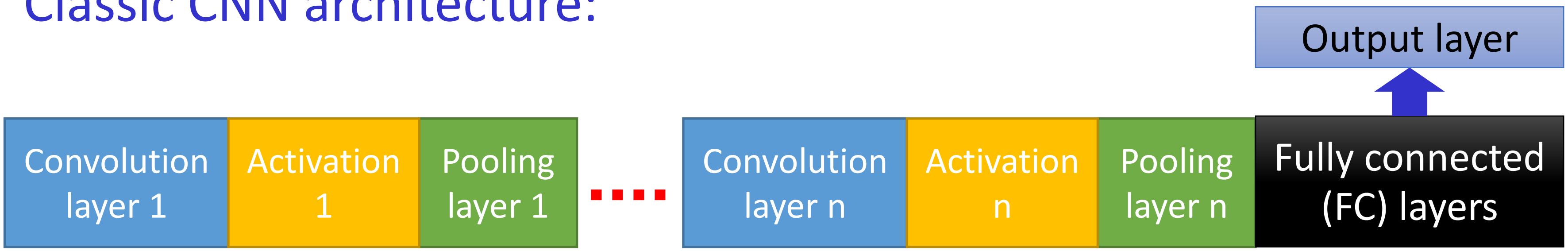
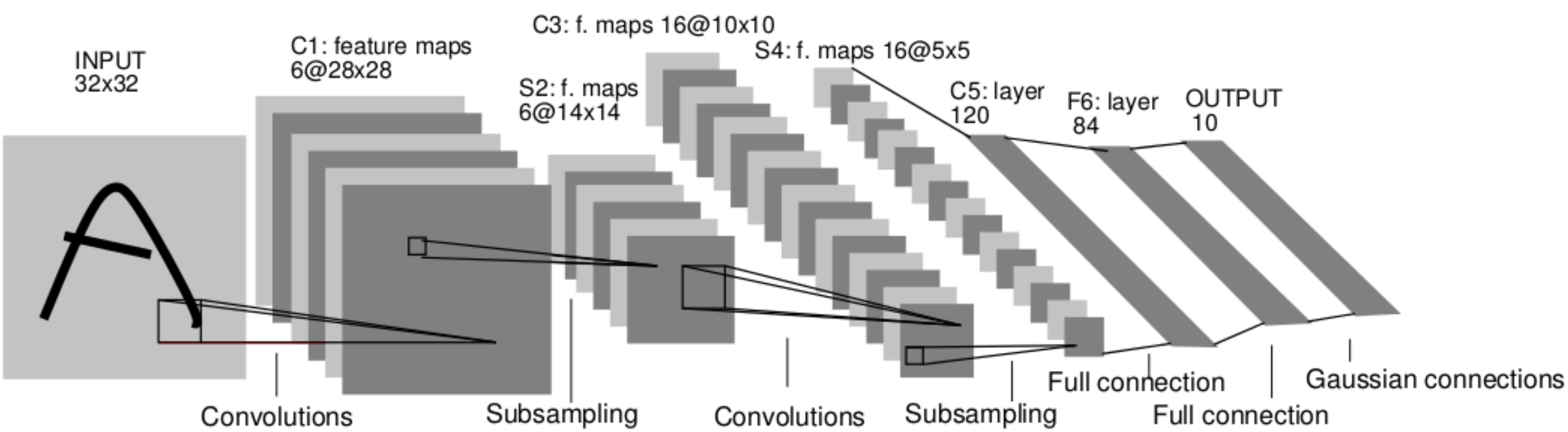
Tutorial on Pytorch

Part 2

Review: CNN architecture

Classic CNN architecture:





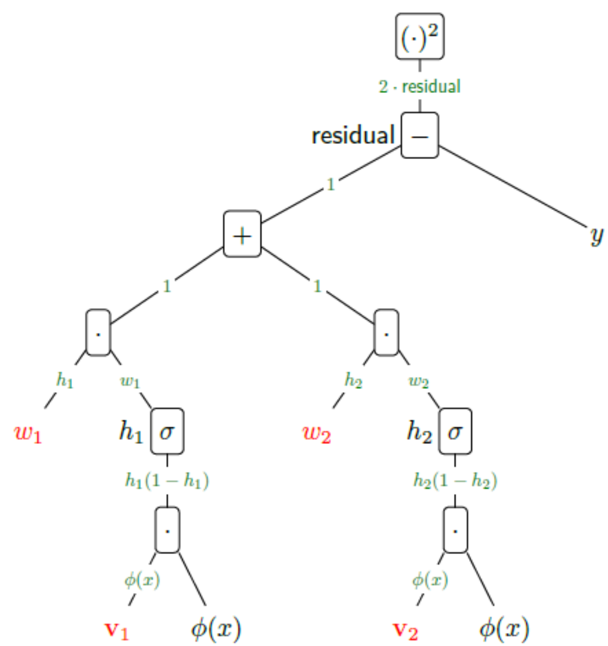
Review: training procedure of CNN

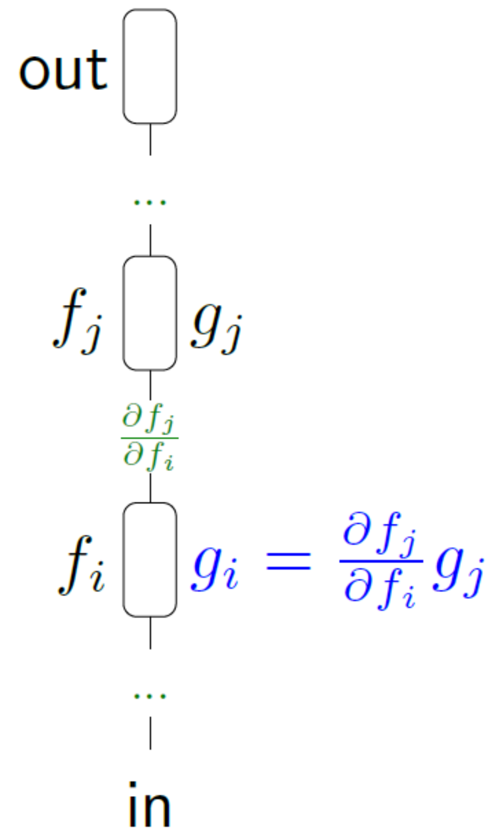
- Define network architecture
- Iteratively get a batch of training data
  - Forward propagation to compute loss （error）
  - Backward propagation to compute gradient
  - Update weights: weight = weight – learning_rate * gradient

Review: PyTorch auto-gradient

- Tensors in PyTorch can be viewed as a variables
- PyTorch tracks all operations on tensors if you set .requires_grad as True
- After finishing all operations, just call .backward to compute all gradients automatically

Recall the computation graph we used:



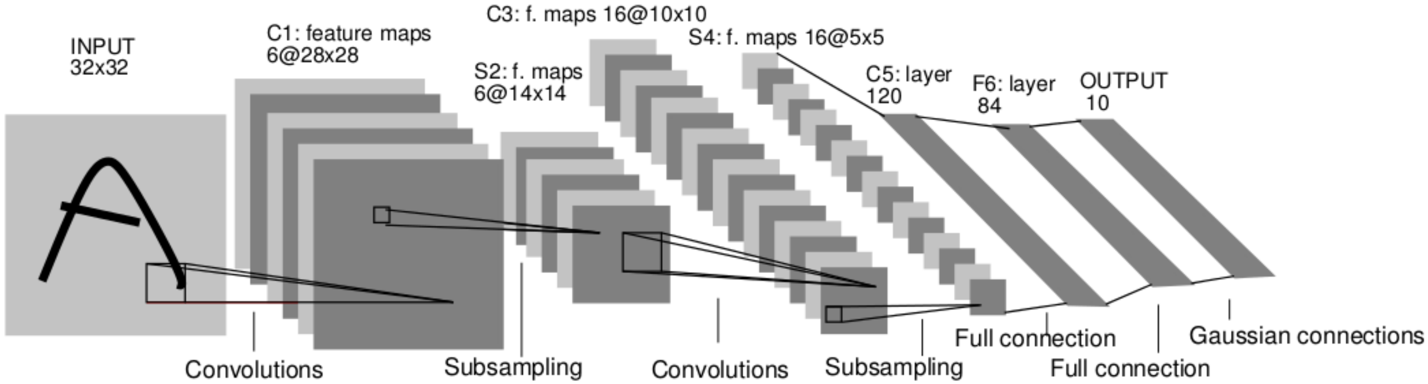


Roadmap

- PyTorch basics

- PyTorch CNN

Define the network



Depends on autograd to define models and differentiate them

Required modules:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

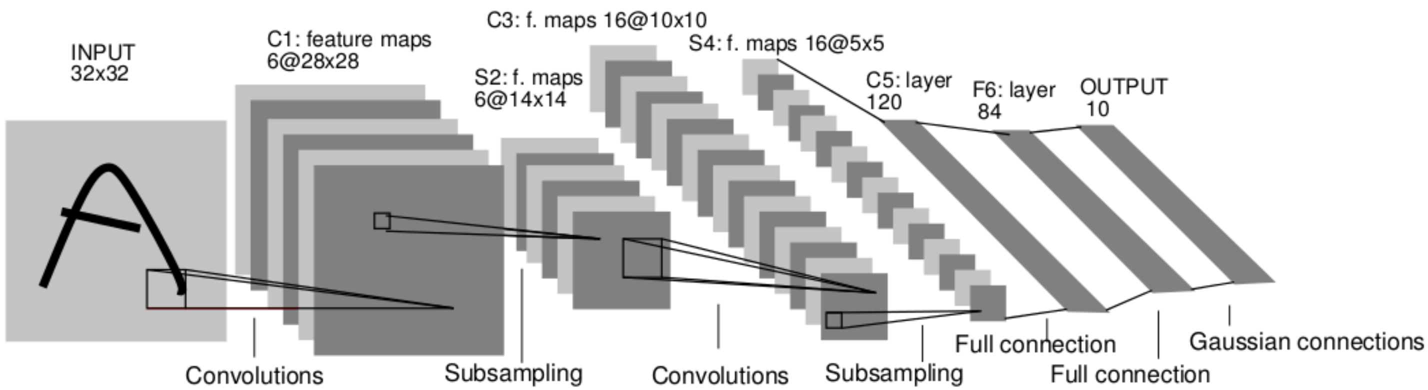
Contains useful functions such as ReLU

Define the net: convolution object

nn. Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, bias=True)

- in_channels: the number of channels of the input feature map
- out_channels: the number of channels of the output feature map
- kernel_size: (h, w), actually is h x w x in_channels
- stride: (s_h, s_w)
- padding: (p_h, p_w)
- bias: with (True) or without (False) bias

$$w_{out} = \lfloor (w_{in} - k + 2 \times p)/s + 1 \rfloor$$
$$h_{out} = \lfloor (h_{in} - k + 2 \times p)/s + 1 \rfloor$$



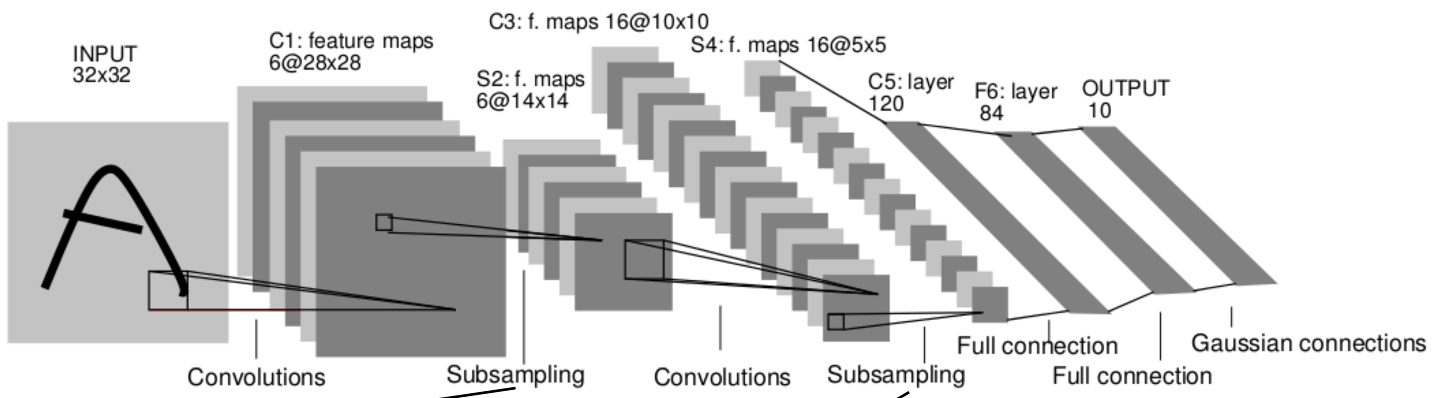
(1,6,5)

(6,16,5)

Define the network: activation and pooling functions

F.relu(input, inplace=False)

F.max_pool2d(input, kernel_size, stride=None, padding=0)

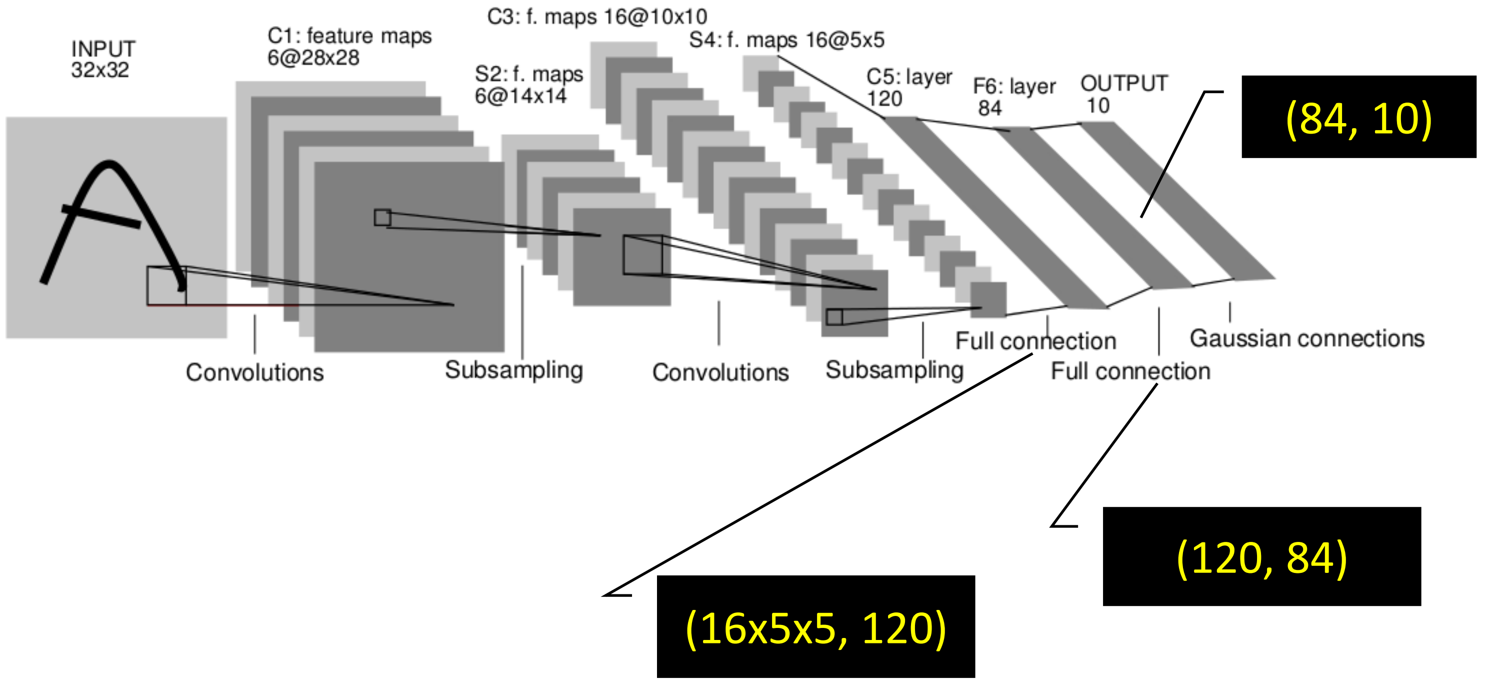


F.max_pool2d(16@10x10, (2,2))

F.max_pool2d(6@28x28, (2,2))

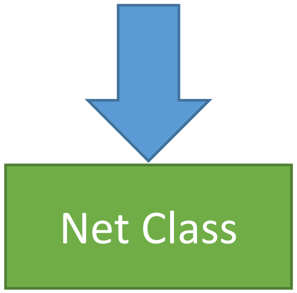
Define the network: FC layer object

nn.Linear(in_feature, out_features, bias=True)



Demo

[LeNet demo]



Output:

```
Net
(
  (conv1):Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Forward propagation

```
net = Net()
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

- torch.nn only support inputs that are a mini-batch of samples, in a form of nSamples x nChannels x Height x Width (e.g., 1 x 1 x 32 x 32)
- Net object is callable with a __call__method inherited from torch.nn.modules.module.Module, which performs forward passing

Output:

```
tensor([[ 0.0484,  0.1541, -0.0510,  0.1475,  0.0433,  0.1539, -0.0636, -0.0635,
        0.0050, -0.0705]], grad_fn=<AddmmBackward>)
```

Loss function

Recall: loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

There are a bunch of loss functions to use in PyTorch, for example here:

```
net = Net()
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)

target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1) # make it the same shape as output
criterion = nn.MSELoss()
loss = criterion(output, target)
print(loss)

criterion = nn.MSELoss()
loss = criterion(output, target)
print(loss)
```

Track the computation graph

Now, if you follow loss in the backward direction, using its .grad_fn attribute, you will see a graph of computations that looks like:

input -> conv2d -> relu -> maxpool2d -> conv2d -> relu
-> maxpool2d-> linear -> relu -> linear -> relu -> linear
-> log_softmax -> nll_loss

So, when we call loss.backward(), the whole graph is differentiated w.r.t. the loss, and all Tensors in the graph that has requires_grad=True will have their .grad tensor accumulated with the gradient.

Remember we are processing a batch of examples

Backward propagation

To backpropagate the error all we have to do is to implement loss.backward(). You need to clear the existing gradients though, else gradients will be accumulated to existing gradients.

```
net.zero_grad()
print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)
loss.backward()
print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

Output:

```
conv1.bias.grad before backward
None
conv1.bias.grad after backward
tensor([-0.0198,  0.0211, -0.0077, -0.0060,  0.0203, -0.0385])
```

Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

weight = weight - learning_rate x gradient

We can implement this using simple Python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

Update the weights

To enable other optimizers, you need to import torch.optim:

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Hand-written digits recognition

[demo]

Summary

- Define PyTorch network by inheriting the nn.Module Class
- PyTroch processes data in mini-batch
- Training routine in PyTorch: forward propagation, backward propagation, weights updating
- CNN achieves the state-of-the-art performance on image classification