-03,  3.2806827e-03,  7.5852138e-04,
       -8.6192405e-03,  4.7119224e-04,  4.0141800e-03,  7.3123192e-03,
        1.2263604e-03, -3.6677346e-03,  7.6747532e-03,  2.6283704e-03,
       -5.2596563e-03,  9.1438899e-03,  7.5290736e-04,  2.1912642e-03,
       -1.3122517e-03,  2.7251124e-04,  1.4139673e-02, -7.8402469e-03,
       -9.3492651e-03, -4.4032726e-03, -3.2758382e-03, -3.1463532e-03,
       -3.5077983e-03, -5.3096837e-03,  7.1873302e-03,  1.3310919e-03,
        3.1510117e-03, -2.1779467e-03, -7.0777619e-03, -1.0372761e-02,
       -7.4911714e-03,  2.9235333e-03,  3.1753455e-03,  1.0760212e-03,
       -7.2330739e-03,  8.4539142e-04, -4.5021912e-03,  9.9154962e-03,
       -1.0023791e-02, -3.2740554e-03,  6.6957260e-03,  1.2569914e-02],
      dtype=float32), array([ 0.07518467, -0.23806761,  0.11993587,  0.31851584,
0.14662236,
       -0.3654036 , -0.06795713, -0.14011636,  0.30916268,  0.2592399 ,
       -0.7118044 ,  0.8887813 ,  0.46486712,  0.05374847,  0.03266971,
       -0.15905637,  0.07406325,  0.06484315,  0.03075639,  0.07513023,
       -0.14354105, -0.6631058 ,  0.05565241, -0.3742415 , -0.18825121,
       -0.47192264, -0.09478843, -0.06960767, -0.2854469 ,  0.2205986 ,
        0.19589275, -0.04675749, -0.3307558 ,  0.22638233,  0.5365144 ,
        0.19667663,  0.15562062,  0.22945598,  0.25150523,  0.07854117,
        0.14242823,  0.15566525, -0.35090858,  0.55911493, -0.2344317 ,
        0.21442248,  0.22150934,  0.33612645, -0.7630219 , -0.1022023 ,
        0.66010237, -0.44831467,  0.2859286 ,  0.18697807,  0.2773238 ,
       -0.48933837,  0.6281309 , -0.11731863,  0.17869318,  0.10476243,
       -0.6921828 , -0.02581748, -0.46877655,  0.2802579 , -0.09904595,
        0.08651955,  0.28119826, -0.28610712,  0.24288248, -0.30499718,
        0.04648856, -0.0864483 , -0.5317389 , -0.30847043, -0.5149043 ,
       -0.02730505, -0.31240174, -0.45665336,  0.06429321,  0.29724795,
       -0.26807064,  0.02772093, -0.16615611,  0.3163753 ], dtype=float32),
array([0.22477359, 1.0347339 , 0.7486249 , 0.40476063, 0.42696926,
       0.42591017, 0.612687  , 0.7021379 , 0.45002234, 0.35263273,
       0.38093355, 0.87399155, 0.5277404 , 0.42995656, 0.562011  ,
       0.2959268 , 0.5093457 , 0.511318  , 0.34488073, 0.3042865 ,
       0.3128121 , 0.41255078, 0.52376693, 0.5515215 , 0.46627423,
       0.6693281 , 0.5281365 , 0.95788   , 0.4758021 , 0.5478395 ,
       0.38592792, 0.575158  , 0.39694583, 0.46985617, 0.4626579 ,
       0.6317741 , 0.4686879 , 0.5727954 , 0.42916563, 0.20152645,
       0.36457056, 0.49399778, 0.66119134, 0.4123708 , 0.3563009 ,
       0.4447202 , 0.81850296, 0.6930745 , 0.6340535 , 0.46438023,
       0.65476143, 0.36419857, 0.34220582, 0.35531923, 0.4170095 ,
       0.40114412, 0.49306446, 0.47866508, 0.622773  , 0.37152532,
       0.9927862 , 0.75206023, 0.28231904, 0.65123004, 0.33853737,
       0.2686711 , 0.65686554, 0.30565846, 0.5855498 , 0.35017025,
       0.5662803 , 0.35115564, 1.2152779 , 0.46923798, 0.38477525,
       0.38572305, 0.61428064, 0.3489162 , 1.0426228 , 0.61489004,
       0.3622287 , 0.3956569 , 0.41328984, 0.30106717], dtype=float32)]
```

[7]:

```
# test set metrics:
metrics.append(model.evaluate(x_test,y_test))
```

```
10000/10000 [==============================] - 8s 787us/step
```

1.3 4.3 With Batch Norm for all layers

[8]:

```
model = Sequential()

# C1 Convolutional Layer
model.add(BatchNormalization(input_shape=(28,28,1)))
model.add(Conv2D(6, kernel_size=(5,5), strides=(1,1), use_bias=False,␣
 ↪padding="same"))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(1, 1), padding='valid'))

# C3 Convolutional Layer
model.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), use_bias=False,␣
 ↪padding='valid'))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S4 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

# C5 Fully Connected Convolutional Layer
model.add(Conv2D(120, kernel_size=(5, 5), strides=(1, 1), padding='valid'))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Flatten the CNN output so that we can connect it with fully connected layers
model.add(layers.Flatten())

# FC6 Fully Connected Layer
model.add(Dense(84))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Output Layer with softmax activation
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='SGD',␣
 ↪metrics=['accuracy'])
```

[9]:

```
model.fit(x_train,y_train,epochs=10,batch_size=256)
```

```
Epoch 1/10
60000/60000 [==============================] - 66s 1ms/step - loss: 0.3236 -
accuracy: 0.9198
Epoch 2/10
60000/60000 [==============================] - 75s 1ms/step - loss: 0.1522 -
accuracy: 0.9646
Epoch 3/10
60000/60000 [==============================] - 68s 1ms/step - loss: 0.1126 -
accuracy: 0.9743
Epoch 4/10
60000/60000 [==============================] - 69s 1ms/step - loss: 0.0919 -
accuracy: 0.9794
Epoch 5/10
60000/60000 [==============================] - 83s 1ms/step - loss: 0.0788 -
accuracy: 0.9823
Epoch 6/10
60000/60000 [==============================] - 70s 1ms/step - loss: 0.0693 -
accuracy: 0.9848
Epoch 7/10
60000/60000 [==============================] - 73s 1ms/step - loss: 0.0624 -
accuracy: 0.9863
Epoch 8/10
60000/60000 [==============================] - 69s 1ms/step - loss: 0.0569 -
accuracy: 0.9876
Epoch 9/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.0526 -
accuracy: 0.9884
Epoch 10/10
60000/60000 [==============================] - 58s 974us/step - loss: 0.0488 -
accuracy: 0.9892
```

```
       ␣
 ↪---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call␣
 ↪last)
```

```
        <ipython-input-9-20a51b161df8> in <module>
          1 model.fit(x_train,y_train,epochs=10,batch_size=256)
    ----> 2 metrics.appemd(model.evaluate(x_test,y_test))


        AttributeError: 'list' object has no attribute 'appemd'
```

[10]:

```
metrics.append(model.evaluate(x_test,y_test))
```

```
10000/10000 [==============================] - 8s 785us/step
```

1.4 4.4 Using dropout instead of Batch Norm

[11]:

```
model = Sequential()

# C1 Convolutional Layer
model.add(Dropout(0.2,input_shape=(28,28,1)))
model.add(Conv2D(6, kernel_size=(5,5), strides=(1,1), use_bias=False,␣
 ↪padding="same"))
model.add(Dropout(0.5))
model.add(Activation('tanh'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(1, 1), padding='valid'))

# C3 Convolutional Layer
model.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), use_bias=False,␣
 ↪padding='valid'))
model.add(Dropout(0.5))
model.add(Activation('tanh'))

# S4 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

# C5 Fully Connected Convolutional Layer
model.add(Conv2D(120, kernel_size=(5, 5), strides=(1, 1), padding='valid'))
model.add(Dropout(0.5))
model.add(Activation('tanh'))

#Flatten the CNN output so that we can connect it with fully connected layers
model.add(layers.Flatten())

# FC6 Fully Connected Layer
model.add(Dense(84))
model.add(Dropout(0.5))
model.add(Activation('tanh'))

#Output Layer with softmax activation
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='SGD',␣
 ↪metrics=['accuracy'])
```

[12]:
```
model.fit(x_train,y_train,epochs=10,batch_size=256)
metrics.append(model.evaluate(x_test,y_test))
```

```
Epoch 1/10
60000/60000 [==============================] - 79s 1ms/step - loss: 1.5579 -
accuracy: 0.5539
Epoch 2/10
60000/60000 [==============================] - 79s 1ms/step - loss: 0.8369 -
accuracy: 0.7811
Epoch 3/10
60000/60000 [==============================] - 80s 1ms/step - loss: 0.6750 -
accuracy: 0.8190
Epoch 4/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.5863 -
accuracy: 0.8419
Epoch 5/10
60000/60000 [==============================] - 80s 1ms/step - loss: 0.5317 -
accuracy: 0.8558
Epoch 6/10
60000/60000 [==============================] - 79s 1ms/step - loss: 0.4911 -
accuracy: 0.8661
Epoch 7/10
60000/60000 [==============================] - 80s 1ms/step - loss: 0.4585 -
accuracy: 0.8736
Epoch 8/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.4340 -
accuracy: 0.8811
Epoch 9/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.4086 -
accuracy: 0.8867
Epoch 10/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.3897 -
accuracy: 0.8908
10000/10000 [==============================] - 6s 591us/step
```

1.5 4.5 Batch Norm + Dropout

[13]:

```
model = Sequential()

# C1 Convolutional Layer
model.add(Dropout(0.2,input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Conv2D(6, kernel_size=(5,5), strides=(1,1), use_bias=False,␣
 ↪padding="same"))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S2 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(1, 1), padding='valid'))

# C3 Convolutional Layer
model.add(Conv2D(16, kernel_size=(5, 5), strides=(1, 1), use_bias=False,␣
 ↪padding='valid'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('tanh'))

# S4 Pooling Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'))

# C5 Fully Connected Convolutional Layer
model.add(Conv2D(120, kernel_size=(5, 5), strides=(1, 1), padding='valid'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Flatten the CNN output so that we can connect it with fully connected layers
model.add(layers.Flatten())

# FC6 Fully Connected Layer
model.add(Dense(84))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('tanh'))

#Output Layer with softmax activation
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='SGD',␣
 ↪metrics=['accuracy'])
```

[16]:

```
model.fit(x_train,y_train,epochs=10,batch_size=256)
metrics.append(model.evaluate(x_test,y_test))
```

```
Epoch 1/10
60000/60000 [==============================] - 106s 2ms/step - loss: 0.2106 -
accuracy: 0.9438
Epoch 2/10
60000/60000 [==============================] - 107s 2ms/step - loss: 0.2072 -
accuracy: 0.9437
Epoch 3/10
60000/60000 [==============================] - 103s 2ms/step - loss: 0.1992 -
accuracy: 0.9469
Epoch 4/10
60000/60000 [==============================] - 100s 2ms/step - loss: 0.1966 -
accuracy: 0.9464
Epoch 5/10
60000/60000 [==============================] - 107s 2ms/step - loss: 0.1927 -
accuracy: 0.9479
Epoch 6/10
60000/60000 [==============================] - 106s 2ms/step - loss: 0.1867 -
accuracy: 0.9491
Epoch 7/10
60000/60000 [==============================] - 104s 2ms/step - loss: 0.1844 -
accuracy: 0.9495
Epoch 8/10
60000/60000 [==============================] - 100s 2ms/step - loss: 0.1808 -
accuracy: 0.9512
Epoch 9/10
60000/60000 [==============================] - 103s 2ms/step - loss: 0.1770 -
accuracy: 0.9513
Epoch 10/10
60000/60000 [==============================] - 101s 2ms/step - loss: 0.1742 -
accuracy: 0.9525
10000/10000 [==============================] - 14s 1ms/step
```

[19]:

```
print(metrics)
```

```
[[0.04980983104780316, 0.987500011920929], [0.050857153448462485,
0.9872000217437744], [0.2754036287669092, 0.9246000051498413],
[0.24856007757689805, 0.9294000267982483], [0.18753284983639606,
0.9469000101089478]]
```

1.6 Comparison between Loss and Accuracy

Model	Loss on Test Set	Accuracy of Model on the Test Set
2- Batch Normalization for Hidden layers + Standard Normalization for Input payer	0.04980983104780316	0.987500011920929
3- Batch Normalization for all layers	0.050857153448462485	0.9872000217437744
4- Dropout only	0.248560077576898	0.9294000267982483
5- Dropout + Batch normalization on all layers	0.18753284983639606	0.9469000101089478

From the above, after 10 epochs with a default learning rate and a batch size of 256:

1) Standard Normalization on the input layer and Batch normalization on the rest of the hidden layers seems to give the best performance.

2) Dropout in general performs worse than using normalization

3) Using both dropout and batch normalization performs better than dropout alone, but is significantly worse than using batch normalization alone.