4_1_CNN_Introduction-execise

December 27, 2021

1 Credits

Originally created for a previous version of the 02456-deep-learning course material, but converted to PyTorch. See repos for credits.

2 Dependancies and supporting functions

Loading dependancies and supporting functions by running the code block below.

[ ]:
```
%matplotlib inline
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
```

3 Convolutional Neural networks 101

Convolution neural networks are one of the most succesfull types of neural networks for image recognition and an integral part of reigniting the interest in neural networks. They are able to extract structural relations in the data such as spatial in images or temporal in time series.

In this lab we'll experiment with inserting 2D-convolution layers in the fully connected neural networks introduced in lab 1 (`1_Feedforward`). We'll further experiment with stacking of convolution layers, max pooling and strided convolutions which are all important techniques in current convolution neural network architectures. Lastly we'll try to visualize the learned convolution filters and try to understand what kind of features they learn to recognize.

If you haven't watched Jason Yosinski and colleague awesome video on visualizing convolutional networks you definitely should do so now.

If you are unfamilar with the the convolution operation https://github.com/vdumoulin/conv_arithmetic have a nice visualization of different convolution variants. For a more indepth tutorial please see http://cs231n.github.io/convolutional-networks/ or http://neuralnetworksanddeeplearning.com/chap6.html.

3.1 Reminder: What are convolutional networks?

ConvNets are in may respects very similar to the dense feedforward networks we saw previously: * The network is still organized into layers * Each layer is parameterized by weights and biases

* Each layer has an element-wise non-linear transformation (activation function) * There are no cycles in the connections (more on this in later labs)

So what is the difference? The networks we saw previously are called *dense* because each unit receives input from all the units in the previous layer. This is not the case for ConvNets. In ConvNets each unit is only connected to a small subset of the input units. This is called the *receptive field* of the unit.

Let us look at a quick example. The input (green matrix) is `1x5x5` dimensional tensor - i.e. it has one `channel` (like a grayscale image) and the map is of size `5x5`. Let us define a `1x3x3` kernel (yellow submatrix). The kernel weights are indicated by red in the bottom right of each elment. The computation can be thought of as first an elementwise multiplication, and then summing the results. Here we use a stride of 1, as shown in this animation:

[GIF courtesy of Stanford] (Figure 1) After having convolved the image we perform an elementwise non-linear transformation on the *convolved features*. In this example the input is a 2D *feature map* with depth 1.

4 Assignment 1

4.0.1 Assignment 1.1: Manual calculations

Perform the following computation, and write the result below.

Input

0	0	0	1
0	0	1	2
0	0	2	3
0	1	2	3

Kernel

0	0	2
0	1	2
0	2	3

(Figure 2)

1. Manually convolve the input, and compute the convolved features. No padding and stride of 1.
2. Perform `2x2` max pooling on the convolved features. Stride of 2.

Answer:

4.0.2 Assignment 1.2: Output dimensionality

Given the following 3D tensor input `(channel, height, width)` , a given amount (`channels_out`) of filters `(channels_in, filter_height, filter_width)`, stride `(height, width)` and padding `(height, width)`, calculate the output dimensionality if it's valid.

1. input tensor with dimensionality (1, 28, 28) and 16 filters of size (1, 5, 5) with stride (1, 1) and padding (0, 0)

- **Answer:**

2. input tensor with dimensionality (2, 32, 32) and 24 filters of size (2, 3, 3) with stride (1, 1) and padding (0, 0)

- **Answer:**

3. input tensor with dimensionality (10, 32, 32) and 3 filters of size (10, 2, 2) with stride (2, 2) and padding (0, 0)

- **Answer:**

4. input tensor with dimensionality (11, 8, 16) and 7 filters of size (11, 3, 3) with stride (2, 2) and padding (1, 1)

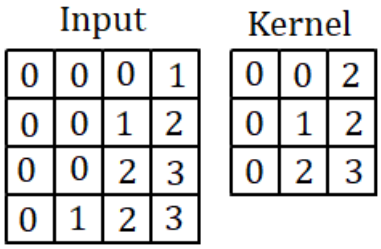
- **Answer:**

5. input tensor with dimensionality (128, 256, 256) and 112 filters of size (128, 3, 3) with stride (1, 1) and padding (1, 1)

- **Answer:**

[9]:
```
from IPython.display import Image
from IPython.display import display
display(Image('images/convolutions.gif',width=400),\
Image('./images/conv_exe.png',width=400))
```

```
<IPython.core.display.Image object>
```



5 Data: MNIST

The code below downloads and loads the same MNIST dataset as before. Note however that the data has a different shape this time, namely `(num_samples, num_channels, height, width)`.

[ ]:
```
## Download the MNIST dataset, if you haven't already

!if [ ! -f mnist.npz ]; then wget -N https://www.dropbox.com/s/qxywaq7nx19z72p/
 ↪mnist.npz; else echo "mnist.npz already downloaded"; fi
```

```
--2021-12-24 22:43:46--  https://www.dropbox.com/s/qxywaq7nx19z72p/mnist.npz
Resolving www.dropbox.com (www.dropbox.com)… 162.125.5.18,
2620:100:601d:18::a27d:512
Connecting to www.dropbox.com (www.dropbox.com)|162.125.5.18|:443… connected.
HTTP request sent, awaiting response… 301 Moved Permanently
Location: /s/raw/qxywaq7nx19z72p/mnist.npz [following]
--2021-12-24 22:43:46--  https://www.dropbox.com/s/raw/qxywaq7nx19z72p/mnist.npz
```

```
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response… 302 Found
Location: https://ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com/cd/0/in
line/Bcf_mV0Msej4iqjxA4TkWQI3OJhgWjZCAaGKEwidzRlb89zjprMEanU8oQ3afJikZlG-
gObTabaJuj8-7t5gpGgqgIad4n4Iy-5KllcRXmI4rX-BRjrEPDnt1faPSbe-
CO2UrXicCETnMnjvGMdJCZzv/file# [following]
--2021-12-24 22:43:46--  https://ucf8fbf780e495d7f9190e14d296.dl.dropboxusercont
ent.com/cd/0/inline/Bcf_mV0Msej4iqjxA4TkWQI3OJhgWjZCAaGKEwidzRlb89zjprMEanU8oQ3a
fJikZlG-gObTabaJuj8-7t5gpGgqgIad4n4Iy-5KllcRXmI4rX-BRjrEPDnt1faPSbe-
CO2UrXicCETnMnjvGMdJCZzv/file
Resolving ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com
(ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com)… 162.125.5.15,
2620:100:601d:15::a27d:50f
Connecting to ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com
(ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com)|162.125.5.15|:443…
connected.
HTTP request sent, awaiting response… 302 Found
Location: /cd/0/inline2/BcdiajxXXeIv-OWJFq-rvDEcL8FFHVWhJag6y7fSGKB-HEgUGyFjh9bB
jejSZ0GLJv50-H5MktiDklYBgBJx9gY5BL9h-tfhhvYmSg51TslIiT_QpjANQk7ZONH8QTEjaaUJKpSl
Y-4Wjfp7f-DxKBwzpRc7e9sH8ycoR86SalFvmPy3xmWaPflP7QM78sggEUzBfXPpOibAxv9ui-QEBrf-
hQTOrm7Yqfs1nLo8Mc6AtWf8DpTCvjkz0ZvABH9m4e6lVorpUTPlSB2xCw_D3IERq93LjVMg4fyuW-kI
Lipb6M9vQ74mY9XRh0QkmIh8_3Nr461MaTjIGPg9fV9tS2vAF1_Y07Enu1Fd8knsIHBpL6WQm1J2A2Bj
_-8-04qKfXQ/file [following]
--2021-12-24 22:43:47--  https://ucf8fbf780e495d7f9190e14d296.dl.dropboxusercont
ent.com/cd/0/inline2/BcdiajxXXeIv-OWJFq-rvDEcL8FFHVWhJag6y7fSGKB-HEgUGyFjh9bBjej
SZ0GLJv50-H5MktiDklYBgBJx9gY5BL9h-tfhhvYmSg51TslIiT_QpjANQk7ZONH8QTEjaaUJKpSlY-4
Wjfp7f-DxKBwzpRc7e9sH8ycoR86SalFvmPy3xmWaPflP7QM78sggEUzBfXPpOibAxv9ui-QEBrf-
hQTOrm7Yqfs1nLo8Mc6AtWf8DpTCvjkz0ZvABH9m4e6lVorpUTPlSB2xCw_D3IERq93LjVMg4fyuW-kI
Lipb6M9vQ74mY9XRh0QkmIh8_3Nr461MaTjIGPg9fV9tS2vAF1_Y07Enu1Fd8knsIHBpL6WQm1J2A2Bj
_-8-04qKfXQ/file
Reusing existing connection to
ucf8fbf780e495d7f9190e14d296.dl.dropboxusercontent.com:443.
HTTP request sent, awaiting response… 200 OK
Length: 17069878 (16M) [application/octet-stream]
Saving to: ‘mnist.npz’

mnist.npz           100%[===================>]  16.28M  46.4MB/s    in 0.4s

Last-modified header missing -- time-stamps turned off.
2021-12-24 22:43:48 (46.4 MB/s) - ‘mnist.npz’ saved [17069878/17069878]
```

[ ]:
```
## LOAD the mnist data.

# To speed up training we'll only work on a subset of the data.
# Note that we reshape the data from
#  (nsamples, num_features) = (nsamples, nchannels*rows*cols)
```

```
#    -> (nsamples, nchannels, rows, cols)
# in order to retain the spatial arrangements of the pixels
data = np.load('mnist.npz')
num_classes = 10
nchannels, rows, cols = 1, 28, 28

x_train = data['X_train'][:1000].astype('float32')
x_train = x_train.reshape((-1, nchannels, rows, cols))
targets_train = data['y_train'][:1000].astype('int32')

x_valid = data['X_valid'][:500].astype('float32')
x_valid = x_valid.reshape((-1, nchannels, rows, cols))
targets_valid = data['y_valid'][:500].astype('int32')

x_test = data['X_test'][:500].astype('float32')
x_test = x_test.reshape((-1, nchannels, rows, cols))
targets_test = data['y_test'][:500].astype('int32')

print("Information on dataset")
print("x_train", x_train.shape)
# print("x_train", x_train)
print("targets_train", targets_train.shape)
print("x_valid", x_valid.shape)
print("targets_valid", targets_valid.shape)
print("x_test", x_test.shape)
print("targets_test", targets_test.shape)
```

```
Information on dataset
x_train (1000, 1, 28, 28)
targets_train (1000,)
x_valid (500, 1, 28, 28)
targets_valid (500,)
x_test (500, 1, 28, 28)
targets_test (500,)
```

[ ]:
```
## plot a few MNIST examples

idx, dim, classes = 0, 28, 10
# create empty canvas
canvas = np.zeros((dim*classes, classes*dim))

# fill with tensors
for i in range(classes):
    for j in range(classes):
        canvas[i*dim:(i+1)*dim, j*dim:(j+1)*dim] = x_train[idx].reshape((dim,␣
↪dim))
        idx += 1
```

```
# visualize matrix of tensors as gray scale image
plt.figure(figsize=(6, 6))
plt.axis('off')
plt.imshow(canvas, cmap='gray')
plt.title('MNIST handwritten digits')
plt.show()
```

6 Define a simple feed forward neural network

[ ]:
```
import torch
from torch.autograd import Variable
from torch.nn.parameter import Parameter
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.nn.init as init
```

```
from torch.nn import Linear, Conv2d, BatchNorm2d, MaxPool2d, Dropout2d
from torch.nn.functional import relu, elu, relu6, sigmoid, tanh, softmax
```

[ ]:
```
# hyperameters of the model
num_classes = 10
channels = x_train.shape[1]
print(channels)
height = x_train.shape[2]
width = x_train.shape[3]
num_filters_conv1 = 16
kernel_size_conv1 = 5 # [height, width]
stride_conv1 = 1 # [stride_height, stride_width]
num_l1 = 100
padding_conv1 = 0

def compute_conv_dim(dim_size):
    return int((dim_size - kernel_size_conv1 + 2 * padding_conv1) /␣
 ↪stride_conv1 + 1)

# define network
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # out_dim = (input_dim - filter_dim + 2padding) / stride + 1
        self.conv_1 = Conv2d(in_channels=channels,
                             out_channels=num_filters_conv1,
                             kernel_size=kernel_size_conv1,
                             stride=stride_conv1)

        self.conv_out_height = compute_conv_dim(height)
        self.conv_out_width = compute_conv_dim(width)

        # add dropout to network
        self.dropout = Dropout2d(p=0.5)
        self.l1_in_features = num_filters_conv1 * self.conv_out_height * self.
 ↪conv_out_width
        # self.l1_in_features = channels * height * width

        self.l_1 = Linear(in_features=self.l1_in_features,
                          out_features=num_l1,
                          bias=True)
        self.l_out = Linear(in_features=num_l1,
                            out_features=num_classes,
                            bias=False)
```

```
    def forward(self, x): # x.size() = [batch, channel, height, width]
        x = relu(self.conv_1(x))
        # torch.Tensor.view: http://pytorch.org/docs/master/tensors.html?
↪highlight=view#torch.Tensor.view
        #   Returns a new tensor with the same data as the self tensor,
        #   but of a different size.
        # the size -1 is inferred from other dimensions
        x = x.view(-1, self.l1_in_features)
        x = self.dropout(relu(self.l_1(x)))
        # x = relu(self.l_1(x))
        return softmax(self.l_out(x), dim=1)


net = Net()
print(net)
```

```
1
Net(
  (conv_1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.5, inplace=False)
  (l_1): Linear(in_features=9216, out_features=100, bias=True)
  (l_out): Linear(in_features=100, out_features=10, bias=False)
)
```

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

[ ]:
```
#Test the forward pass with dummy data
x = np.random.normal(0,1, (5, 1, 28, 28)).astype('float32')
out = net(Variable(torch.from_numpy(x)))
out.size(), out
```

[ ]:
```
(torch.Size([5, 10]),
 tensor([[0.1321, 0.1248, 0.0911, 0.0820, 0.0845, 0.1053, 0.0908, 0.1005,
0.0929,
          0.0959],
         [0.1060, 0.1136, 0.1061, 0.0752, 0.0928, 0.0925, 0.0905, 0.1020,
0.1027,
          0.1187],
         [0.1160, 0.0926, 0.0980, 0.0705, 0.1034, 0.0965, 0.0996, 0.1140,
0.0816,
          0.1277],
         [0.1184, 0.0914, 0.1198, 0.0995, 0.0859, 0.0894, 0.1066, 0.1109,
0.0737,
          0.1044],
         [0.1237, 0.0944, 0.1058, 0.0855, 0.0909, 0.1126, 0.1149, 0.1069,
0.0737,
          0.0917]], grad_fn=<SoftmaxBackward0>))
```

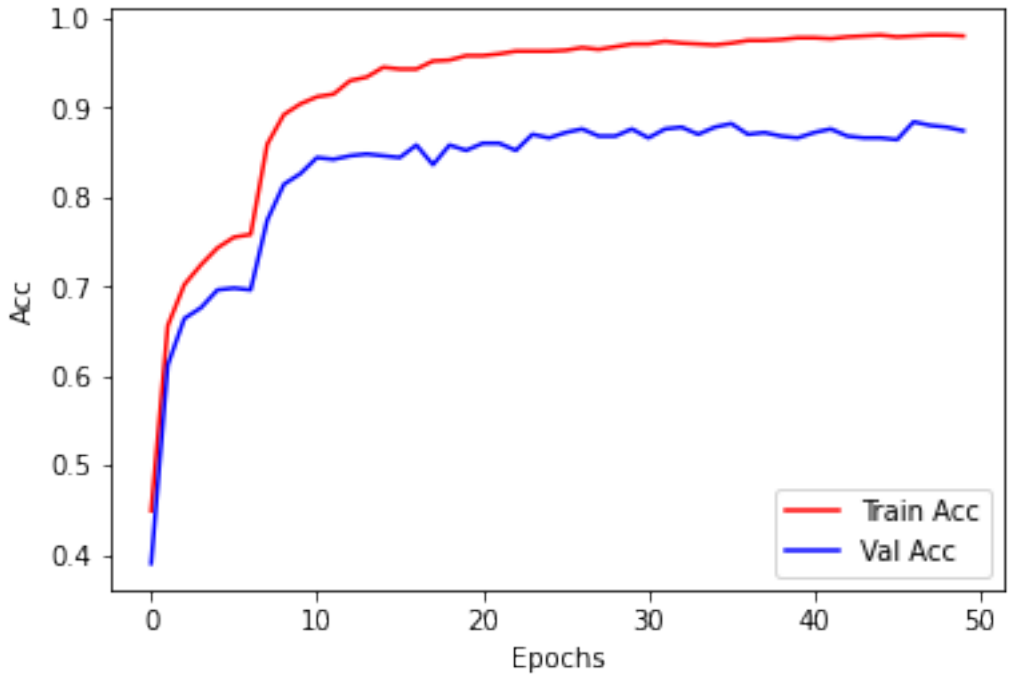
Notice how the output's probabilities are nicely distributed. The built-in nn functions (layers) alreay have a sane initialization of the weights.

[ ]:
```
# we could have done this ourselves,
# but we should be aware of sklearn and it's tools
from sklearn.metrics import accuracy_score

batch_size = 100
num_epochs = 50
num_samples_train = x_train.shape[0]
num_batches_train = num_samples_train // batch_size
num_samples_valid = x_valid.shape[0]
num_batches_valid = num_samples_valid // batch_size

train_acc, train_loss = [], []
valid_acc, valid_loss = [], []
test_acc, test_loss = [], []
cur_loss = 0
losses = []

get_slice = lambda i, size: range(i * size, (i + 1) * size)

for epoch in range(num_epochs):
    # Forward -> Backprob -> Update params
    ## Train
    cur_loss = 0
    net.train()
    for i in range(num_batches_train):
        slce = get_slice(i, batch_size)
        x_batch = Variable(torch.from_numpy(x_train[slce]))
        output = net(x_batch)

        # compute gradients given loss
        target_batch = Variable(torch.from_numpy(targets_train[slce]).long())
        batch_loss = criterion(output, target_batch)
        optimizer.zero_grad()
        batch_loss.backward()
        optimizer.step()

        cur_loss += batch_loss
    losses.append(cur_loss / batch_size)

    net.eval()
    ### Evaluate training
    train_preds, train_targs = [], []
    for i in range(num_batches_train):
        slce = get_slice(i, batch_size)
```

```
        x_batch = Variable(torch.from_numpy(x_train[slce]))

        output = net(x_batch)
        preds = torch.max(output, 1)[1]

        train_targs += list(targets_train[slce])
        train_preds += list(preds.data.numpy())

    ### Evaluate validation
    val_preds, val_targs = [], []
    for i in range(num_batches_valid):
        slce = get_slice(i, batch_size)
        x_batch = Variable(torch.from_numpy(x_valid[slce]))

        output = net(x_batch)
        preds = torch.max(output, 1)[1]
        val_preds += list(preds.data.numpy())
        val_targs += list(targets_valid[slce])

    train_acc_cur = accuracy_score(train_targs, train_preds)
    valid_acc_cur = accuracy_score(val_targs, val_preds)

    train_acc.append(train_acc_cur)
    valid_acc.append(valid_acc_cur)

    if epoch % 10 == 0:
        print("Epoch %2i : Train Loss %f , Train acc %f, Valid acc %f" % (
                epoch+1, losses[-1], train_acc_cur, valid_acc_cur))

epoch = np.arange(len(train_acc))
plt.figure()
plt.plot(epoch, train_acc, 'r', epoch, valid_acc, 'b')
plt.legend(['Train Acc', 'Val Acc'])
plt.xlabel('Epochs')
plt.ylabel('Acc')

### Evaluate test set
x_batch = Variable(torch.from_numpy(x_test))
output = net(x_batch)
preds = torch.max(output, 1)[1]
print("\nTest set Acc:  %f" % (accuracy_score(list(targets_test), list(preds.
 ↪data.numpy()))))
```

```
Epoch  1 : Train Loss 0.221385 , Train acc 0.449000, Valid acc 0.390000
Epoch 11 : Train Loss 0.162806 , Train acc 0.912000, Valid acc 0.844000
Epoch 21 : Train Loss 0.155060 , Train acc 0.958000, Valid acc 0.860000
Epoch 31 : Train Loss 0.153146 , Train acc 0.971000, Valid acc 0.866000
Epoch 41 : Train Loss 0.150639 , Train acc 0.978000, Valid acc 0.872000
```

```
Test set Acc:  0.908000
```



7 Assignment 2

1. Note the performance of the standard feedforward neural network. Add a 2D convolution layer before the dense hidden layer and confirm that it increases the generalization performance of the network (try num_filters=16 and filter_size=5 as a starting point).
2. Notice that the size of the image reduces. This can cause loss of information in convolutional networks that apply many convolutional layers. To avoid such add adequate padding to the convolutional layer.
3. Can the performance be increases even further by stacking more convolution layers ?
4. Maxpooling is a technique for decreasing the spatial resolution of an image while retaining the important features. Effectively this gives a local translational invariance and reduces the computation by a factor of four. In the classification algorithm which is usually desirable. Try to either:
    - add a maxpool layer (add arguement kernel_size=2, stride=2) after the convolution layer, or
    - set add stride=2 to the arguments of the convolution layer, make it fit with the kernel size

Verify that this decreases spatial dimension of the image (`print(l_conv_x.size())` or `print(l_maxpool_x.size())` in your forward pass). Does this increase the performance of the

network (you may need to stack multiple layers or increase the number of filters to increase performance) ?

8 Visualization of filters

Convolution filters can be interpreted as spatial feature detectors picking up different image features such as edges, corners etc. Below we provide code for visualization of the filters. The best results are obtained with fairly large filters of size 9 and either 16 or 36 filters.

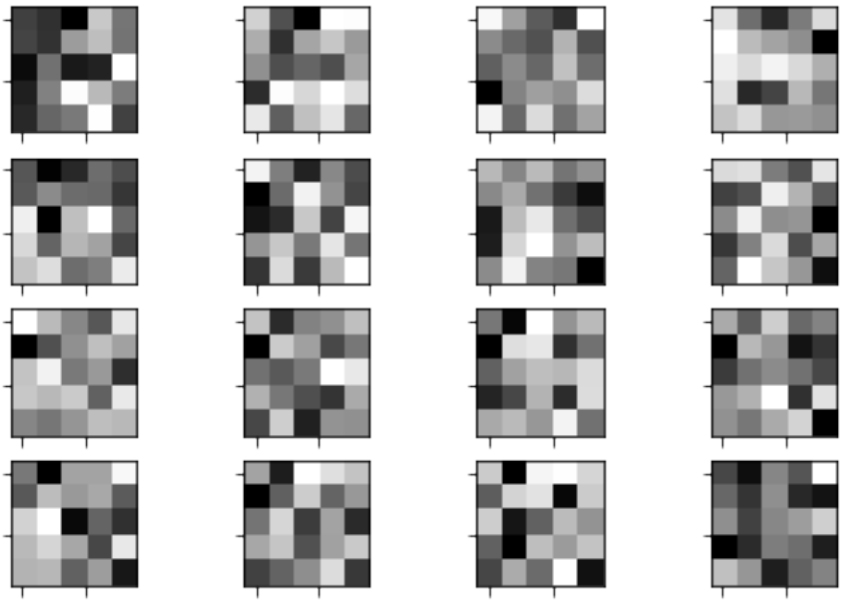
[ ]:
```
# to start with we print the names of the weights in our network
names_and_vars = {x[0]: x[1] for x in net.named_parameters()}
print(names_and_vars.keys())
```

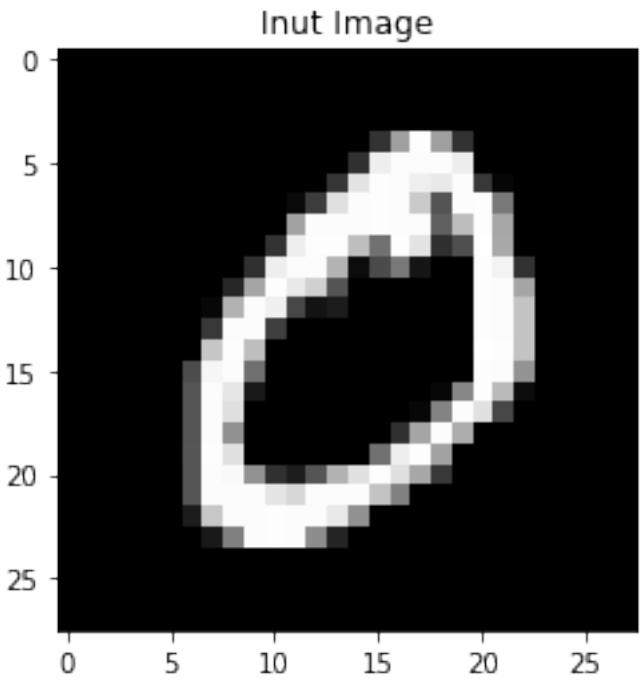
```
dict_keys(['conv_1.weight', 'conv_1.bias', 'l_1.weight', 'l_1.bias',
'l_out.weight'])
```

[ ]:
```
### ERROR - If you get a key error, then you need to define l_conv1 in your␣
↪model!
if not 'conv_1.weight' in names_and_vars:
    print("You need to go back and define a convolutional layer in the network.
↪")
else:
    np_W = names_and_vars['conv_1.weight'].data.numpy() # get the filter values␣
↪from the first conv layer
    print(np_W.shape, "i.e. the shape is (channels_out, channels_in,␣
↪filter_height, filter_width)")
    channels_out, channels_in, filter_size, _ = np_W.shape
    n = int(channels_out**0.5)

    # reshaping the last dimension to be n by n
    np_W_res = np_W.reshape(filter_size, filter_size, channels_in, n, n)
    fig, ax = plt.subplots(n,n)
    print("learned filter values")
    for i in range(n):
        for j in range(n):
            ax[i,j].imshow(np_W_res[:,:,0,i,j],␣
↪cmap='gray',interpolation='none')
            ax[i,j].xaxis.set_major_formatter(plt.NullFormatter())
            ax[i,j].yaxis.set_major_formatter(plt.NullFormatter())

    idx = 1
    plt.figure()
    plt.imshow(x_train[idx,0],cmap='gray',interpolation='none')
    plt.title('Inut Image')
    plt.show()

    #visalize the filters convolved with an input image
```

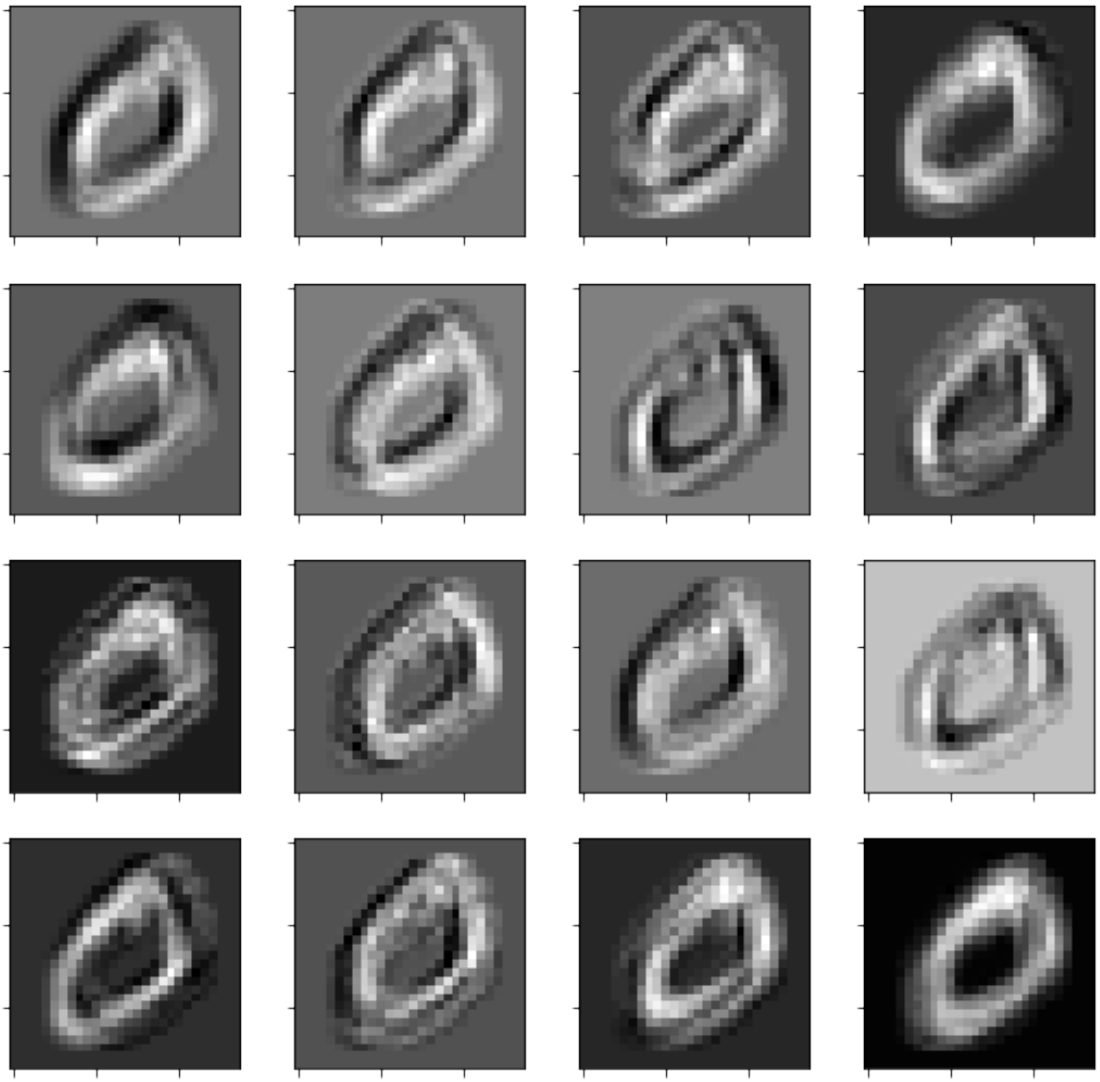
```
    from scipy.signal import convolve2d
    np_W_res = np_W.reshape(filter_size, filter_size, channels_in, n, n)
    fig, ax = plt.subplots(n,n,figsize=(9,9))
    print("Response from input image convolved with the filters")
    for i in range(n):
        for j in range(n):
            ax[i,j].imshow(convolve2d(x_train[1,0],np_W_res[:,:
↪,0,i,j],mode='same'),
                           cmap='gray',interpolation='none')
            ax[i,j].xaxis.set_major_formatter(plt.NullFormatter())
            ax[i,j].yaxis.set_major_formatter(plt.NullFormatter())
```

(16, 1, 5, 5) i.e. the shape is (channels_out, channels_in, filter_height, filter_width)
learned filter values





Response from input image convolved with the filters



9 Assignment 3

The visualized filters will likely look most like noise due to the small amount of training data.

1. Try to use 10000 traning examples instead and visualise the filters again
2. Dropout is a very usefull technique for preventing overfitting. Try to add a DropoutLayer after the convolution layer and hidden layer. This should increase both performance and the "visual appeal" of the filters
    - remember to use `net.train()` and `net.eval()` properly.
3. Batch normalization is a recent innovation for improving generalization performance. Try to insert batch normalization layers into the network to improve performance.
    - remember to use `net.train()` and `net.eval()` properly.

Again, if you didn't already, you really should watch this video.