COMPUTER VISION

PLUS PYTORCH

ARIHARASUDHAN

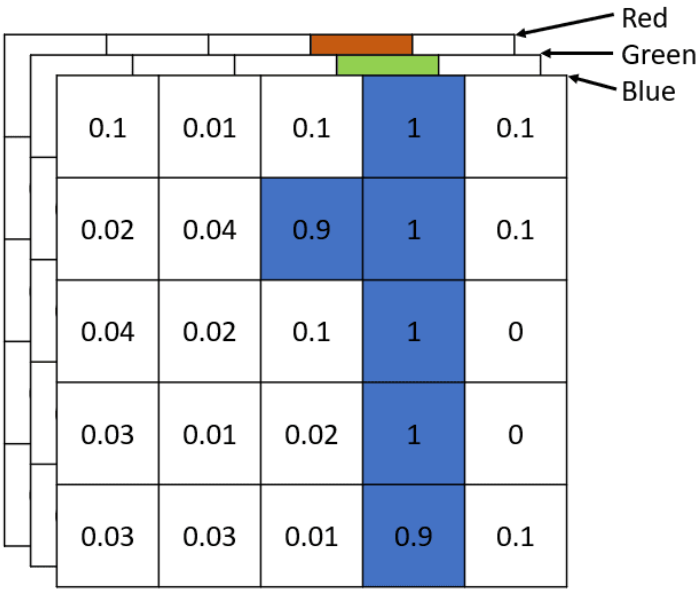
COMPUTER VISION

One of the successful branches of artificial intelligence is **computer vision**, which allows computer to gain some insights from digital images and/or video. **Neural networks** can be successfully used for computer vision tasks.

IMAGES AS TENSORS

Images consist of pixels. Some images may have multi-channels. If the given image is an RGB image, it would have three channels for representing RED-GREEN-BLUE.

0.1	0.01	0.1	1	0.1
0.02	0.04	0.9	1	0.1
0.04	0.02	0.1	1	0
0.03	0.01	0.02	1	0
0.03	0.03	0.01	0.9	0.1



We'll be playing with the MNIST dataset. So, let's load it and start playing. Let's visualize it using matplotlib.

In [4]:
```
#Import the packages needed.
import torch
import torchvision
import matplotlib.pyplot as plt
import numpy as np
```

In [5]:
```
from torchvision.transforms import ToTensor

data_train = torchvision.datasets.MNIST('./data',
        download=True,train=True,transform=ToTensor())
data_test = torchvision.datasets.MNIST('./data',
        download=True,train=False,transform=ToTensor())
```

In [6]:
```
fig,ax = plt.subplots(1,7)
for i in range(7):
    ax[i].imshow(data_train[i][0].view(28,28))
    ax[i].set_title(data_train[i][1])
    ax[i].axis('off')
```

The architecture of the MNIST dataset can be viewed as the following :

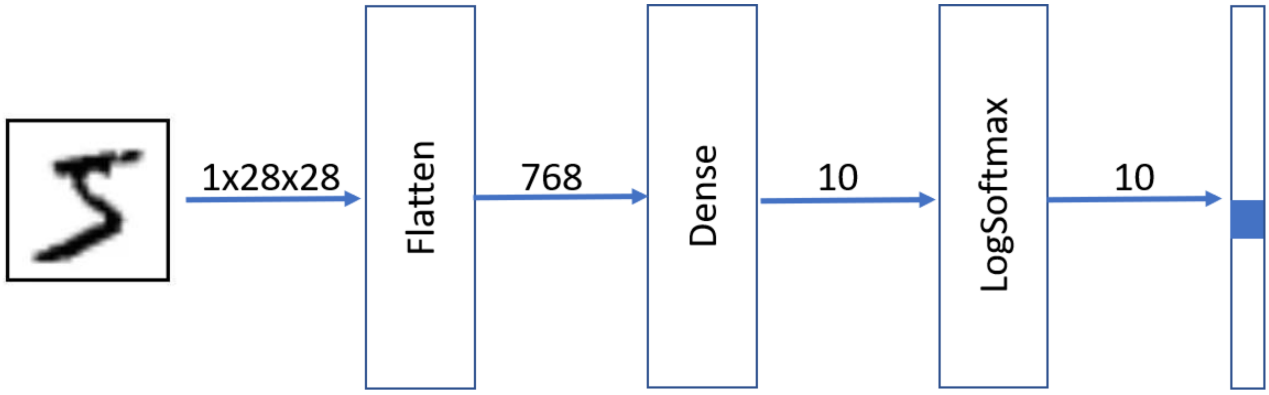
```
In [8]: print('Training samples:',len(data_train))
        print('Test samples:',len(data_test))

        print('Tensor size:',data_train[0][0].size())
        print('First 10 digits are:', [data_train[i][1] for i in range(10)])

        Training samples: 60000
        Test samples: 10000
        Tensor size: torch.Size([1, 28, 28])
        First 10 digits are: [5, 0, 4, 1, 9, 2, 1, 3, 1, 4]
```

Clearly, there are 60000 training samples and 10000 test samples. Each image is of the dimension 1x28x28 where 1 mentions the number of channels in an image. First 10 digits are 5,0,4,1,9,2,1,3,1 and 4. Each sample is a tuple. First element is the actual image of a digit, represented by a tensor of shape 1x28x28. Second element is a **label** that specifies which digit is represented by the tensor.

SIMPLE DENSE NET

A basic **neural network** in PyTorch consists of a number of **layers**. The simplest network would include just one fully-connected layer, which is called **Linear** layer, with 784 inputs (one input for each pixel of the input image) and 10 outputs (one output for each class). The dimension of our digit images is 1×28×28, i.e. each image contains 28×28=784 different pixels. Because linear layer expects its input as one-dimensional vector, we need to insert another layer into the network, called **Flatten**, to change input tensor shape from 1×28×28 to 784.

After Flatten, there is a main linear layer (called Dense in PyTorch terminology) that converts 784 inputs to 10 outputs-one per class. We want n-th output of the network to return the probability of the input digit being equal to n. Because the output of a fully-connected layer is not normalized to be between 0 and 1, it cannot be thought of as probability. Moreover, if want outputs to be probabilities of different digits, they all need to add up to 1. To turn output vectors into probability vector, a function called **Softmax** is often used as the last activation function.



Let’s use pytorchcv as a helper to load the data.

```
In [11]: !wget https://raw.githubusercontent.com/MicrosoftDocs/pytorchfundamentals/main/computer-vision-py

--2023-04-20 11:37:53--  https://raw.githubusercontent.com/MicrosoftDocs/pytorchfundamentals/main/computer-vision-pytorch/pytorchcv.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.111.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6540 (6.4K) [text/plain]
Saving to: ‘pytorchcv.py’

pytorchcv.py        100%[===================>]   6.39K  --.-KB/s    in 0s

2023-04-20 11:37:55 (30.8 MB/s) - ‘pytorchcv.py’ saved [6540/6540]
```

Torchinfo provides information complementary to what is provided by print(your_model) in PyTorch, similar to Tensorflow's model.summary(). Let’s insatll that too using pip command. Now, let’s import all the necessary modules.

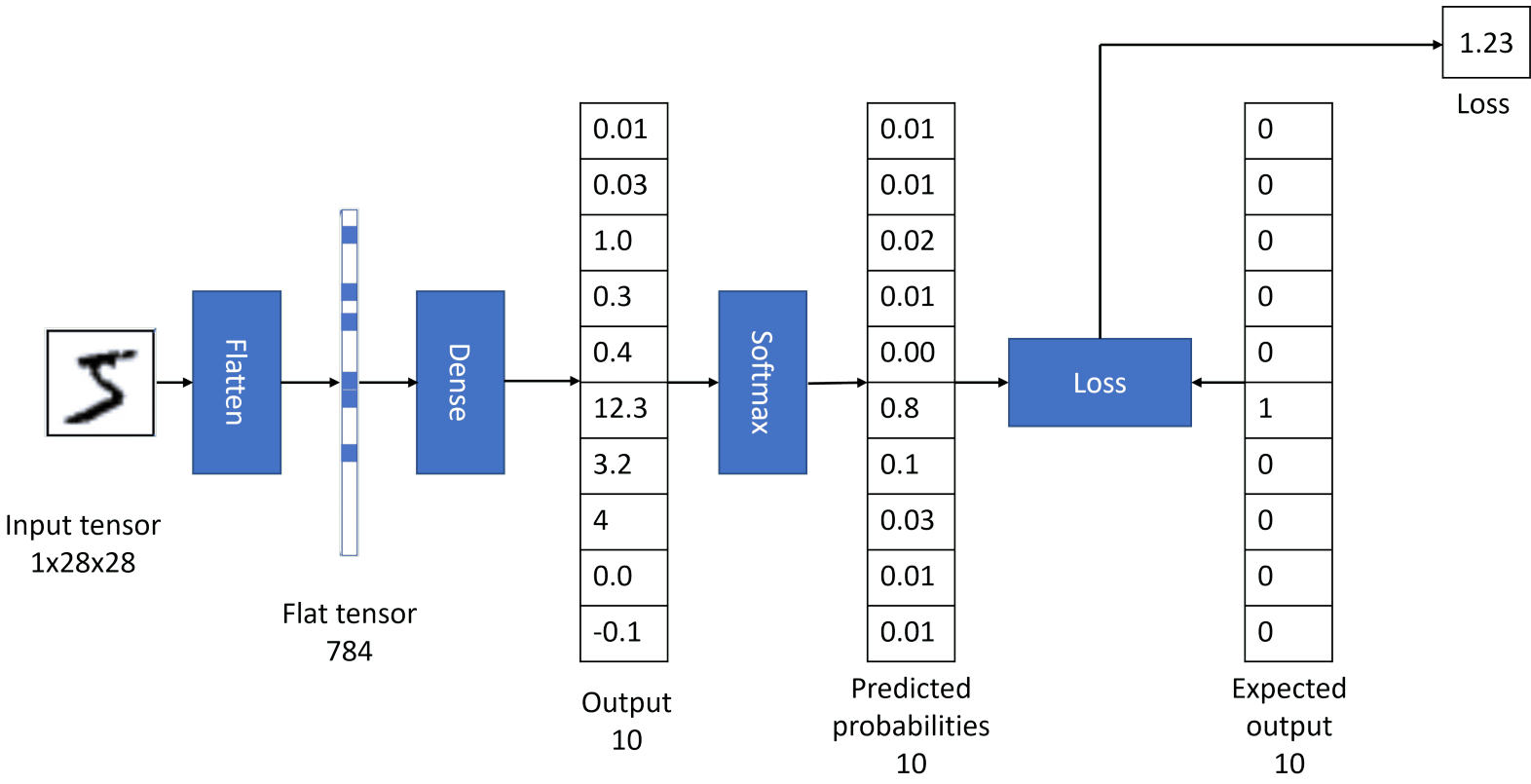
Let’s import and load the mnist dataset as shown below.

```
In [15]: import torch
         import torch.nn as nn
         import torchvision
         import matplotlib.pyplot as plt
         from torchinfo import summary
         from pytorchcv import load_mnist, plot_results
         load_mnist()
```

Now,let’s create our DenseNet.

```
In [16]: net = nn.Sequential(
                nn.Flatten(),
                nn.Linear(784,10), # 784 inputs, 10 outputs
                nn.LogSoftmax())
```

What happens after each layers is illustrated below.



After loading the data, let's define a train function of one epoch as shown below:

In [20]:
```
train_loader = torch.utils.data.DataLoader(data_train,batch_size=64)
test_loader = torch.utils.data.DataLoader(data_test,batch_size=64)
```

In [22]:
```
def train_epoch(net,dataloader,lr=0.01,optimizer=None,loss_fn = nn.NLLLoss()):
    optimizer = optimizer or torch.optim.Adam(net.parameters(),lr=lr)
    net.train()
    total_loss,acc,count = 0,0,0
    for features,labels in dataloader:
        optimizer.zero_grad()
        out = net(features)
        loss = loss_fn(out,labels) #cross_entropy(out,labels)
        loss.backward()
        optimizer.step()
        total_loss+=loss
        _,predicted = torch.max(out,1)
        acc+=(predicted==labels).sum()
        count+=len(labels)
    return total_loss.item()/count, acc.item()/count
train_epoch(net,train_loader)
```

```
/home/ari-pt7127/anaconda3/lib/python3.9/site-packages/torch/nn/modules/container.py:217: UserWarning: Implicit dimension cho
ice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
```

And, let's define a function for validation.

In [25]:
```
def validate(net, dataloader,loss_fn=nn.NLLLoss()):
    net.eval()
    count,acc,loss = 0,0,0
    with torch.no_grad():
        for features,labels in dataloader:
            out = net(features)
            loss += loss_fn(out,labels)
            pred = torch.max(out,1)[1]
            acc += (pred==labels).sum()
            count += len(labels)
    return loss.item()/count, acc.item()/count
validate(net,test_loader)
```

```
(0.005866207122802734, 0.8941)
```

Normally when training a neural network, we train the model for several epochs observing training and validation accuracy. In the beginning, both training and validation accuracy should increase, as the network picks up the patterns in the dataset. However, at some point it can happen that training accuracy increases while validation accuracy starts to decrease. That would be an indication of **overfitting**, that is model does well on your training dataset, but not on new data. The following function does both the training and validation.

Function for both training and validation :

```
In [*]: def train(net,train_loader,test_loader,optimizer=None,lr=0.01,epochs=10,loss_fn=nn.NLLLoss()):
            optimizer = optimizer or torch.optim.Adam(net.parameters(),lr=lr)
            res = { 'train_loss' : [], 'train_acc': [], 'val_loss': [], 'val_acc': []}
            for ep in range(epochs):
                tl,ta = train_epoch(net,train_loader,optimizer=optimizer,lr=lr,loss_fn=loss_fn)
                vl,va = validate(net,test_loader,loss_fn=loss_fn)
                print(f"Epoch {ep:2}, Train acc={ta:.3f}, Val acc={va:.3f},
                      Train loss={tl:.3f}, Val loss={vl:.3f}")
                res['train_loss'].append(tl)
                res['train_acc'].append(ta)
                res['val_loss'].append(vl)
                res['val_acc'].append(va)
            return res
        net = nn.Sequential(
                nn.Flatten(),
                nn.Linear(784,10), # 784 inputs, 10 outputs
                nn.LogSoftmax())
        hist = train(net,train_loader,test_loader,epochs=5)
```

And, here we get the losses for each epochs.

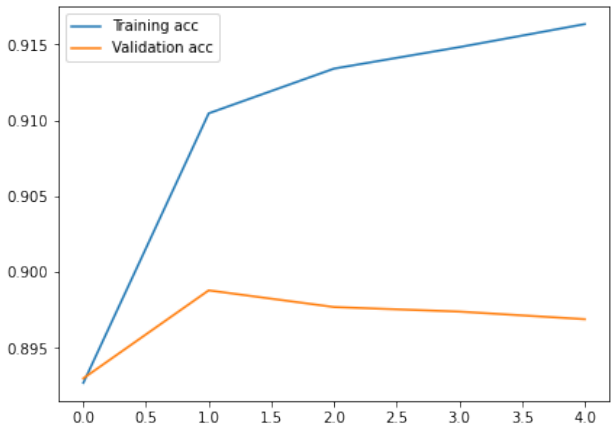
```
Epoch  0, Train acc=0.893, Val acc=0.893,Train loss=0.006, Val loss=0.006
Epoch  1, Train acc=0.910, Val acc=0.899,Train loss=0.005, Val loss=0.006
Epoch  2, Train acc=0.913, Val acc=0.898,Train loss=0.005, Val loss=0.006
Epoch  3, Train acc=0.915, Val acc=0.897,Train loss=0.005, Val loss=0.006
Epoch  4, Train acc=0.916, Val acc=0.897,Train loss=0.005, Val loss=0.006
```

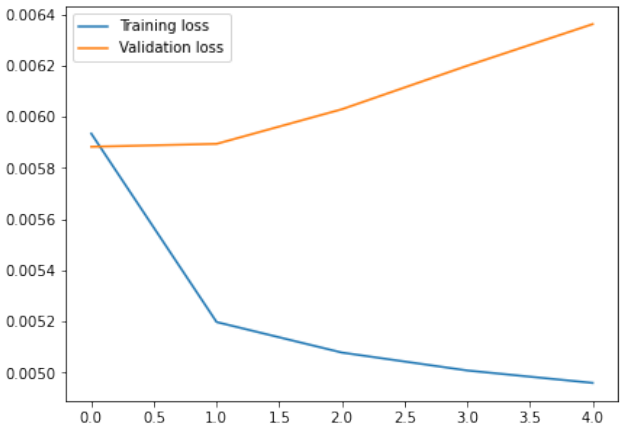
Now, we have the changess in losses in the variable hist. Let’s plot the histogram.

Use matplotlib for plotting.

```
plt.figure(figsize=(15,5))
plt.subplot(121)
plt.plot(hist['train_acc'], label='Training acc')
plt.plot(hist['val_acc'], label='Validation acc')
plt.legend()
plt.subplot(122)
plt.plot(hist['train_loss'], label='Training loss')
plt.plot(hist['val_loss'], label='Validation loss')
plt.legend()
```

The progress can be monitored.





THE MLP

To increase the accuracy, we can add some more hidden layers.

```
net = nn.Sequential(
        nn.Flatten(),
        nn.Linear(784,100),     # 784 inputs, 100 outputs
        nn.ReLU(),              # Activation Function
        nn.Linear(100,10),      # 100 inputs, 10 outputs
        nn.LogSoftmax(dim=0))
```

Let’s get a summary of the neural network we’ve just created.

```
summary(net,input_size=(1,28,28))
```

```
/home/ari-pt7127/anaconda3/lib/python3.9/site-packages/torchinfo/torchinfo.py:477: UserWarning: TypedStorage is deprecated. I
t will be removed in the future and UntypedStorage will be the only storage class. This should only matter to you if you are
using storages directly.  To access UntypedStorage directly, use tensor.untyped_storage() instead of tensor.storage()
  action_fn=lambda data: sys.getsizeof(data.storage()),
/home/ari-pt7127/anaconda3/lib/python3.9/site-packages/torch/storage.py:665: UserWarning: TypedStorage is deprecated. It will
be removed in the future and UntypedStorage will be the only storage class. This should only matter to you if you are using s
torages directly.  To access UntypedStorage directly, use tensor.untyped_storage() instead of tensor.storage()
  return super().__sizeof__() + self.nbytes()


==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 10]                   --
├─Flatten: 1-1                           [1, 784]                  --
├─Linear: 1-2                            [1, 100]                  78,500
├─ReLU: 1-3                              [1, 100]                  --
├─Linear: 1-4                            [1, 10]                   1,010
├─LogSoftmax: 1-5                        [1, 10]                   --
==========================================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
Total mult-adds (M): 0.08
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.32
Estimated Total Size (MB): 0.32
==========================================================================================
```

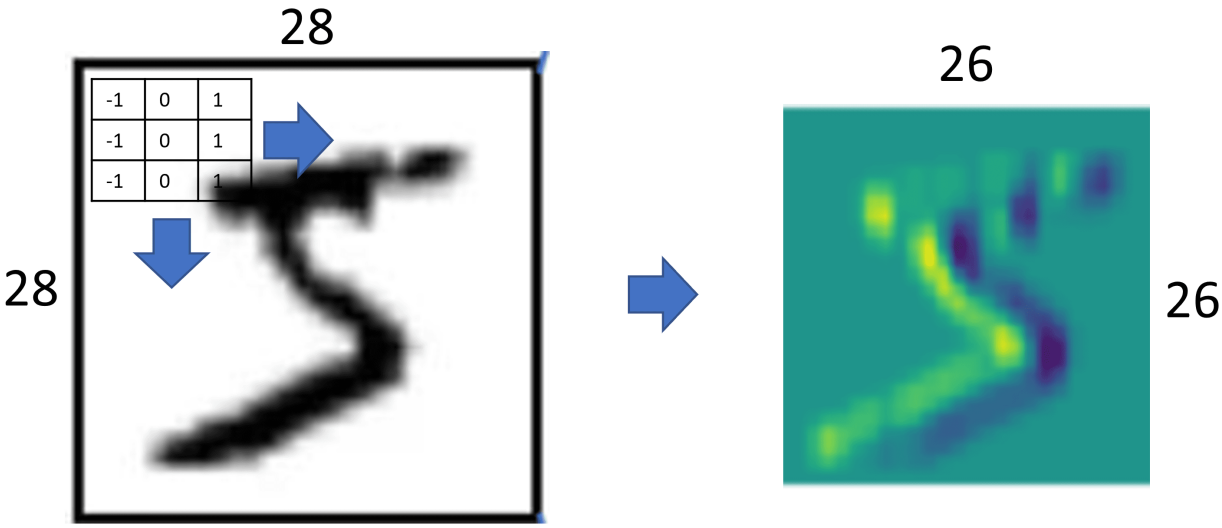
There is another syntax that we can use to define the same network by using classes :

```
from torch.nn.functional import relu, log_softmax
class MyNet(nn.Module):
    def __init__(self):
        super(MyNet, self).__init__()
        self.flatten = nn.Flatten()
        self.hidden = nn.Linear(784,100)
        self.out = nn.Linear(100,10)
```

```
    def forward(self, x):
        x = self.flatten(x)
        x = self.hidden(x)
        x = relu(x)
        x = self.out(x)
        x = log_softmax(x,dim=0)
        return x
net = MyNet()
summary(net,input_size=(1,28,28),device='cpu')
```

THE CNN

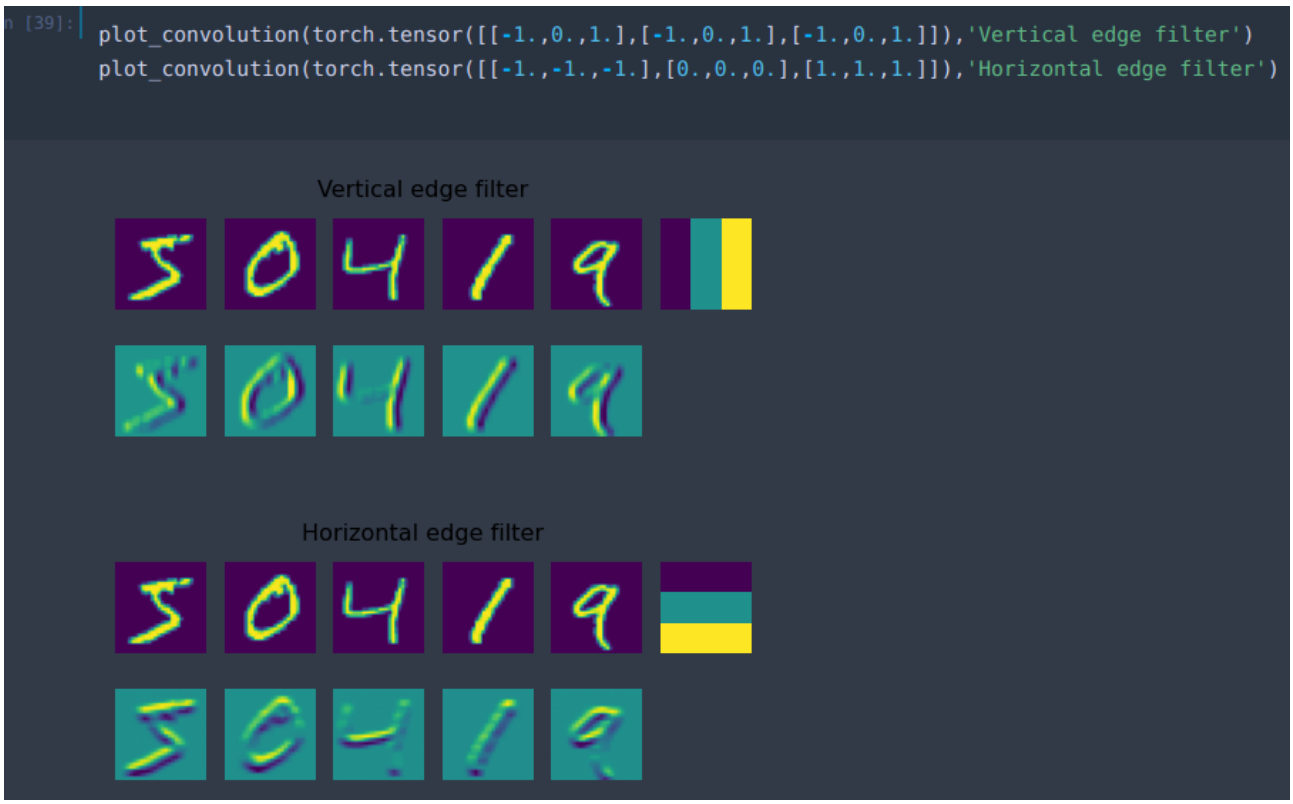
In deep learning, a convolutional neural network (CNN) is a class of artificial neural network most commonly applied to analyze visual imagery. We can use kernels to extract patterns from an image.



Convolutional layers are defined using nn.Conv2d construction. We need to specify the following:

- in_channels - number of input channels. In our case we are dealing with a grayscale image, thus number of input channels is 1. Color image has 3 channels (RGB).
- out_channels - number of filters to use. We will use 9 different filters, which will give the network plenty of opportunities to explore which filters work best for our scenario.
- kernel_size is the size of the sliding window.

```
[39]: plot_convolution(torch.tensor([[-1.,0.,1.],[-1.,0.,1.],[-1.,0.,1.]]),'Vertical edge filter')
      plot_convolution(torch.tensor([[-1.,-1.,-1.],[0.,0.,0.],[1.,1.,1.]]),'Horizontal edge filter')
```



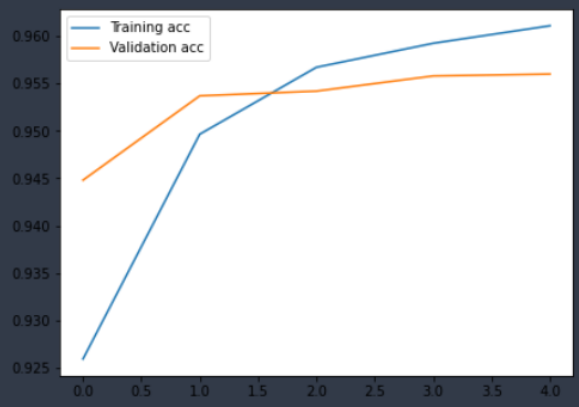
Let’s define it using the class syntax.

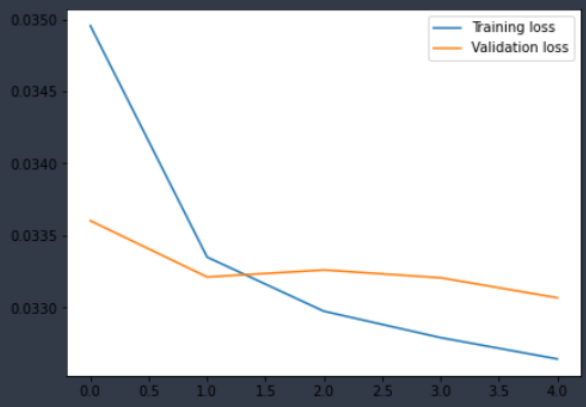
```
In [ ]: class OneConv(nn.Module):
            def __init__(self):
                super(OneConv, self).__init__()
                self.conv = nn.Conv2d(in_channels=1,out_channels=9,kernel_size=(5,5))
                self.flatten = nn.Flatten()
                self.fc = nn.Linear(5184,10)

            def forward(self, x):
                x = nn.functional.relu(self.conv(x))
                x = self.flatten(x)
                x = nn.functional.log_softmax(self.fc(x),dim=1)
                return x

        net = OneConv()

        summary(net,input_size=(1,1,28,28))
```

Now, let’s train the network using the already defined functions.

```
In [42]: hist = train(net,train_loader,test_loader,epochs=5)
         plot_results(hist)
```

```
Epoch  0, Train acc=0.926, Val acc=0.945, Train loss=0.035, Val loss=0.034
Epoch  1, Train acc=0.950, Val acc=0.954, Train loss=0.033, Val loss=0.033
Epoch  2, Train acc=0.957, Val acc=0.954, Train loss=0.033, Val loss=0.033
Epoch  3, Train acc=0.959, Val acc=0.956, Train loss=0.033, Val loss=0.033
Epoch  4, Train acc=0.961, Val acc=0.956, Train loss=0.033, Val loss=0.033
```





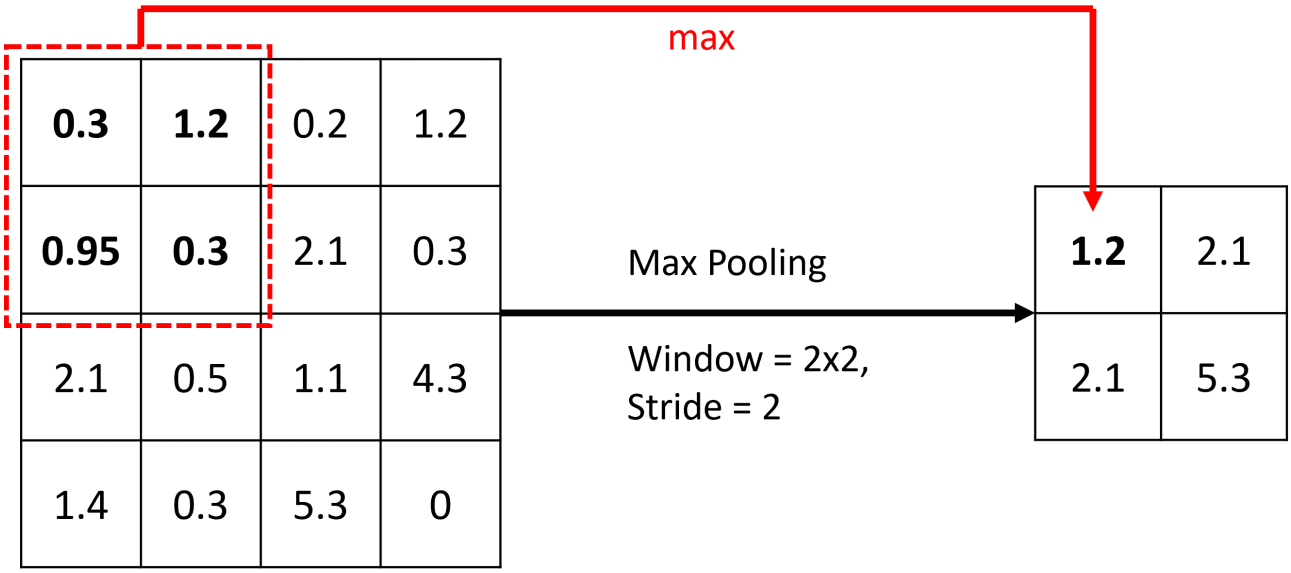
MULTILAYERED CNN

We can use the same idea of convolution to extract higher-level patterns in the image. For example, rounded edges of digits such as 8 and 9 can be composed from a number of smaller strokes.

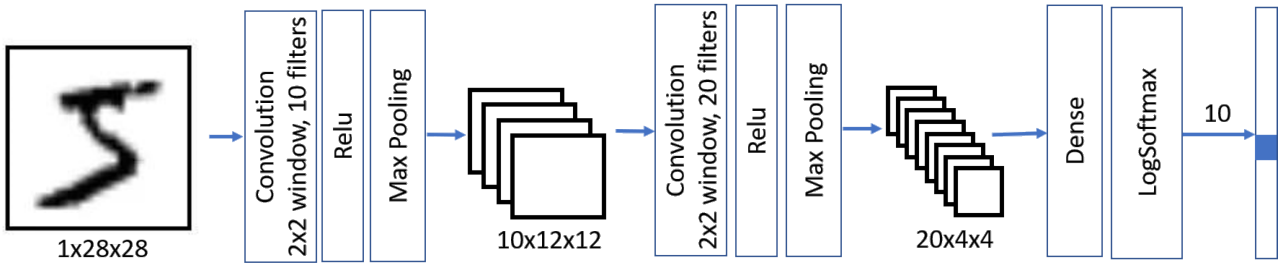
POOLING LAYERS

To recognize those patterns, we can build another layer of convolution filters on top of the result of the first layer.First convolutional layers looks for primitive patterns, such as horizontal or vertical lines. Next level of convolutional layers on top of them look for higher-level patterns, such as primitive shapes. More convolutional layers can combine those shapes into some parts of the picture, up to the final object that we are trying to classify. This creates a hierarchy of extracted patterns. When doing so, we also need to apply one trick: reducing the spatial size of the image.

Once we have detected there is a horizontal stoke within a sliding window, it is not so important at which exact pixel it occurred. Thus we can "scale down" the size of the image, which is done using one of the **pooling layers**:



We usually use multiple kernels in a layer just for selecting the best one.



Now, let's implement the multilayered CNN.

```
In [46]: class MultiLayerCNN(nn.Module):
             def __init__(self):
                 super(MultiLayerCNN, self).__init__()
                 self.conv1 = nn.Conv2d(1, 10, 5)
                 self.pool = nn.MaxPool2d(2, 2)
                 self.conv2 = nn.Conv2d(10, 20, 5)
                 self.fc = nn.Linear(320,10)

             def forward(self, x):
                 x = self.pool(nn.functional.relu(self.conv1(x)))
                 x = self.pool(nn.functional.relu(self.conv2(x)))
                 x = x.view(-1, 320)
                 x = nn.functional.log_softmax(self.fc(x),dim=1)
                 return x

         net = MultiLayerCNN()
         summary(net,input_size=(1,1,28,28))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MultiLayerCNN                            [1, 10]                   --
├─Conv2d: 1-1                            [1, 10, 24, 24]           260
├─MaxPool2d: 1-2                         [1, 10, 12, 12]           --
├─Conv2d: 1-3                            [1, 20, 8, 8]             5,020
├─MaxPool2d: 1-4                         [1, 20, 4, 4]             --
├─Linear: 1-5                            [1, 10]                   3,210
==========================================================================================
Total params: 8,490
Trainable params: 8,490
Non-trainable params: 0
Total mult-adds (M): 0.47
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.03
Estimated Total Size (MB): 0.09
==========================================================================================
```

And, let’s train it indeed.

```
hist = train(net,train_loader,test_loader,epochs=5)
```

```
Epoch  0, Train acc=0.954, Val acc=0.964, Train loss=0.002, Val loss=0.002
Epoch  1, Train acc=0.978, Val acc=0.979, Train loss=0.001, Val loss=0.001
Epoch  2, Train acc=0.981, Val acc=0.983, Train loss=0.001, Val loss=0.001
Epoch  3, Train acc=0.983, Val acc=0.983, Train loss=0.001, Val loss=0.001
Epoch  4, Train acc=0.984, Val acc=0.977, Train loss=0.001, Val loss=0.001
```

TRANSFER LEARNING

Training CNNs can take a lot of time, and a lot of data is required for that task. However, much of the time is spent to learn the best low-level filters that a network is using to extract patterns from images. A natural question arises - can we use a neural network trained on one dataset and adapt it to classifying different images without full training process?

This approach is called **transfer learning**, because we transfer some knowledge from one neural network model to another. In transfer learning, we typically start with a pre-trained model, which has been trained on some large image dataset, such as **ImageNet**. Those models can already do a good job extracting different features from generic images, and in many cases just building a classifier on top of those extracted features can yield a good result.

In [49]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from torchinfo import summary
import numpy as np
import os
from pytorchcv import train, plot_results, display_dataset, train_long, check_image_dir
```

Let’s download the Cat vs. Dogs dataset from kaggle and extract them all. check_image_dir goes over the whole dataset image by image, tries to load the image and check if it can be loaded correctly. All corrupt images are deleted.

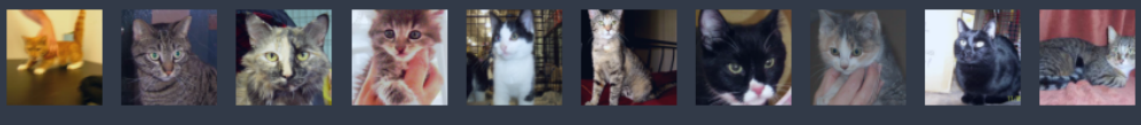
```
In [51]: if not os.path.exists('data/kagglecatsanddogs_5340.zip'):
             !wget -P data -q https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA7

In [52]: import zipfile
         if not os.path.exists('data/PetImages'):
             with zipfile.ZipFile('data/kagglecatsanddogs_5340.zip', 'r') as zip_ref:
                 zip_ref.extractall('data')

In [53]: check_image_dir('data/PetImages/Cat/*.jpg')
         check_image_dir('data/PetImages/Dog/*.jpg')

          Corrupt image: data/PetImages/Cat/666.jpg
          Corrupt image: data/PetImages/Dog/11702.jpg

          /home/ari-pt7127/anaconda3/lib/python3.9/site-packages/PIL/TiffImagePlugin.py:819: UserWarning: Truncated File Read
            warnings.warn(str(msg))
```

Next, let's load the images into PyTorch dataset, converting them to tensors and doing some normalization.

We define image transformation pipeline by composing several primitive transformations using Compose:

- Resize resizes our image to 256x256 dimensions
- CenterCrop gets the central part of the image with size 224x224. Pre-trained VGG network has been trained on 224x224 images, thus we need to bring our dataset to this size.
- ToTensor normalizes pixel intensities to be in 0..1 range, and convert images to PyTorch tensors
- std_normalize transform is additional normalization step specific for VGG

```
In [54]: std_normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])
         trans = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 std_normalize])
         dataset = torchvision.datasets.ImageFolder('data/PetImages',transform=trans)
         trainset, testset = torch.utils.data.random_split(dataset,[20000,len(dataset)-20000])

         display_dataset(dataset)
```



Now, we are ready with the Dataset. But, for transfer learning, we need an existing trained model. Let’s get it right now. It’s the so-called VGG.

```
In [*]: !mkdir -p models
        !wget -P models https://github.com/MicrosoftDocs/pytorchfundamentals/raw/main/computer-vision-pyt

--2023-04-20 12:46:45--  https://github.com/MicrosoftDocs/pytorchfundamentals/raw/main/computer-vision-pytorch/vgg16-397923af.pth
Resolving github.com (github.com)... 20.207.73.82
Connecting to github.com (github.com)|20.207.73.82|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://media.githubusercontent.com/media/MicrosoftDocs/pytorchfundamentals/main/computer-vision-pytorch/vgg16-397923af.pth [following]
--2023-04-20 12:46:45--  https://media.githubusercontent.com/media/MicrosoftDocs/pytorchfundamentals/main/computer-vision-pytorch/vgg16-397923af.pth
Resolving media.githubusercontent.com (media.githubusercontent.com)... 185.199.111.133, 185.199.108.133, 185.199.109.133, ...
Connecting to media.githubusercontent.com (media.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 553433881 (528M) [application/octet-stream]
Saving to: ‘models/vgg16-397923af.pth’

vgg16-397923af.pth   61%[==========>        ] 323.04M  4.01MB/s    eta 31s
```

Next, we'll load the weights into the pre-trained VGG-16 model by using the load_state_dict method. Then, use the eval method to set the model to inference mode.

```
In [56]: file_path = 'models/vgg16-397923af.pth'

         vgg = torchvision.models.vgg16()
         vgg.load_state_dict(torch.load(file_path))
         vgg.eval()

         sample_image = dataset[0][0].unsqueeze(0)
         res = vgg(sample_image)
         print(res[0].argmax())

          tensor(282)
```

Now let's try to see if those features can be used to classify images. Let's manually take some portion of images (800 in our case), and pre-compute their feature vectors.

We will store the result in one big tensor called feature_tensor, and also labels into label_tensor:

In [*]:
```
bs = 8
dl = torch.utils.data.DataLoader(dataset,batch_size=bs,shuffle=True)
num = bs*100
feature_tensor = torch.zeros(num,512*7*7)
label_tensor = torch.zeros(num)
i = 0
for x,l in dl:
    with torch.no_grad():
        f = vgg.features(x)
        feature_tensor[i:i+bs] = f.view(bs,-1)
        label_tensor[i:i+bs] = l
        i+=bs
        print('.',end='')
        if i>=num:
            break
```

```
................................................................................
```

Now, let’s train our network.

In [61]:
```
vgg_dataset = torch.utils.data.TensorDataset(feature_tensor,label_tensor.to(torch.long))
train_ds, test_ds = torch.utils.data.random_split(vgg_dataset,[700,100])
train_loader = torch.utils.data.DataLoader(train_ds,batch_size=32)
test_loader = torch.utils.data.DataLoader(test_ds,batch_size=32)
net = torch.nn.Sequential(torch.nn.Linear(512*7*7,2),torch.nn.LogSoftmax())
history = train(net,train_loader,test_loader)
```

```
Epoch  0, Train acc=0.916, Val acc=0.950, Train loss=0.068, Val loss=0.082
Epoch  1, Train acc=0.981, Val acc=0.970, Train loss=0.021, Val loss=0.041
Epoch  2, Train acc=0.983, Val acc=0.970, Train loss=0.011, Val loss=0.038
Epoch  3, Train acc=0.990, Val acc=0.960, Train loss=0.011, Val loss=0.093
Epoch  4, Train acc=1.000, Val acc=0.970, Train loss=0.000, Val loss=0.043
Epoch  5, Train acc=0.999, Val acc=0.960, Train loss=0.000, Val loss=0.065
Epoch  6, Train acc=1.000, Val acc=0.960, Train loss=0.000, Val loss=0.095
Epoch  7, Train acc=1.000, Val acc=0.960, Train loss=0.000, Val loss=0.098
Epoch  8, Train acc=1.000, Val acc=0.960, Train loss=0.000, Val loss=0.099
Epoch  9, Train acc=1.000, Val acc=0.960, Train loss=0.000, Val loss=0.099
```

Save it and load it whenever you need it.

```
In [62]: torch.save(vgg,'data/cats_dogs.pth')
```

There is the so-called mobilenet which can also be used for transfer learning.

SOURCES : MICROSOFT LEARNING