2. Define the Network Architecture

July 8, 2018

0.1 Define the Convolutional Neural Network

After you've looked at the data you're working with and, in this case, know the shapes of the images and of the keypoints, you are ready to define a convolutional neural network that can *learn* from this data.

In this notebook and in `models.py`, you will: 1. Define a CNN with images as input and keypoints as output 2. Construct the transformed FaceKeypointsDataset, just as before 3. Train the CNN on the training data, tracking loss 4. See how the trained model performs on test data 5. If necessary, modify the CNN structure and model hyperparameters, so that it performs *well* *

* What does *well* mean?

"Well" means that the model's loss decreases during training **and**, when applied to test image data, the model produces keypoints that closely match the true keypoints of each face. And you'll see examples of this later in the notebook.

0.2 CNN Architecture

Recall that CNN's are defined by a few types of layers: * Convolutional layers * Maxpooling layers * Fully-connected layers

You are required to use the above layers and encouraged to add multiple convolutional layers and things like dropout layers that may prevent overfitting. You are also encouraged to look at literature on keypoint detection, such as this paper, to help you determine the structure of your network.

0.2.1 TODO: Define your model in the provided file `models.py` file

This file is mostly empty but contains the expected name and some TODO's for creating your model.

0.3 PyTorch Neural Nets

To define a neural network in PyTorch, you define the layers of a model in the function `__init__` and define the feedforward behavior of a network that employs those initialized layers in the function `forward`, which takes in an input image tensor, `x`. The structure of this Net class is shown below and left for you to fill in.

Note: During training, PyTorch will be able to perform backpropagation by keeping track of the network's feedforward behavior and using autograd to calculate the update to the weights in the network.

Define the Layers in `__init__` As a reminder, a conv/pool layer may be defined like this (in `__init__`):

```
# 1 input image channel (for grayscale images), 32 output channels/feature maps, 3x3 square conv
self.conv1 = nn.Conv2d(1, 32, 3)

# maxpool that uses a square window of kernel_size=2, stride=2
self.pool = nn.MaxPool2d(2, 2)
```

Refer to Layers in `forward` Then referred to in the `forward` function like this, in which the conv1 layer has a ReLu activation applied to it before maxpooling is applied:

```
x = self.pool(F.relu(self.conv1(x)))
```

Best practice is to place any layers whose weights will change during the training process in `__init__` and refer to them in the `forward` function; any layers or functions that always behave in the same way, such as a pre-defined activation function, should appear *only* in the `forward` function.

Why models.py You are tasked with defining the network in the `models.py` file so that any models you define can be saved and loaded by name in different notebooks in this project directory. For example, by defining a CNN class called `Net` in `models.py`, you can then create that same architecture in this and other notebooks by simply importing the class and instantiating a model:

```
from models import Net
net = Net()
```

```
In [1]: # load the data if you need to; if you have already loaded the data, you may comment thi
        # -- DO NOT CHANGE THIS CELL -- #
        !mkdir /data
        !wget -P /data/ https://s3.amazonaws.com/video.udacity-data.com/topher/2018/May/5aea1b91
        !unzip -n /data/train-test-data.zip -d /data
```

```
mkdir: cannot create directory /data: File exists
--2018-07-08 21:36:56--  https://s3.amazonaws.com/video.udacity-data.com/topher/2018/May/5aea1b9
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.104.101
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.104.101|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 338613624 (323M) [application/zip]
Saving to: /data/train-test-data.zip.4

train-test-data.zip 100%[===================>] 322.93M  41.1MB/s    in 7.8s

2018-07-08 21:37:04 (41.3 MB/s) - /data/train-test-data.zip.4 saved [338613624/338613624]
```

```
Archive:  /data/train-test-data.zip
```

Note: Workspaces automatically close connections after 30 minutes of inactivity (including inactivity while training!). Use the code snippet below to keep your workspace alive during training. (The active_session context manager is imported below.)

```
from workspace_utils import active_session

with active_session():
    train_model(num_epochs)
```

```
In [2]: # import the usual resources
        import matplotlib.pyplot as plt
        import numpy as np

        # import utilities to keep workspaces alive during model training
        from workspace_utils import active_session

        # watch for any changes in model.py, if it changes, re-load it automatically
        %load_ext autoreload
        %autoreload 2
```

```
In [3]: ## TODO: Define the Net in models.py

        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import matplotlib.pyplot as plt

        ## TODO: Once you've define the network, you can instantiate it
        # one example conv layer has been provided for you
        from models import Net

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

```
Net(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (conv1_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout2d(p=0.2)
  (conv2): Conv2d(32, 48, kernel_size=(5, 5), stride=(1, 1))
  (conv2_bn): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout2): Dropout2d(p=0.2)
  (conv3): Conv2d(48, 48, kernel_size=(3, 3), stride=(1, 1))
  (conv3_bn): BatchNorm2d(48, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout3): Dropout2d(p=0.2)
  (conv4): Conv2d(48, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=7744, out_features=4096, bias=True)
  (fc1_bn): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1_drop): Dropout(p=0.4)
  (fc2): Linear(in_features=4096, out_features=1028, bias=True)
  (output): Linear(in_features=1028, out_features=136, bias=True)
)
```

0.4 Transform the dataset

To prepare for training, create a transformed dataset of images and keypoints.

0.4.1 TODO: Define a data transform

In PyTorch, a convolutional neural network expects a torch image of a consistent size as input. For efficient training, and so your model's loss does not blow up during training, it is also suggested that you normalize the input images and keypoints. The necessary transforms have been defined in `data_load.py` and you **do not** need to modify these; take a look at this file (you'll see the same transforms that were defined and applied in Notebook 1).

To define the data transform below, use a composition of: 1. Rescaling and/or cropping the data, such that you are left with a square image (the suggested size is 224x224px) 2. Normalizing the images and keypoints; turning each RGB image into a grayscale image with a color range of [0, 1] and transforming the given keypoints into a range of [-1, 1] 3. Turning these images and keypoints into Tensors

These transformations have been defined in `data_load.py`, but it's up to you to call them and create a `data_transform` below. **This transform will be applied to the training data and, later, the test data**. It will change how you go about displaying these images and keypoints, but these steps are essential for efficient training.

As a note, should you want to perform data augmentation (which is optional in this project), and randomly rotate or shift these images, a square image size will be useful; rotating a 224x224 image by 90 degrees will result in the same shape of output.

In [4]:
```
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils

# the dataset we created in Notebook 1 is copied in the helper file `data_load.py`
from data_load import FacialKeypointsDataset
# the transforms we defined in Notebook 1 are in the helper file `data_load.py`
from data_load import Rescale, RandomCrop, Normalize, ToTensor


## TODO: define the data_transform using transforms.Compose([all tx's, . , .])
# order matters! i.e. rescaling should come before a smaller crop
```

```
        data_transform=transforms.Compose([Rescale(250),
                                           RandomCrop(224),
                                           Normalize(),
                                           ToTensor()])

        # testing that you've defined a transform
        assert(data_transform is not None), 'Define a data_transform'
```

```
In [5]: # create the transformed dataset
        transformed_dataset = FacialKeypointsDataset(csv_file='/data/training_frames_keypoints.c
                                                     root_dir='/data/training/',
                                                     transform=data_transform)


        print('Number of images: ', len(transformed_dataset))

        # iterate through the transformed dataset and print some stats about the first few sampl
        for i in range(4):
            sample = transformed_dataset[i]
            print(i, sample['image'].size(), sample['keypoints'].size())
```

```
Number of images:  3462
0 torch.Size([1, 224, 224]) torch.Size([68, 2])
1 torch.Size([1, 224, 224]) torch.Size([68, 2])
2 torch.Size([1, 224, 224]) torch.Size([68, 2])
3 torch.Size([1, 224, 224]) torch.Size([68, 2])
```

0.5 Batching and loading data

Next, having defined the transformed dataset, we can use PyTorch's DataLoader class to load the training data in batches of whatever size as well as to shuffle the data for training the model. You can read more about the parameters of the DataLoader, in this documentation.

Batch size Decide on a good batch size for training your model. Try both small and large batch sizes and note how the loss decreases as the model trains. Too large a batch size may cause your model to crash and/or run out of memory while training.

Note for Windows users: Please change the `num_workers` to 0 or you may face some issues with your DataLoader failing.

```
In [6]: # load training data in batches
        # batch_size = 10

        batch_size = 16

        train_loader = DataLoader(transformed_dataset,
                                  batch_size=batch_size,
```

```
                                          shuffle=True,
                                          num_workers=4)
```

0.6 Before training

Take a look at how this model performs before it trains. You should see that the keypoints it predicts start off in one spot and don't match the keypoints on a face at all! It's interesting to visualize this behavior so that you can compare it to the model after training and see how the model has improved.

Load in the test dataset The test dataset is one that this model has *not* seen before, meaning it has not trained with these images. We'll load in this test data and before and after training, see how your model performs on this set!

To visualize this test data, we have to go through some un-transformation steps to turn our images into python images from tensors and to turn our keypoints back into a recognizable range.

```
In [7]: # load in the test data, using the dataset class
        # AND apply the data_transform you defined above

        # create the test dataset
        test_dataset = FacialKeypointsDataset(csv_file='/data/test_frames_keypoints.csv',
                                              root_dir='/data/test/',
                                              transform=data_transform)
```

```
In [8]: # load test data in batches
        # batch_size = 10

        batch_size = 16

        test_loader = DataLoader(test_dataset,
                                 batch_size=batch_size,
                                 shuffle=True,
                                 num_workers=4)
```

0.7 Apply the model on a test sample

To test the model on a test sample of data, you have to follow these steps: 1. Extract the image and ground truth keypoints from a sample 2. Wrap the image in a Variable, so that the net can process it as input and track how it changes as the image moves through the network. 3. Make sure the image is a FloatTensor, which the model expects. 4. Forward pass the image through the net to get the predicted, output keypoints.

This function test how the network performs on the first batch of test data. It returns the images, the transformed images, the predicted keypoints (produced by the model), and the ground truth keypoints.

```
In [9]: # test the model on a batch of test images

        def net_sample_output():
```

```
    # iterate through the test dataset
    for i, sample in enumerate(test_loader):

        # get sample data: images and ground truth keypoints
        images = sample['image']
        key_pts = sample['keypoints']

        # convert images to FloatTensors
        images = images.type(torch.FloatTensor)

        # my code
        #print(images.shape)

        # forward pass to get net output
        output_pts = net(images)

        # my code
        #print(output_pts.shape)

        # reshape to batch_size x 68 x 2 pts
        output_pts = output_pts.view(output_pts.size()[0], 68, -1)

        # break after first image is tested
        if i == 0:
            return images, output_pts, key_pts
```

Debugging tips If you get a size or dimension error here, make sure that your network outputs the expected number of keypoints! Or if you get a Tensor type error, look into changing the above code that casts the data into float types: `images = images.type(torch.FloatTensor)`.

```
In [10]: # call the above function
         # returns: test images, test predicted keypoints, test ground truth keypoints
         test_images, test_outputs, gt_pts = net_sample_output()

         # print out the dimensions of the data to see if they make sense
         print(test_images.data.size())
         print(test_outputs.data.size())
         print(gt_pts.size())
```

```
torch.Size([16, 1, 224, 224])
torch.Size([16, 68, 2])
torch.Size([16, 68, 2])
```

0.8 Visualize the predicted keypoints

Once we've had the model produce some predicted output keypoints, we can visualize these points in a way that's similar to how we've displayed this data before, only this time, we have to "un-transform" the image/keypoint data to display it.

Note that I've defined a *new* function, `show_all_keypoints` that displays a grayscale image, its predicted keypoints and its ground truth keypoints (if provided).

```
In [11]: def show_all_keypoints(image, predicted_key_pts, gt_pts=None):
             """Show image with predicted keypoints"""
             # image is grayscale
             plt.imshow(image, cmap='gray')
             plt.scatter(predicted_key_pts[:, 0], predicted_key_pts[:, 1], s=20, marker='.', c='
             # plot ground truth points as green pts
             if gt_pts is not None:
                 plt.scatter(gt_pts[:, 0], gt_pts[:, 1], s=20, marker='.', c='g')
```

Un-transformation Next, you'll see a helper function. `visualize_output` that takes in a batch of images, predicted keypoints, and ground truth keypoints and displays a set of those images and their true/predicted keypoints.

This function's main role is to take batches of image and keypoint data (the input and output of your CNN), and transform them into numpy images and un-normalized keypoints (x, y) for normal display. The un-transformation process turns keypoints and images into numpy arrays from Tensors *and* it undoes the keypoint normalization done in the Normalize() transform; it's assumed that you applied these transformations when you loaded your test data.

```
In [12]: # visualize the output
         # by default this shows a batch of 10 images
         def visualize_output(test_images, test_outputs, gt_pts=None, batch_size=10):

             for i in range(batch_size):
                 plt.figure(figsize=(20,10))
                 ax = plt.subplot(1, batch_size, i+1)

                 # un-transform the image data
                 image = test_images[i].data   # get the image from it's Variable wrapper
                 image = image.numpy()   # convert to numpy array from a Tensor
                 image = np.transpose(image, (1, 2, 0))   # transpose to go from torch to numpy

                 # un-transform the predicted key_pts data
                 predicted_key_pts = test_outputs[i].data
                 predicted_key_pts = predicted_key_pts.numpy()
                 # undo normalization of keypoints
                 predicted_key_pts = predicted_key_pts*50.0+100

                 # plot ground truth points for comparison, if they exist
                 ground_truth_pts = None
                 if gt_pts is not None:
```

```
            ground_truth_pts = gt_pts[i]
            ground_truth_pts = ground_truth_pts*50.0+100

        # call show_all_keypoints
        show_all_keypoints(np.squeeze(image), predicted_key_pts, ground_truth_pts)

        plt.axis('off')

    plt.show()

# call it
visualize_output(test_images, test_outputs, gt_pts)
```

0.9 Training

Loss function Training a network to predict keypoints is different than training a network to predict a class; instead of outputting a distribution of classes and using cross entropy loss, you may want to choose a loss function that is suited for regression, which directly compares a predicted value and target value. Read about the various kinds of loss functions (like MSE or L1/SmoothL1 loss) in this documentation.

0.9.1 TODO: Define the loss and optimization

Next, you'll define how the model will train by deciding on the loss function and optimizer.

```
In [13]: ## TODO: Define the loss and optimization
         import torch.optim as optim

         #criterion = nn.MSELoss()
         criterion = nn.SmoothL1Loss(reduce=False, size_average=False)

         optimizer = optim.Adam(net.parameters(),lr=0.001) # default 0.001
         # optimizer = torch.optim.SGD(net.parameters(), lr = 0.5)
```

0.10 Training and Initial Observation

Now, you'll train on your batched training data from `train_loader` for a number of epochs.

To quickly observe how your model is training and decide on whether or not you should modify it's structure or hyperparameters, you're encouraged to start off with just one or two epochs at first. As you train, note how your the model's loss behaves over time: does it decrease quickly at first and then slow down? Does it take a while to decrease in the first place? What happens if you change the batch size of your training data or modify your loss function? etc.

Use these initial observations to make changes to your model and decide on the best architecture before you train for many epochs and create a final model.

```
In [14]: def train_net(n_epochs):

             # prepare the net for training
             net.train()

             for epoch in range(n_epochs):  # loop over the dataset multiple times

                 running_loss = 0.0
                 loss_per_10_batch=[]
                 loss_per_epoch=[]
                 # train on batches of data, assumes you already have train_loader
                 for batch_i, data in enumerate(train_loader):
                     # get the input images and their corresponding labels
                     images = data['image']
                     key_pts = data['keypoints']

                     # flatten pts
                     key_pts = key_pts.view(key_pts.size(0), -1)

                     # convert variables to floats for regression loss
                     # key_pts = key_pts.type(torch.FloatTensor)
                     # images = images.type(torch.FloatTensor)

                      # convert variables to long for regression loss
                     key_pts = key_pts.type(torch.LongTensor)
                     images = images.type(torch.LongTensor)

                     # forward pass to get outputs
                     output_pts = net(images)

                     # mycode
                     output_pts = output_pts.type(torch.LongTensor)

                     # calculate the loss between predicted and target keypoints
                     loss = criterion(output_pts, key_pts)

                     # zero the parameter (weight) gradients
                     optimizer.zero_grad()

                     # backward pass to calculate the weight gradients
                     loss.backward()

                     # update the weights
                     optimizer.step()

                     # print loss statistics
                     erery_loss=loss.item()
                     running_loss += loss.item()
```

```
                    loss_per_epoch.append(erery_loss)

                    if batch_i % 10 == 9:    # print every 10 batches
                        print('Epoch: {}, Batch: {}, Avg. Loss: {}'.format(epoch + 1, batch_i+1
                        #running_loss = 0.0

                        loss_per_10_batch.append(running_loss/10)
                        #last_loss=running_loss
                        running_loss = 0.0


            plt.figure()
            #plt.semilogy(loss_per_10_batch)
            plt.plot(loss_per_10_batch)
            plt.grid()
            plt.xlabel('10th batch number')
            plt.ylabel('loss')
            plt.ylim(0.0,5.0)

            plt.figure()
            plt.plot(loss_per_epoch)
            plt.grid()
            plt.xlabel('epoch')
            plt.ylabel('loss')
            plt.ylim(0.0,5.0)

            plt.show()

            print('Finished Training')
```

In [15]:
```
# train your network
# n_epochs = 1 # start small, and increase when you've decided on your model structure

# my code
n_epochs =1

# this is a Workspaces-specific context manager to keep the connection
# alive while training your model, not part of pytorch
with active_session():
    train_net(n_epochs)
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-15-875163d95bcc> in <module>()
  8 # alive while training your model, not part of pytorch
```

```
      9 with active_session():
---> 10     train_net(n_epochs)


    <ipython-input-14-2a59d1ceeedd> in train_net(n_epochs)
     27
     28             # forward pass to get outputs
---> 29             output_pts = net(images)
     30
     31             # mycode


    /opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)


    /home/workspace/models.py in forward(self, x)
    103
    104     def forward(self, x):
--> 105         x = F.relu(self.pool1(self.conv1_bn(self.conv1(x))))
    106         x = self.dropout1(x)
    107         x = F.relu(self.pool2(self.conv2_bn(self.conv2(x))))


    /opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)


    /opt/conda/lib/python3.6/site-packages/torch/nn/modules/conv.py in forward(self, input)
    299     def forward(self, input):
    300         return F.conv2d(input, self.weight, self.bias, self.stride,
--> 301                         self.padding, self.dilation, self.groups)
    302
    303


    RuntimeError: thnn_conv2d_forward is not implemented for type torch.LongTensor
```

0.11 Test data

See how your model performs on previously unseen, test data. We've already loaded and transformed this data, similar to the training data. Next, run your trained model on these images to see what kind of keypoints are produced. You should be able to see if your model is fitting each new face it sees, if the points are distributed randomly, or if the points have actually overfitted the training data and do not generalize.

```
In [ ]: # get a sample of test data again
        test_images, test_outputs, gt_pts = net_sample_output()

        print(test_images.data.size())
        print(test_outputs.data.size())
        print(gt_pts.size())
```

```
In [ ]: ## TODO: visualize your test output
        # you can use the same function as before, by un-commenting the line below:

        visualize_output(test_images, test_outputs, gt_pts)
```

Once you've found a good model (or two), save your model so you can load it and use it later!

Save your models but please **delete any checkpoints and saved models before you submit your project** otherwise your workspace may be too large to submit.

```
In [ ]: ## TODO: change the name to something uniqe for each new model
        model_dir = 'saved_models/'
        model_name = 'keypoints_model_1.pt'

        # after training, save your model parameters in the dir 'saved_models'
        torch.save(net.state_dict(), model_dir+model_name)
```

After you've trained a well-performing model, answer the following questions so that we have some insight into your training and architecture selection process. Answering all questions is required to pass this project.

0.11.1 Question 1: What optimization and loss functions did you choose and why?

Answer: write your answer here (double click to edit this cell) criterion = nn.MSELoss() optimizer = optim.Adam(net.parameters(),lr=0.001)"

my answer is as below: I use MSELoss(Mean squared error,MSE) as my loss functions,mathematical formula is sqrt(((predictions - targets) ** 2).mean())It is a common regression loss and calculation simply.It is easy to understand for me.

I use Adam(Adaptive Moment Estimation) as my optimizer,calculation efficiently and less memory requirement. Convergence more quickly at lr=0.001 than lr=0.01,and predicted keypoints visualize better than lr=0.01. I have modified the learning rate at 0.001. I use cpu to compute. my reference material of Adam as below https://www.jianshu.com/p/548049548fb9

I have tried SGD optimizer.compared with Adam optimizer,the predicted keypoint represent not well,and has slower convergence.

0.11.2 Question 2: What kind of network architecture did you start with and how did it change as you tried different architectures? Did you decide to add more convolutional layers or any layers to avoid overfitting the data?

Answer: write your answer here At beginning, I use a 3 convolutional layer nural network.I tried to train it,and the result didn't show well. I add normalization layer,and the error at initializtion became smaller. I add dropout layer to avoid overfitting the data.

0.11.3 Question 3: How did you decide on the number of epochs and batch_size to train your model?

Answer: write your answer here

When dataset is large,I use batch_size to train net.When batch_size is too small ,the training data is hard to convergence,and cause underfitting.Increasing batch_size within a reasonable range,memory utilization improves.If batch_size increasing too large,then the memory capacity is insufficient.The choice of batchsize is to find the best balance between memory efficiency and memory capacity.

Differernt dataset have different appropriate numbers of epoches.at beginning,I use less epoches,may cause under-fitting,and the result didn't represent well.So I increase the epoches gradually.

0.12 Feature Visualization

Sometimes, neural networks are thought of as a black box, given some input, they learn to produce some output. CNN's are actually learning to recognize a variety of spatial patterns and you can visualize what each convolutional layer has been trained to recognize by looking at the weights that make up each convolutional kernel and applying those one at a time to a sample image. This technique is called feature visualization and it's useful for understanding the inner workings of a CNN.

In the cell below, you can see how to extract a single filter (by index) from your first convolutional layer. The filter should appear as a grayscale grid.

```
In [ ]: # Get the weights in the first conv layer, "conv1"
        # if necessary, change this to reflect the name of your first conv layer
        weights1 = net.conv1.weight.data

        w = weights1.numpy()

        filter_index = 0

        print(w[filter_index][0])
        print(w[filter_index][0].shape)

        # display the filter weights
        plt.imshow(w[filter_index][0], cmap='gray')
```

0.13 Feature maps

Each CNN has at least one convolutional layer that is composed of stacked filters (also known as convolutional kernels). As a CNN trains, it learns what weights to include in it's convolutional kernels and when these kernels are applied to some input image, they produce a set of **feature maps**. So, feature maps are just sets of filtered images; they are the images produced by applying a convolutional kernel to an input image. These maps show us the features that the different layers of the neural network learn to extract. For example, you might imagine a convolutional kernel that detects the vertical edges of a face or another one that detects the corners of eyes. You can see what kind of features each of these kernels detects by applying them to an image. One such example is shown below; from the way it brings out the lines in an the image, you might characterize this as an edge detection filter.

Next, choose a test image and filter it with one of the convolutional kernels in your trained CNN; look at the filtered output to get an idea what that particular kernel detects.

0.14 ### TODO: Filter an image to see the effect of a convolutional kernel

In [ ]:
```
##TODO: load in and display any image from the transformed test dataset
import cv2
image = test_images[1].data   # get the image from it's Variable wrapper
image = image.numpy()   # convert to numpy array from a Tensor  C H W
image = np.transpose(image, (1, 2, 0))   # transpose to go from torch to numpy image  op
print(image.shape)

## TODO: Using cv's filter2D function,
## apply a specific set of filter weights (like the one displayed above) to the test ima
print(w[0][0]*10)

filter_image=cv2.filter2D(image,-1,w[0][0])
#fig,(ax1,ax2)=plt.subplot(1,2)
fig,(ax1,ax2)=plt.subplots(1,2)

ax1.set_title('original image')
ax2.set_title('filtered image')
ax1.imshow(image.squeeze(),cmap='gray')
ax2.imshow(filter_image,cmap='gray')
```

0.14.1 Question 4: Choose one filter from your trained CNN and apply it to a test image; what purpose do you think it plays? What kind of feature do you think it detects?

Answer: (does it detect vertical lines or does it blur out noise, etc.) write your answer here It blur out noise,and plays a role in edging image in vertical mainly.since the filter weights w[0][0] have the alignment order as below,the weight round in int format infer roughly:

[[1 -1 1 -1 1] [0 -1 1 -1 -0] [-0 1 0 1 1] [1 -1 -0 -1 -0] [-1 1 -0 -0 -1]]

1 —

1.1 Moving on!

Now that you've defined and trained your model (and saved the best model), you are ready to move on to the last notebook, which combines a face detector with your saved model to create a facial keypoint detection system that can predict the keypoints on *any* face in an image!