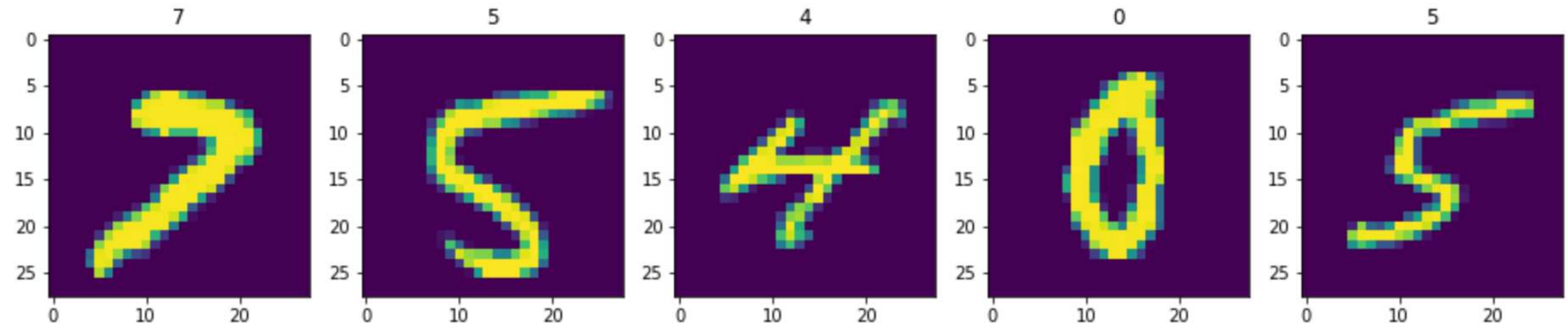
Machine Learning

Lecture 15-16

One of the benchmarks for a classification model is how well it performs on a standard data set known as MNIST.

It is a large number of handwritten digits 0-9 (70,000), digitized to images of size 28 x 28 pixels.

The model is trained on a subset of the images and then the accuracy on the test set is computed



```
import torch
import torch.nn as nn
import torchvision.models as models
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
```

Each image is a gray scale with each pixel having a value between 0 and 255.

We first turn the images into tensors and normalize the images so the pixels have values between 0.5 and 1.0.

We can do this very easily using the transforms in torchvision

```
1  trans = transforms.Compose([transforms.ToTensor(),
2                              transforms.Normalize((0.5,),(1.0,))])
```

We can download the data from torchvision.datasets

```
train_set = dsets.MNIST(root=root,train=True,transform=trans,download=True)
test_set = dsets.MNIST(root=root,train=False,transform=trans,download=True)
```

and at the same time apply the transforms

The downloaded files are already pytorch datasets

```
type(train_set)
```

executed in 4ms, finished 14:32:36 2019-06-03

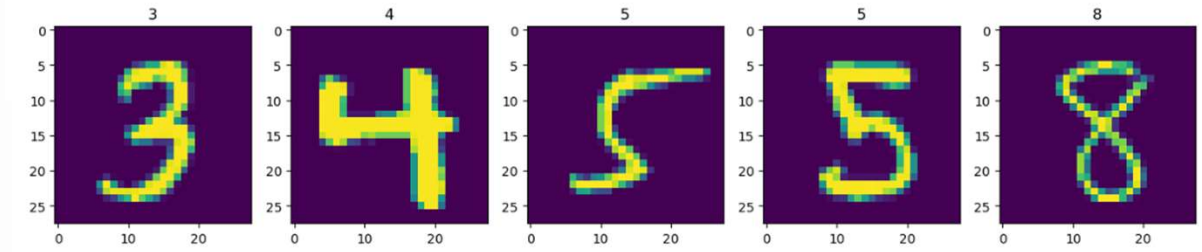
```
torchvision.datasets.mnist.MNIST
```

so we can put them directly into DataLoader objects

```
from torch.utils.data import DataLoader
batch_size=100
train_loader = DataLoader(dataset=train_set,batch_size=batch_size,shuffle=True)
test_loader = DataLoader(dataset=test_set,batch_size=batch_size,shuffle=False)
```

We can display a random sample of images

```
1  fig, axes = plt.subplots(1,5,figsize=(16,16))
2
3  for i in range(5):
4      j = np.random.randint(0,len(train_set))
5
6      axes[i].imshow(train_set[j][0].numpy().reshape(28,28)
7      axes[i].set_title(train_set[j][1])
```



Our model is a very simple sequential model which we can write without actually define a class, using nn.Sequential

```
model = nn.Sequential(nn.Linear(28*28,500),
                      nn.ReLU(),
                      nn.Linear(500,256),
                      nn.ReLU(),
                      nn.Linear(256,10)
                        )
```

The model takes in a 28*28 = 784 dimensional vector.

The first layer outputs a 500 dimensional vector and uses a ReLU activation function

The next layer takes as input a 500 dimensional vector and outputs a 256 dimensional vector and uses a ReLU activation function.

The final layer inputs a 256 dimensional vector and outputs a 10 dimensional vector, one dimension for each of the digits.

We use SGD with momentum as the optimizer and the loss function is CrossEntropy

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

criterion = nn.CrossEntropyLoss()
```

We get the loss function by applying the softmax function to the output vector

$$\underline{x} = (x_0, x_1, x_2, \ldots, x_9)$$

$$p_i(\underline{x}) = p(i|\underline{x}) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

If we let $y_{\underline{x}}$ be the label of the input (i.e. the actual digit the input image represents) and we also let $y_{\underline{x}}$ denote the one_hot_encoded vector i.e. the vector with 1 in the y'th position and 0s in the other positions then the CrossEntropy loss function is

$$\mathcal{L} = -\frac{1}{N}\sum_{\underline{x}} y_{\underline{x}} \cdot \log p(\underline{x})$$

```python
epochs = 10
losses = []
for epoch in range(epochs):
    ave_loss = 0.0
    for  i,(x,y) in enumerate(train_loader):

        x = x.reshape(-1,28*28).cuda()
        y = y.cuda()
        output = model(x)

        optimizer.zero_grad()

        loss = criterion(output,y)

        ave_loss = ave_loss * 0.9 + loss.item() * 0.1

        loss.backward()

        optimizer.step()

        if (i+1) % 100 == 0 or (i+1) == len(train_loader):
            losses.append(ave_loss)
            print('==>>> epoch: %d, batch index: %d, train loss: %.6f' % (epoch, i+1, ave_loss))

```

We flatten the image to a 784 dimensional vector

The training set contains of tuples of a tensor of dimension (28,28) and a tensor of dimension 0

```
train_set[9]
```

executed in 33ms, finished 15:24:32 2019-06-03

```
         -0.5000, -0.5000, -0.5000, -0.0216,  0.4882, -0.1784, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000],
        [-0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000],
        [-0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000],
        [-0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000],
        [-0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000,
         -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000, -0.5000]]]),
 tensor(4))
```

The train_loader object outputs batches of these tuples i.e. tensors of dimension (100,28,28) (we wrap the train_loader in an enumerate object, which just sequentially assigns a number to each entry).

Each output for each output i,(x,y) we flatten the (100,28,28) tensor to a (100,28*28 = 784) dimensional tensor which we can run through the network

```
epochs = 10
losses = []
for epoch in range(epochs):
    ave_loss = 0.0
    for  i,(x,y) in enumerate(train_loader):

        x = x.reshape(-1,28*28).cuda()
        y = y.cuda()
        output = model(x)
```

We compute the running average loss

```
optimizer.zero_grad()

loss = criterion(output,y)

ave_loss = ave_loss * (i/(i+1)) +  loss.item()*(1/(i+1))
```

and print out the average loss for each 100 batches (each batch has 100 data items)

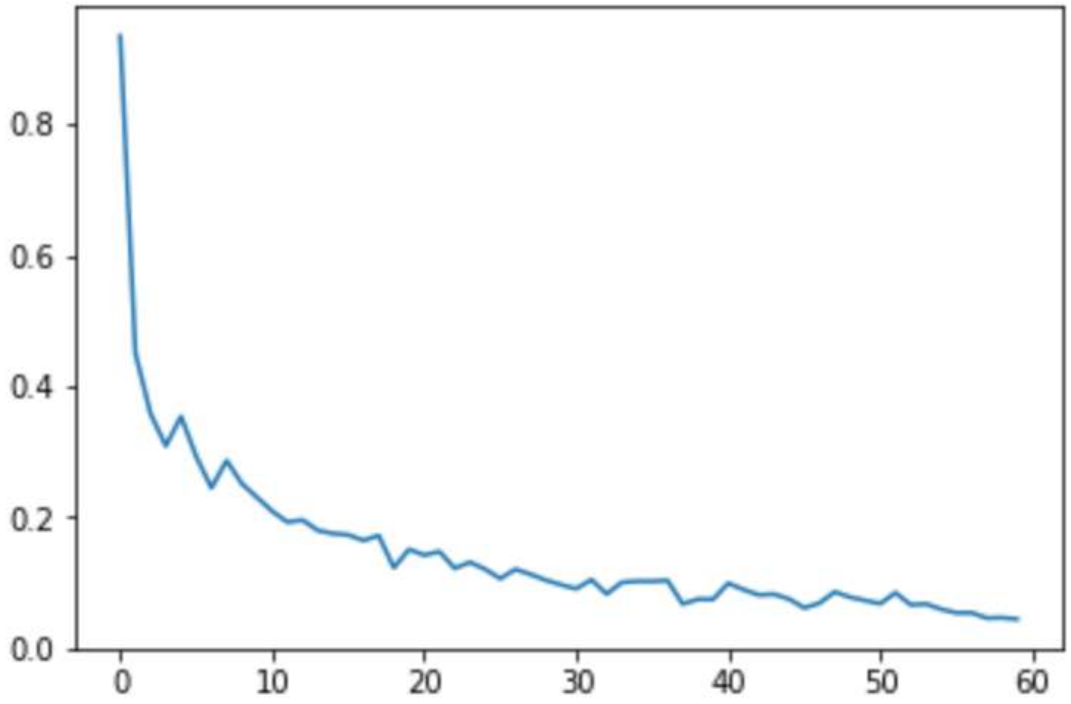
```
        loss.backward()

        optimizer.step()

        if (i+1) % 100 == 0 or (i+1) == len(train_loader):
            losses.append(ave_loss)
            print('==>>> epoch: %d, batch index: %d, train loss: %.6f' % (epoch, i+1, ave_loss))
```

ıted in 1m 27.9s, finished 13:33:59 2019-06-03

```
·> epoch: 0, batch index: 100, train loss: 0.934449
·> epoch: 0, batch index: 200, train loss: 0.450055
·> epoch: 0, batch index: 300, train loss: 0.357601
·> epoch: 0, batch index: 400, train loss: 0.308669
·> epoch: 0, batch index: 500, train loss: 0.353723
·> epoch: 0, batch index: 600, train loss: 0.292397
·> epoch: 1, batch index: 100, train loss: 0.244728
```

```
1  plt.plot(losses);
```

executed in 161ms, finished 16:51:41 2019-06-03



Next we use the trained network to predict the labels for the test set

```
predictions = []
for i in range(len(test_set)):
    z = model(test_set[i][0].reshape(-1,28*28).cuda())
    predictions.append(z.argmax().cpu().item())
```

Remark that the prediction is the label with the highest probability under the softmax distribution but since the softmax function is increasing we can compute this simply by computing the position of the maximal coordinate in the 10-dimensional output of the network

```
1  predictions = np.array(predictions)
```

executed in 5ms, finished 13:34:25 2019-06-03

```
1  test_set.test_labels.numpy()
```

executed in 6ms, finished 13:34:27 2019-06-03

```
array([7, 2, 1, ..., 4, 5, 6], dtype=int64)
```

```
1  len(predictions[predictions == test_set.test_labels.numpy()])/10000
```

executed in 4ms, finished 13:34:32 2019-06-03

```
0.976
```

Adding another layer

```
model = nn.Sequential(nn.Linear(28*28,500),
                      nn.ReLU(),
                      nn.Linear(500,256),
                      nn.ReLU(),
                      nn.Linear(256,128),
                      nn.ReLU(),
                      nn.Linear(128,10)
                        )
```

gives a slightly higher precision on the test set

```
len(predictions[predictions == test_set.test_labels.numpy()])/10000
```

executed in 15ms, finished 17:07:09 2019-06-03

```
0.9785
```

The first model has 500*256 + 256*10 + 500 + 10 = 131,070 parameters

The second has 500*256 + 256*128 + 128*10 + 500 + 256 + 128 + 10 = 128,000 + 32,768 + 1280 + 500 + 256 + 128 + 10 = 162,814

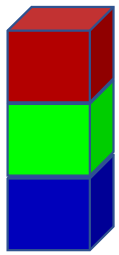
Convolutional Networks

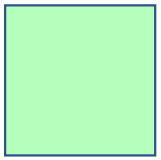
Convolutional networks have been very successful in image classification and they can also be used in analysing sequences. We will first look at an example with classifying images.

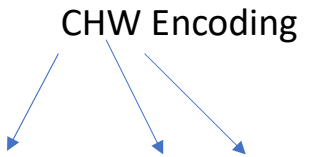
An image in a computer is represented by layers of matrices. For instance a 128×128 RGB image consist of 3 layers of 128×128 matrices. Each entry in one of the three layers (Red, Green or Blue) is a number (between 0 and 255) which indicates the intensity of the particular color.

A pixel

R = 179
G = 255
B = 187







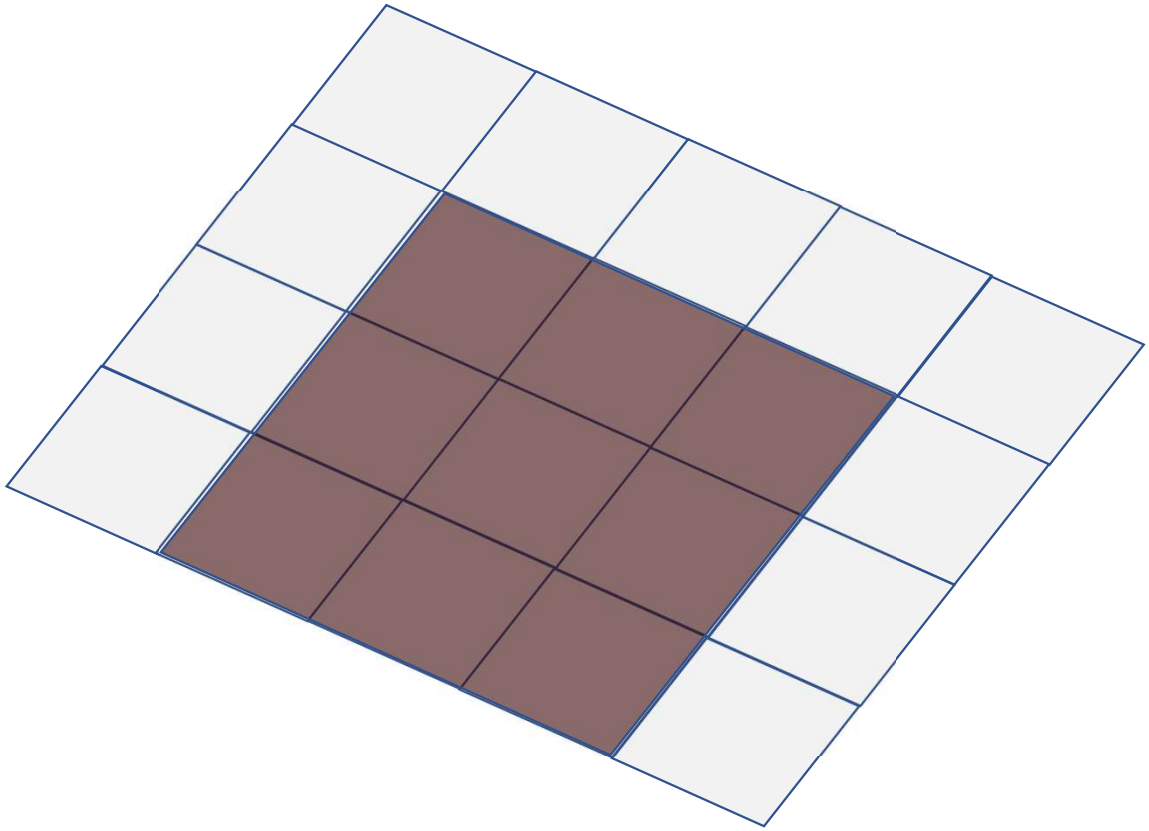
Thus a color image is represented by a tensor of dimension $3 \times height \times width$. This encoding is CHW which is used by Pytorch, Tensorflow used HWC.

A gray scale image is represented by a single matrix, where each entry represents the gray scale value (from 0 = Black to 255 = White) so a gray scale image is encoded as a $1 \times height \times width$ tensor.

Gray scale = 166

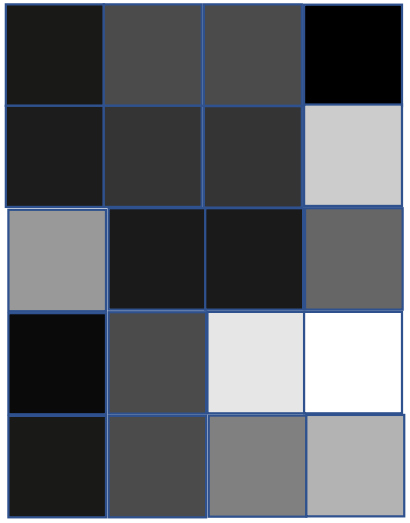
The *convolution* of two matrices A and B (say $n \times m$) is simply the dot product i.e. $A * B = \sum_{ij} a_{ij} b_{ij}$

In the computer graphics world, a matrix A is called a filter and by convolving A with a submatrix of an image we obtain a new numerical value



Assume the image has gray scale values

$$\begin{bmatrix} 0.1 & 0.3 & 0.3 & 0.0 \\ 0.15 & 0.2 & 0.2 & 0.8 \\ 0.6 & 0.1 & 0.1 & 0.4 \\ 0.04 & 0.3 & 0.9 & 1.0 \\ 0.1 & 0.3 & 0.5 & 0.7 \end{bmatrix}$$



Let A be the matrix

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

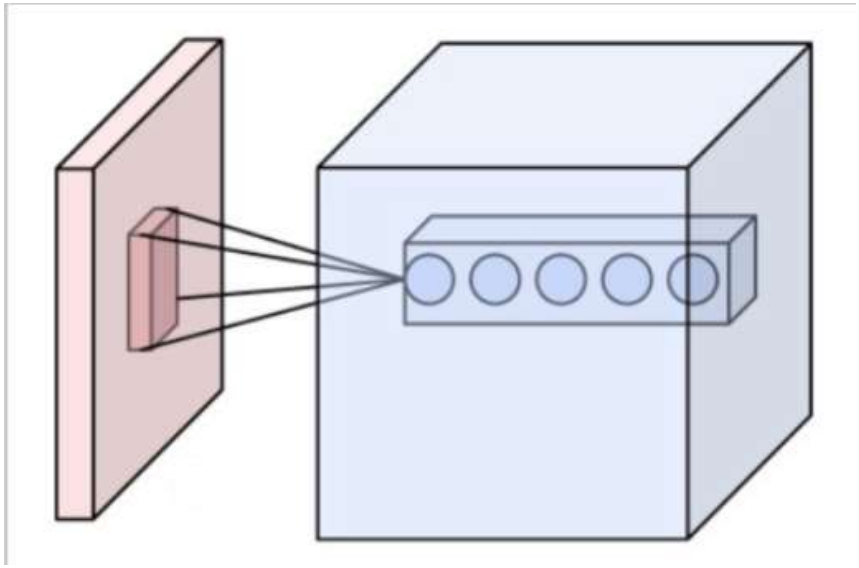
Placing the matrix A in the upper left corner and convolving we get 0.85, moving it one pixel to the right and convolving we get 1.4. Moving it over to the left edge, one pixel down, we get 1.35 continuing this way, we get the convolution with A as the matrix

$$\begin{bmatrix} 0.85 & 1.4 \\ 1.35 & 1.9 \\ 1.94 & 2.3 \end{bmatrix} + bias$$

We may add a bias b (=a number), which is added to every entry in the output matrix. So the data for the filter is the matrix A and the bias b

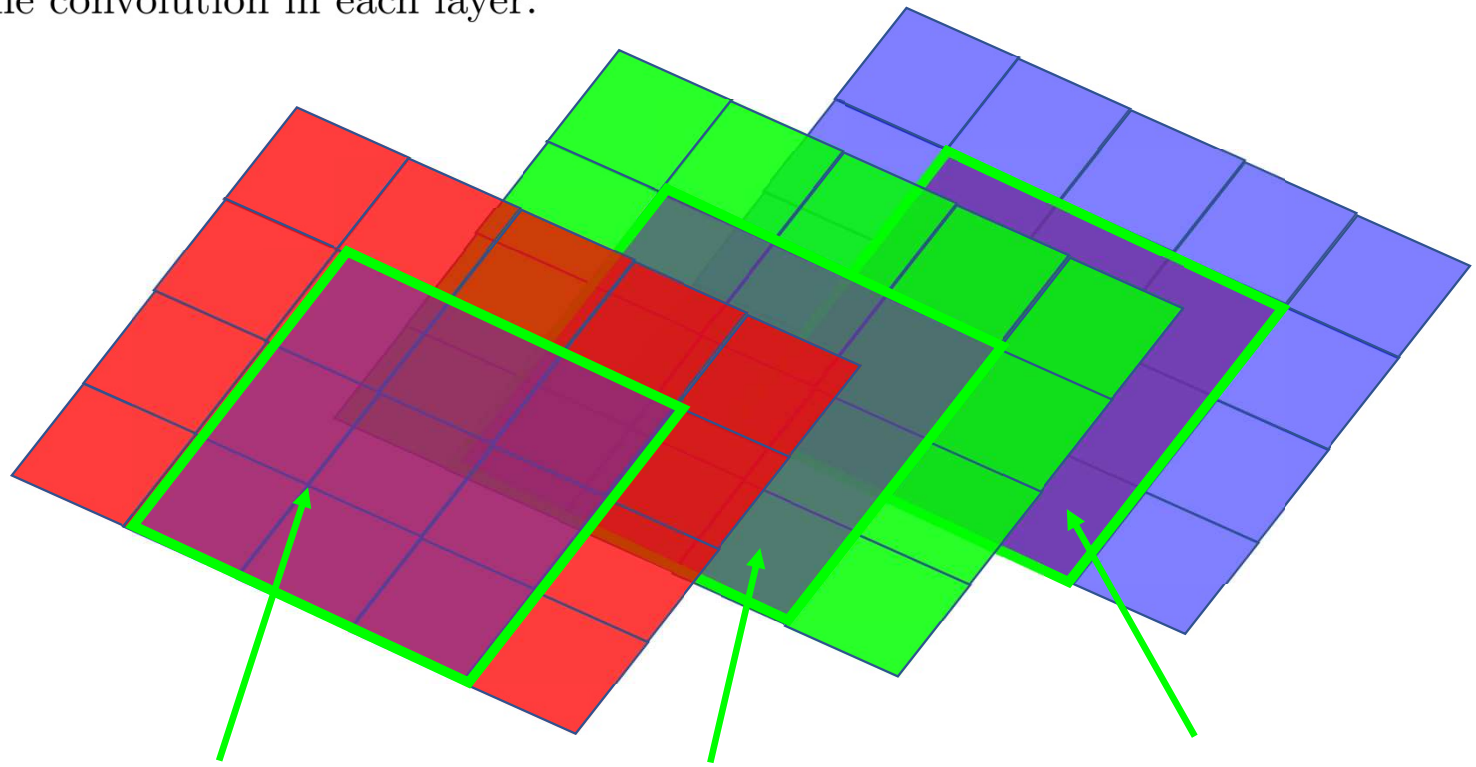
Often the convolution matrix will not fit precisely into the image. One can pad the image with 0's to make this process give a matrix of the same size as the original. In our case we would get a 2×3 output from the convolution with a 3×3 matrix.

The matrix we convolve with is known as a *filter* or a *kernel*. The output of the convolution is called a *feature map*. A single convolutional layer will ususally have several kernels and so will output several feature maps.



A convolutional layer will convolve with a number of kernels and hence will produce an array of feature maps. If the image has several layers, each layer will generate an array of feature maps.

When we are convolving a multi-layer feature map, the output is the sum of the convolution in each layer.



convolution with layer 1 + convolution with layer 2 convolution with layer 3

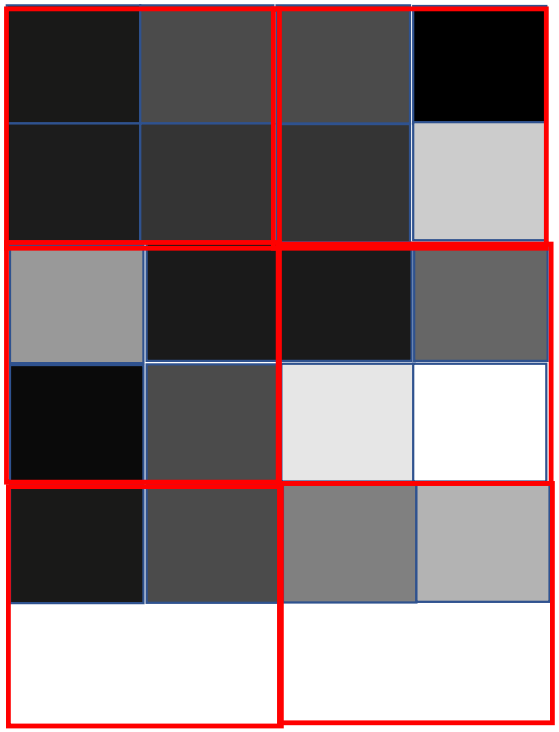
The process of moving the kernel matrix around is done automatically by the convolution layer in Pytorch.

Another type of layer we need is the MaxPool layer. This layer moves a rectangle around the feature map without overlaps. Instead of convolving, it simply takes the max of the values of the pixels within the rectangle.

There is also an Average Pooling layer which takes the average of the values within the rectangle

$$\begin{bmatrix} 0.3 & 0.8 \\ 0.6 & 0.9 \\ 0.3 & 0.7 \end{bmatrix}$$

MaxPool(2,2)



AvgPool(2,2)

$$\begin{bmatrix} 0.1875 & 0.325 \\ 0.251 & 0.6 \\ 0.1 & 0.3 \end{bmatrix}$$

We can use convolution layers to improve our model.

```
cnn_model = nn.Sequential(nn.Conv2d(1,32,3),
                          nn.ReLU(),
                          nn.MaxPool2d(2,2),
                          nn.Flatten(),
                          nn.Linear(5408,256),
                          nn.ReLU(),
                          nn.Linear(256,10))
```

The first layer takes in a tensor of dim = (batch_size,channels,28,28).

In our case we have a batch_size of 100 and we have 1 channel of gray scale values so the input is a 4-d tensor of dimension (100,1,28,28)

The Conv2d(1,32,3) outputs a feature map with 32 channels, it uses 3x3 grid for each filter, this gives 32 feature maps, each of dimension 26x26 so the output of the first layer has dimension (100,32,26,26)

The MaxPool2d(2,2) moves a 2x2 grid across each feature map and takes the max of the 4 values. This produces an output of dimension (100,32,13,13)

Now we want to send this into a linear layer but a linear layer needs batches of vectors i.e. tensors of dimension (100,input_dim) so we need to flatten the (32,13,13) into (32*13*13)=(5408)

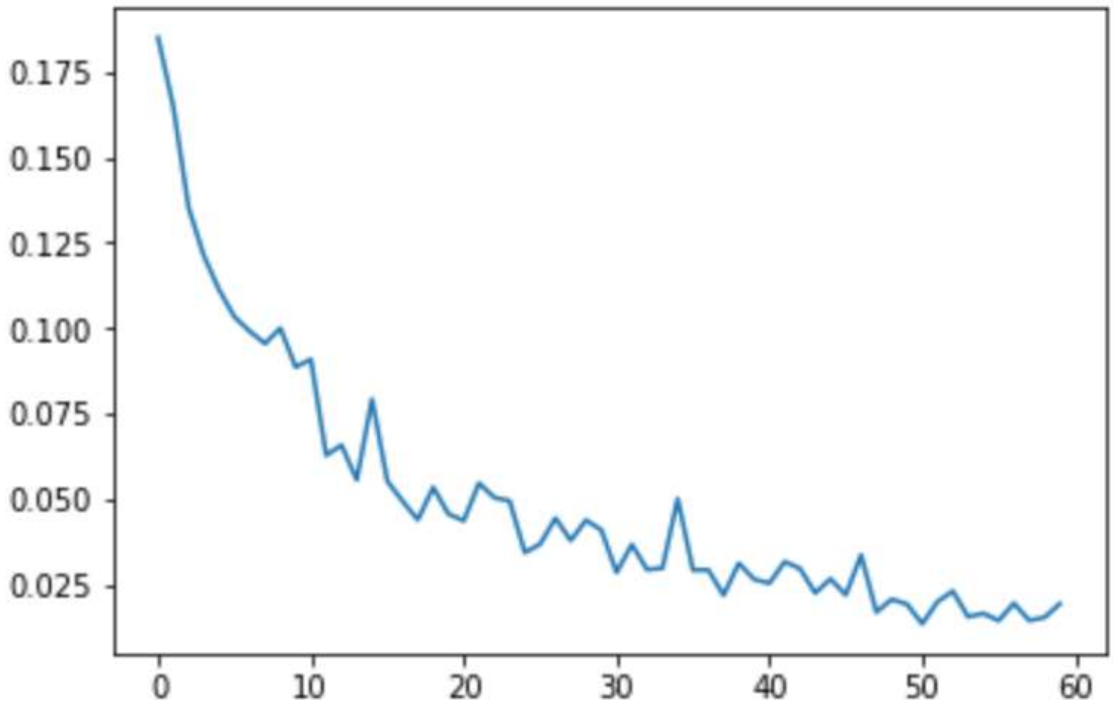
The training loop is identical to the previous except we do not need to flatten the image before we send it through the model

```
epochs = 10
losses = []
for epoch in range(epochs):
    ave_loss = 0.0
    for  i,(x,y) in enumerate(train_loader):

        x = x.cuda()
        y = y.cuda()
        output = cnn_model(x)

        optimizer.zero_grad()

        loss = criterion(output,y)

        ave_loss = ave_loss * 0.9 + loss.item() * 0.1

        loss.backward()

        optimizer.step()

        if (i+1) % 100 == 0 or (i+1) == len(train_loader):
            losses.append(ave_loss)
            print('==>>> epoch: %d, batch index: %d, train loss: %.6f' % (epoch, i+1, ave_loss))
```

No flattening here

```
1  plt.plot(losses);
```

executed in 113ms, finished 10:02:11 2019-06-04



We increase the accuracy on the test set by 43%

(misclassified under the old model = 243, misclassified under the new model = 137, improvement = 243-137/243 = 43%)

```
1  predictions = []
2  for i in range(len(test_set)):
3      z = cnn_model(test_set[i][0].reshape(1,1,28,28).cuda())
4      predictions.append(z.argmax().cpu().item())
```

executed in 10.9s, finished 10:02:28 2019-06-04

```
1  predictions = np.array(predictions)
```

executed in 4ms, finished 10:02:34 2019-06-04

```
1  len(predictions[predictions.T==test_set.test_labels.numpy()])/10000
```

executed in 5ms, finished 10:02:36 2019-06-04

```
0.9863
```

Using more convolutional layers we can improve the test accuracy

```
cnn_model_2 = nn.Sequential(nn.Conv2d(1,32,3),
                            nn.ReLU(),
                            nn.MaxPool2d(2,2),
                            nn.Conv2d(32,64,3),
                            nn.ReLU(),
                            nn.MaxPool2d(2,2),
                            nn.Conv2d(64,64,3),
                            nn.ReLU(),
                            Flatten(),
                            nn.Linear(576,64),
                            nn.ReLU(),
                            nn.Linear(64,10))
```

```
predictions = []
for i in range(len(test_set)):
    z = cnn_model_2(test_set[i][0].reshape(1,1,28,28).cuda())
    predictions.append(z.argmax().cpu().item())
```

executed in 15.4s, finished 10:09:25 2019-06-04

```
predictions = np.array(predictions)
len(predictions[predictions.T==test_set.test_labels.numpy()])/10000
```

executed in 6ms, finished 10:09:28 2019-06-04

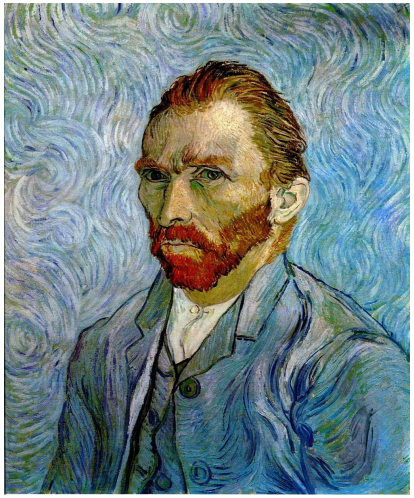
```
0.99
```

Accuracy improvement over the fully connected model:
243-100/243 = 59%

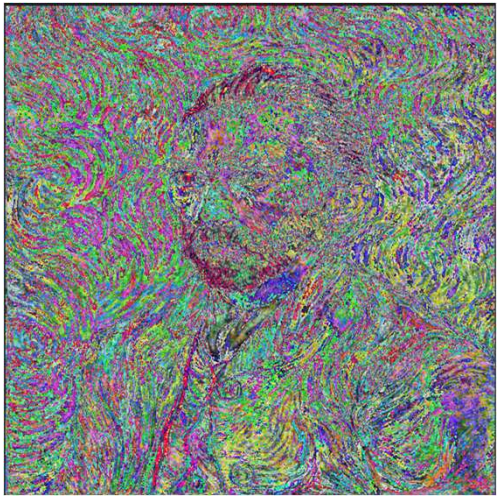
As a final application of convolutional neural networks we shall look at Neural Style Transfer.

This is a way to transfer a style from a painting or image to another image such that the generated image will contain the content but it will be in the style of the painting

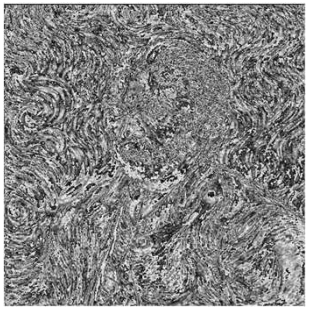
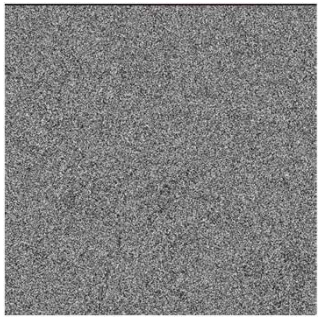
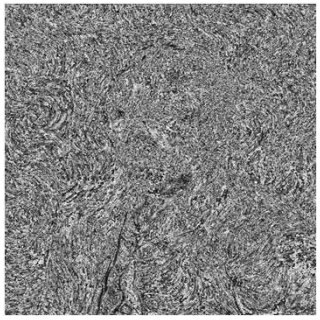
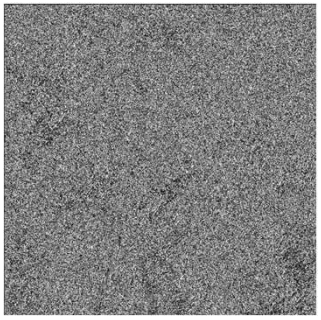
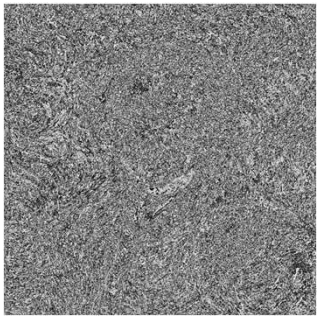
Style image



After normalization

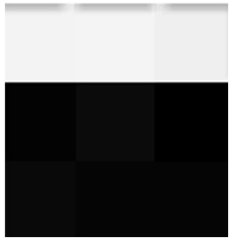


First 6 images (out of 64) of the feature map produced by the first conv layer


Some of the filters in conv_1

```
1 imshow(model.conv_1.weight[0,0])
```



```
[[-0.0535, -0.0493, -0.0679],
 [ 0.0153,  0.0451,  0.0021],
 [ 0.0362,  0.0200,  0.0199]]
```

```
1 imshow(model.conv_1.weight[0,1])
```

```
[[ 0.0170,  0.0554, -0.0062],
 [ 0.1416,  0.2271,  0.1376],
 [ 0.1200,  0.2003,  0.0921]]
```

```
1 imshow(model.conv_1.weight[1,2])
```

```
[[ 0.0673, -0.0954, -0.0380],
 [ 0.0620, -0.1313, -0.1069],
 [ 0.0481,  0.2300, -0.0306]]
```

```
1 imshow(model.conv_1.weight[34,2])
```

```
[[-0.0551, -0.0916, -0.0324],
 [-0.1101,  0.3656, -0.0735],
 [ 0.0272,  0.0712,  0.0137]]
```

We compute the covariance matrix of the images in the feature map by viewing each image as 600*600 samples of a random variable with mean = 0

```
1 first_feature_map = model.conv_1(model[0](style_img))
```

```
1 first_feature_map.size()
```

```
torch.Size([1, 64, 600, 600])
```

```
1 features = first_feature_map.reshape(1 * 64, 600 * 600)
```

```
1 features.size()
```

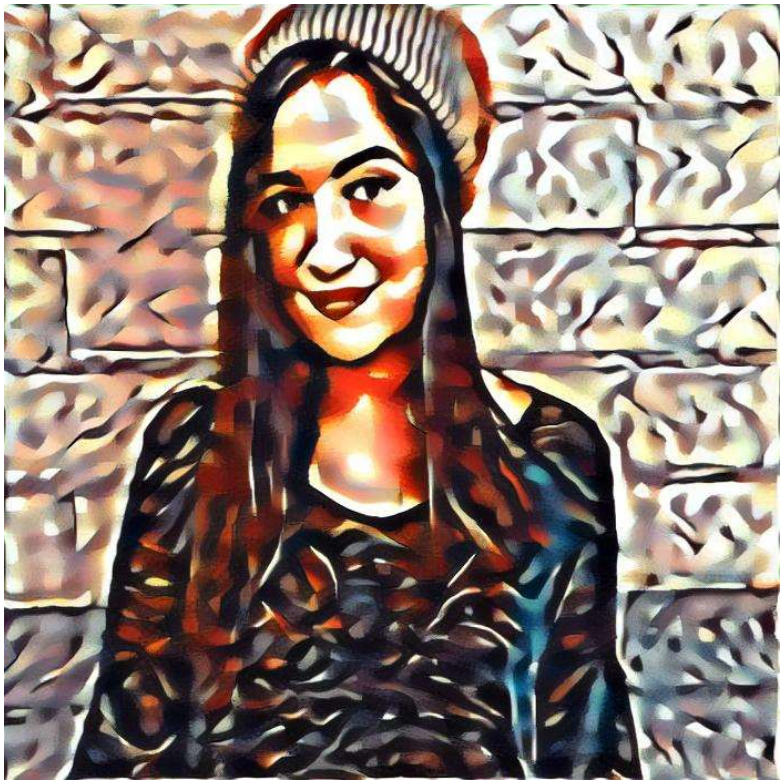
```
torch.Size([64, 360000])
```

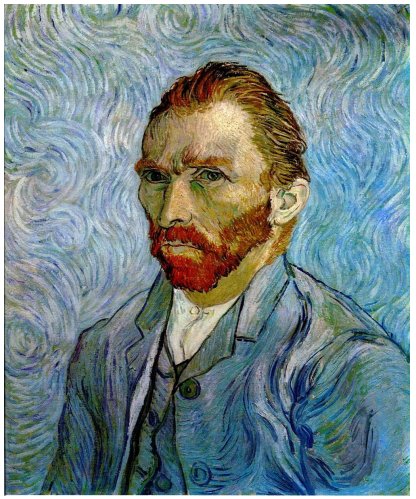
This is the data matrix and so the covariance matrix is estimated by the 64x64 matrix

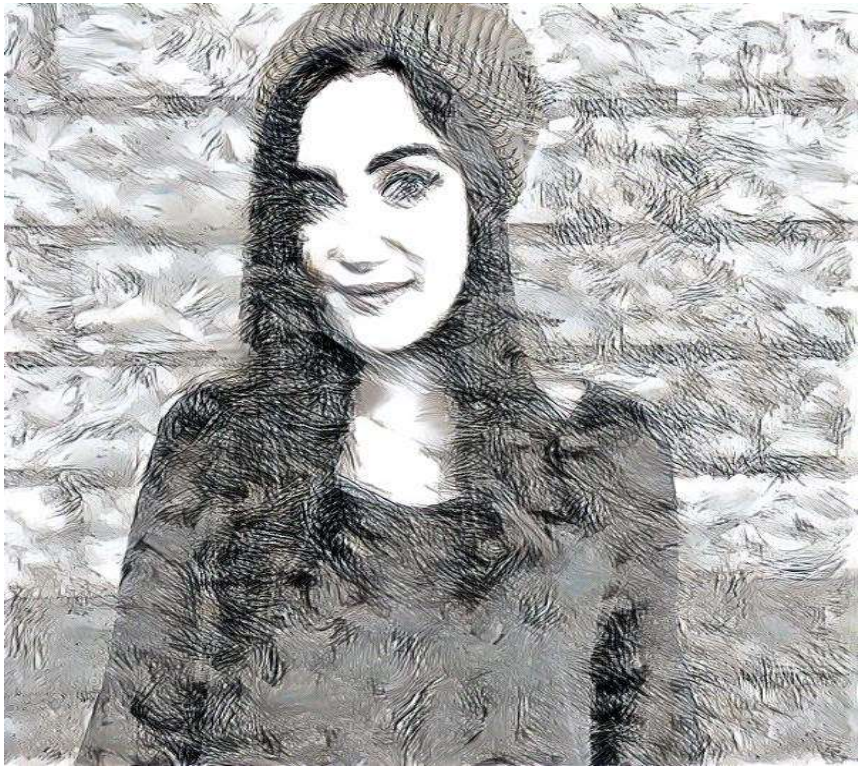
```
1 (features @ features.t())/360000.0
```

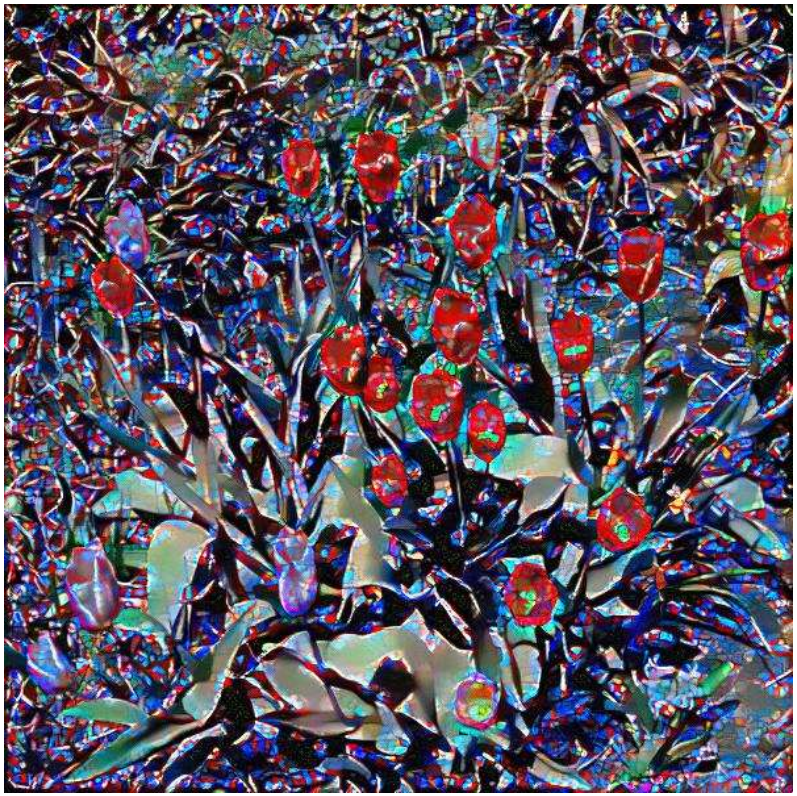
+

=









The idea behind this algorithm is based on the principle that in a multi-level conv network, the first convolutional layers pick up fine details in the image such as brush stroke textures and so on.

As we move up in the network the conv layers will output more and more global information and finally will be a representation of the entire image.

We use a pre-trained network which has been trained on thousands of images to be able to identify the content of an image.

There are many of these pre-trained networks available and mostly one will be able to use one of these for a specific project. These networks may have taken days to train on powerful machines but once they are trained we can use them on laptops. This is known as transfer learning

Here we are using a model known as VGG19

```
1  cnn = models.vgg19(pretrained=True).features.to(torch.device('cuda')).eval()
```

It is included with torchvision (as are numerous other models)

```
1  print(cnn)
```

executed in 5ms, finished 13:42:50 2019-06-04

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

It has 19 conv or maxpool layers.

To do a style transfer we send 3 images through the network:

A style_image, a content_image and an input_image. We try to adjust the input_image to match the convolutions in the lower levels of the style_image and match the convolution of the content_image at a higher level

We normalize our images so the pixel values have mean=0 and std=1. Remark that when we are using RGB images there is a channel for each of the Red, Green and Blue channels so an image is represented by a tensor of dim=(3,height,width).

The VGG19 model has channel means and channel stds

```
cnn_normalization_mean = torch.tensor([0.485,0.456,0.406]).cuda()
cnn_normalization_std = torch.tensor([0.229,0.224,0.225]).cuda()
```

We can of course match images pixel by pixel but for the style_image we instead match the covariance matrices of the feature maps.

Since the means are 0 we can compute covariance using the Gram matrix: $U * U^T$

```
def gram_matrix(input):
    a, b, c, d = input.size()

    features = input.reshape(a * b, c * d)

    G = torch.mm(features,features.t())

    return G.div( a * b * c * d)
```

For a convolution layer Conv2d(channels,h,w) the output will be a tensor of dimension (batch_size,channels,h,w) (in our case the batch_size=1) so the features matrix will be a channels x h*w matrix and the Gram matrix will be a channels x channels matrix.

We use the `style_layers_default = ['conv_1','conv_2','conv_3','conv_4','conv_5']` as the layers where we match Gram matrices for the output_image and the style_image

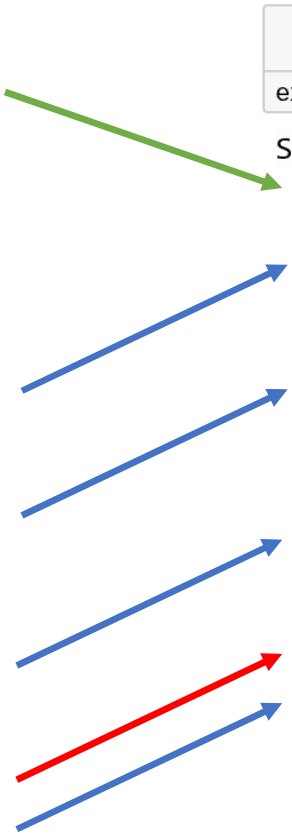
For the content we match the outputs at `content_layers_default = ['conv_4']` directly (sum of squares of the differences in pixel values)

We insert objects to compute these values in the model at the appropriate levels

```python
class ContentLoss(nn.Module):

    def __init__(self,target):
        super(ContentLoss,self).__init__()
        self.target = target.detach()

    def forward(self,input):
        self.loss = F.mse_loss(input,self.target)
        return input
```

```python
class StyleLoss(nn.Module):

    def __init__(self,target_feature):
        super(StyleLoss,self).__init__()
        self.target = gram_matrix(target_feature).detach()

    def forward(self,input):
        G = gram_matrix(input)
        self.loss = F.mse_loss(G,self.target)
        return input
```

```
content_layers_default = ['conv_4']
style_layers_default = ['conv_1','conv_2','conv_3','conv_4','conv_5']

def get_style_model_and_losses(cnn,normalization_mean,
                               normalization_std,
                               content_img,
                               style_img,
                               content_layers=content_layers_default,
                               style_layers=style_layers_default):

    cnn = copy.deepcopy(cnn)

    normalization=Normalization(normalization_mean,normalization_std).cuda()

    content_losses = []
    style_losses = []

    model = nn.Sequential(normalization)

    i = 0

    for layer in cnn.children():
        if isinstance(layer,nn.Conv2d):
            i += 1
            name = 'conv_{}'.format(i)
        elif isinstance(layer,nn.ReLU):
```



```
1  print(model)
```

executed in 7ms, finished 14:41:47 2019-06-04

```
Sequential(
  (0): Normalization()
  (conv_1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (style_loss_1): StyleLoss()
  (relu_1): ReLU()
  (conv_2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (style_loss_2): StyleLoss()
  (relu_2): ReLU()
  (pool_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv_3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (style_loss_3): StyleLoss()
  (relu_3): ReLU()
  (conv_4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (content_loss_4): ContentLoss()
  (style_loss_4): StyleLoss()
  (relu_4): ReLU()
  (pool_4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv_5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (style_loss_5): StyleLoss()
)
```

Remark that we are only using a small part of the full VGG19 model

The total loss is a weighted sum of the sum of the style losses and the content loss

```
for sl in style_losses:
    style_score += sl.loss
for cl in content_losses:
    content_score += cl.loss

style_score *= style_weight
content_score *= content_weight

loss = style_score + content_score
```

We can then minimize this loss function using an LBGFS optimizer

```
1  def get_input_optimizer(input_img):
2      optimizer = optim.LBFGS([input_img.requires_grad_()])
3      return optimizer
4
```

Remark that we are not training a model, it is already trained, but at each step we change the pixel values in the input_image to minimize the total loss