Chapter 7.1 Pattern Recognition: Statistical Classifiers

Advanced Pattern Recognition -- Statistical Classifiers

Xiaojun Qi

Outline

- Perceptron Algorithm
- Neural Networks (Matlab)
- Deep Learning: CNN (PyTorch & Matlab)
- Support Vector Machines (Matlab)
- Representative Deep Convolutional Neural Networks (FYI)

Perceptron Algorithm

- Given two training sets belonging to pattern class w1 and w2, respectively. Let w(1) represent the initial weight vector, which may be arbitrarily chosen. Then at the kth training step:

if $\mathbf{x}(k) \in \omega_1$

and $\mathbf{w}'(k)\mathbf{x}(k) \le 0$

replace $\mathbf{w}(k)$

by $\mathbf{w}(k+1) = \mathbf{w}(k) + c\mathbf{x}(k)$

where c is a correction increment

if $\mathbf{x}(k) \in \omega_2$

and $\mathbf{w}'(k)\mathbf{x}(k) \ge 0$

replace $\mathbf{w}(k)$

by $\mathbf{w}(k+1) = \mathbf{w}(k) - c\mathbf{x}(k)$

Otherwise, leave **w**(*k*) unchanged, that is,

$$\mathbf{w}(k+1) = \mathbf{w}(k)$$

Perception Algorithm

- The algorithm makes a change in w if and only if the pattern being considered at the *kth* training step is misclassified by the weight vector at this step.
- The correction increment c must be positive and is assumed, for now, to be constant.

Perception Algorithm

- The perceptron algorithm is clearly a reward-and-punishment procedure, where the reward for correctly classified patterns is really the absence of punishment. That is:
  - If the pattern is classified correctly, the machine is rewarded by the fact that no change is made in w.
  - If the pattern is misclassified and the w'(k)x(k) is less than zero when it should have been greater than zero, the machine is "punished" by increasing the value of w(k) an amount proportional to x(k).
  - If w'(k)x(k) is greater than zero when it should have been less than zero, the machine is punished in the opposite mode.
- Convergence of the algorithm occurs when a weight vector classifies all patterns correctly.

Example

Consider the patterns shown in Fig. 5.2(a) on next slide (slide 7).

It is desired to apply the perceptron algorithm to these patterns in an attempt to find a solution weight vector. We see by inspection that, since the two pattern classes are linearly separable, the algorithm will be successful.

Solution:
Before the algorithm is applied, all patterns are augmented. The classes then became
ω_1 : {(0, 0, 1)' , (0, 1, 1)' }
ω_2 : {(1, 0, 1)' , (1, 1, 1)'}.
Letting *c* = 1 and **w**(1) = 0, and presenting the patterns in the above order, results in the following sequence of steps are shown on slides 8 through 11.

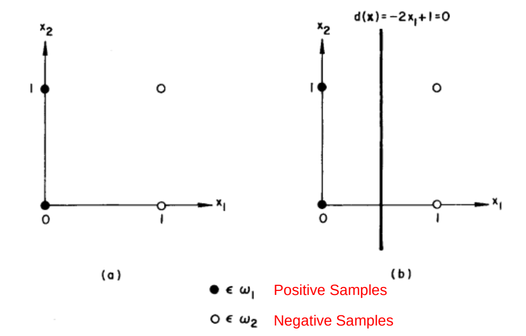


Figure 5.2. Illustration of the perceptron algorithm. (a) Patterns belonging to two classes. (b) Decision boundary determined by training

7

The first iteration through the patterns yields:

$$\mathbf{w}'(1)\mathbf{x}(1) = (0,0,0)\begin{pmatrix}0\\0\\1\end{pmatrix} = 0, \quad \mathbf{w}(2) = \mathbf{w}(1) + \mathbf{x}(1) = \begin{pmatrix}0\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(2)\mathbf{x}(2) = (0,0,1)\begin{pmatrix}0\\1\\1\end{pmatrix} = 1, \quad \mathbf{w}(3) = \mathbf{w}(2) = \begin{pmatrix}0\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(3)\mathbf{x}(3) = (0,0,1)\begin{pmatrix}1\\0\\1\end{pmatrix} = 1, \quad \mathbf{w}(4) = \mathbf{w}(3) - \mathbf{x}(3) = \begin{pmatrix}-1\\0\\0\end{pmatrix}$$

$$\mathbf{w}'(4)\mathbf{x}(4) = (-1,0,0)\begin{pmatrix}1\\1\\1\end{pmatrix} = -1, \quad \mathbf{w}(5) = \mathbf{w}(4) = \begin{pmatrix}-1\\0\\0\end{pmatrix}$$

Keep in mind the following:
The samples (e.g., x(1) and x(2)) in W1 lead to positive results and the samples (e.g., x(3) and x(4)) in W2 lead to negative results.

8

In the previous slide (slide 8), corrections on the weight vector were made in the first and third steps because of misclassification, as indicated in perception algorithm.

Since a solution has been obtained only when the algorithm yields a complete, error-free iteration through all patterns, the training set must be presented again. The machine learning process is continued by letting

$$\mathbf{x}(5) = \mathbf{x}(1) \qquad \mathbf{x}(6) = \mathbf{x}(2)$$

$$\mathbf{x}(7) = \mathbf{x}(3) \qquad \mathbf{x}(8) = \mathbf{x}(4)$$

9

The second iteration through the patterns yields:

$$\mathbf{w}'(5)\mathbf{x}(5) = 0, \quad \mathbf{w}(6) = \mathbf{w}(5) + \mathbf{x}(5) = \begin{pmatrix}-1\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(6)\mathbf{x}(6) = 1, \quad \mathbf{w}(7) = \mathbf{w}(6) = \begin{pmatrix}-1\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(7)\mathbf{x}(7) = 0, \quad \mathbf{w}(8) = \mathbf{w}(7) - \mathbf{x}(7) = \begin{pmatrix}-2\\0\\0\end{pmatrix}$$

$$\mathbf{w}'(8)\mathbf{x}(8) = -2, \quad \mathbf{w}(9) = \mathbf{w}(8) = \begin{pmatrix}-2\\0\\0\end{pmatrix}$$

10

Since two errors occurred in this iteration, the patterns are presented again:

$$\mathbf{w}'(9)\mathbf{x}(9) = 0, \quad \mathbf{w}(10) = \mathbf{w}(9) + \mathbf{x}(9) = \begin{pmatrix}-2\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(10)\mathbf{x}(10) = 1, \quad \mathbf{w}(11) = \mathbf{w}(10) = \begin{pmatrix}-2\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(11)\mathbf{x}(11) = -1, \quad \mathbf{w}(12) = \mathbf{w}(11) = \begin{pmatrix}-2\\0\\1\end{pmatrix}$$

$$\mathbf{w}'(12)\mathbf{x}(12) = -1, \quad \mathbf{w}(13) = \mathbf{w}(12) = \begin{pmatrix}-2\\0\\1\end{pmatrix}$$

11

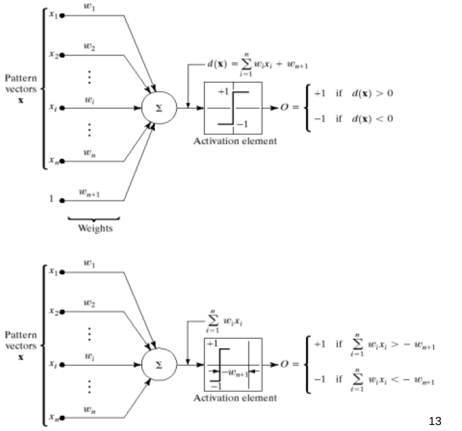
It is easily verified that in the next iteration all patterns are classified correctly. The solution vector is, therefore, **w** = (-2,0,1)'.

The corresponding decision function is $d(\mathbf{x}) = -2x_1 + 1$, which, when set equal to zero, becomes the equation of the decision boundary shown in Fig. 5.2(b).

12

a
b

FIGURE 12.14 Two equivalent representations of the perceptron model for two pattern classes.


x_1
x_2
x_i
x_n
1
w_1
w_2
w_i
w_n
w_{n+1}
Pattern vectors **x**
Weights
Σ
$d(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i + w_{n+1}$
+1
−1
Activation element
$O = \begin{cases} +1 & \text{if } d(\mathbf{x}) > 0 \\ -1 & \text{if } d(\mathbf{x}) < 0 \end{cases}$
$\sum_{i=1}^{n} w_i x_i$
$-w_{n+1}$
$O = \begin{cases} +1 & \text{if } \sum_{i=1}^{n} w_i x_i > -w_{n+1} \\ -1 & \text{if } \sum_{i=1}^{n} w_i x_i < -w_{n+1} \end{cases}$

13

Neural Networks

- There are many problems for which linear discriminants are insufficient for minimum error
- Multilayer neural network (a.k.a., Feedforward NN) is an approach that learns the nonlinearity (i.e., the weights) at the same time as the linear discriminant
  - Adopt the **backpropagation algorithm** or **generalized delta rule** (extension of Least Mean Square algorithm) for training of weights
  - Neural networks are a **flexible heuristic technique** for doing statistical pattern recognition with complicated models
  - **Network architecture or topology** plays an important role for neural net classification and the optimal topology depends on the problem at hand

14

A Sample Neural Network

Network Topology (n-N_A-N_B-N_k-N_j-N_p-W network)
1 Input Layer (the number of nodes n equals to the feature dimensions)
5 Hidden Layers
1 Output Layer

Activation Function
- Sign function
- Sigmoid function

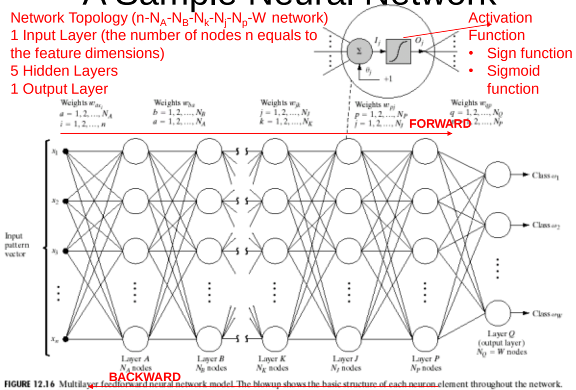


FIGURE 12.16 Multilayer feedforward neural network model. The blowup shows the basic structure of each neuron element throughout the network. The offset, θ_j, is treated as just another weight.

Three Kinds of Data Sets

- Training set - for training. Do forward and backward phase for several epochs to converge on the training set
- Validation set - patterns not used directly for gradient descent training. Used to decide when to stop training (stop training at a minimum of the error on the validation set). Excessive training leads to poor generalization
- Test set - for performance evaluation. Do forward propagation on test set to see how the network generalizes well to the test set.

16

Some Terminologies

- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.

17

Some Terminologies

- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).
- Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

18

Some Parameters

- Learning Rate: Control how fast the training will convergence.
- Momentum: Allow the network to learn more quickly on plateaus (regions of small slope which are due to too many weights).
- Initial Weights: Choose weights randomly from a single distribution for a given layer.

Training Errors

- Sum of the squared difference between desired and actual outputs over output units
  - Desired output t_k
  - Actual output z_k
  - Weights to be modified $\mathbf{w}$

c output units is assumed

$$J(\mathbf{w}) \equiv \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}\left\|\mathbf{t}_k - \mathbf{z}_k\right\|$$

  - This error is some scalar function of the weights
- The weights are adjusted by gradient descent to reduce this measure of error

Gradient Descent for Weight Modification

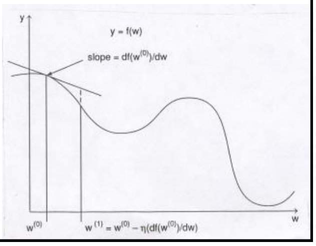
- Backpropagation learning rule is based on gradient descent.
- Weights are initialized with random values.
- Weights are changed in the direction reducing the error.

$$\Delta\mathbf{w} = -\eta\frac{\partial J}{\partial \mathbf{w}} \quad \text{or} \quad \Delta w_i = -\eta\frac{\partial J}{\partial w_i}$$

η : training rate

- Weight update at iteration *m*

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m)$$



Learning Curves

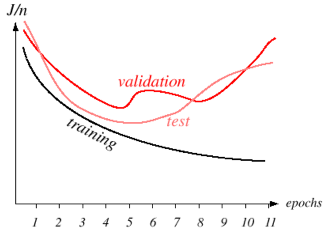


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n\sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Neural Network Workflow

The workflow for using neural network to solve a problem has seven primary steps. (Data collection in step 1, while important, generally occurs outside the MATLAB environment.)

1. Collect data
2. Create the network
3. Configure the network
4. Initialize the weights and biases
5. Train the network
6. Validate the network
7. Use the network

Neural Network (Matlab Example)

- Create a feed-forward backpropagation network to predict the labels for the inputs.
- Here is a problem consisting of inputs and targets that we would like to solve with a network.

```
inputs = [0 1 2 3 4 5 6 7 8 9 10];
targets = [0 1 2 3 4 3 2 1 2 3 4];
```

- We want to create a two-hidden-layer feed-forward network, where the first hidden layer has five neurons and the second hidden layer has one neuron. The TRAINLM network training function (default) is used.

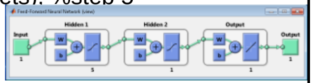
Feedforward Backpropagation Function

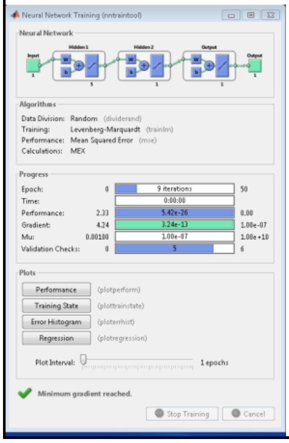
feedforwardnet(hiddenSizes,trainFcn) takes the following two arguments:

- hiddenSizes: Row vector of one or more hidden layer sizes (default = 10)
- trainFcn: Training function (default = 'trainlm')

Feedforward Backpropagation Network

```
inputs = [0 1 2 3 4 5 6 7 8 9 10]; % step 1
targets = [0 1 2 3 4 3 2 1 2 3 4];  % step 1
net = feedforwardnet([5 1]) ;  % steps 2, 3, and 4
net.trainParam.epochs = 50; % step 3
net = train(net, inputs, targets); %step 5

view(net)
outputs = net(inputs); % steps 6 and 7
plot(inputs,targets,inputs,outputs,'o') ;
perf = perform(net, outputs, targets) % return 0.6269
```





Neural Network Training (nntraintool)

Neural Network

Algorithms

Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX

Progress

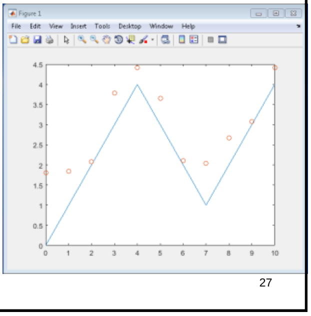
```
Epoch:                   0   9 iterations   50
Time:                        0:00:00
Performance:          2.33   5.42e-26       0.00
Gradient:             4.24   3.24e-13       1.00e-07
Mu:                0.00100   1.00e-07       1.00e+10
Validation Checks:       0   5              6
```

Plots

(plotperform)
(plottrainstate)
(ploterrhist)
(plotregression)

Plot Interval: 1 epochs

✓ Minimum gradient reached.



Deep Learning

- Deep learning is a type of machine learning, in which a model learns to perform classification tasks directly from images, text, or sound.
- Deep learning is usually implemented using a neural network architecture. The term “deep” refers to the number of layers in the network — the more layers, the deeper the network.

Applications of Deep Learning

- A smartphone app giving an instant translation of a foreign street sign
- Autonomous cars
- Tracking
- Object classification
- Speech recognition
- etc.

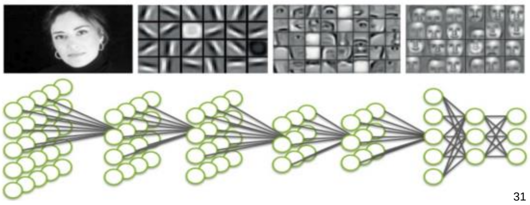
Why Is Deep Learning a Powerful Tool?

- Large datasets are freely available
  - ImageNet
  - PASCAL VoC
- High performance GPU machines are available.
- Pre-trained models are developed by researchers.
  - AlexNet
    - Trained on 1.3 Million high resolution images to recognize 1000 different categories.

Why Deep Representation

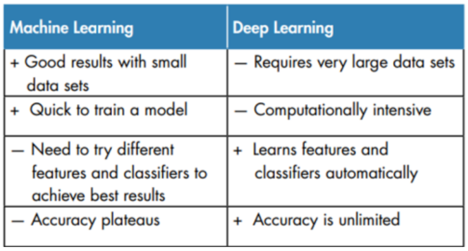
- Understand complex patterns
- Build a hierarchical feature extraction module (something that a simple NN can't do)

simple features →	more complex features →	more and more complex features
e.g. edges	e.g. basic shapes	e.g. complex shapes
	Circles, …	whole objects



Difference Between Deep Learning and Machine Learning

- Deep learning is a subtype of machine learning.
  - With machine learning, you manually extract the relevant features of an image.
  - With deep learning, you feed the raw images directly into a deep neural network that learns the features automatically.

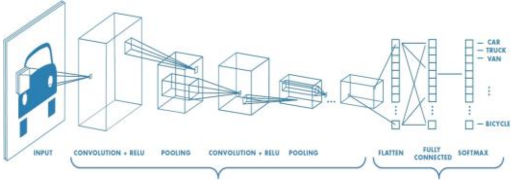


Machine Learning	Deep Learning
+ Good results with small data sets	– Requires very large data sets
+ Quick to train a model	– Computationally intensive
– Need to try different features and classifiers to achieve best results	+ Learns features and classifiers automatically
– Accuracy plateaus	+ Accuracy is unlimited

Convolutional Neural Networks (CNNs)

***CNNs* are a specific type of Deep Neural Networks**

- Suitable for image data
- Weight sharing: convolutional filters
- Fewer parameters in convolutional layers instead of their equivalent neural network parameters



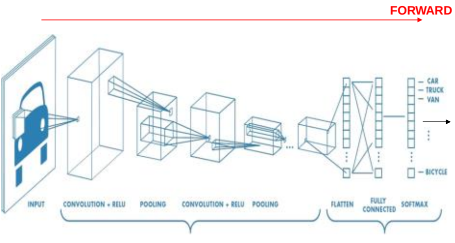
Convolutional Layers Linear Layers

CNNs

- Convolution: Each layer combines (merges, smoothes) patches from previous layers
  - Typically tries to compress large data (images) into a smaller set of robust features
  - Basic convolution can still create many features
- Pooling:
  - This step compresses and smoothes the data
  - Usually takes the average or max value across disjoint patches
- Often convolution filters and pooling are hand crafted – not learned, though tuning can occur
- After this hand-crafted/non-trained/partial-trained convolving the new set of features are used to train a supervised model
- Requires neighborhood regularities in the input space (e.g. images, stationary property)
  - Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part

CNN: Feedforward Phase

- Extract features using convolutional layers (hidden layers)
- Classify features using linear layers (fully connected layers)
- Calculate classification error (loss function)

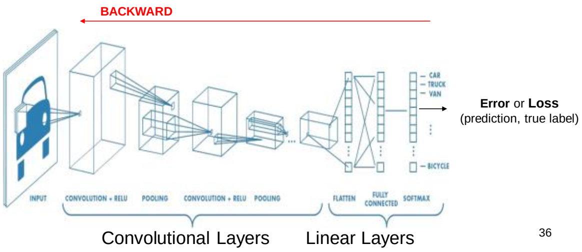


→ **Error** or **Loss** (prediction, true label)

Convolutional Layers Linear Layers

CNN: Backward Phase

1. Take record of the loss
2. Using *Gradient Descent* algorithm to update network parameters to **reduce error**:
   - Linear layer parameters
   - Convolutional layer parameters

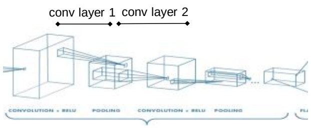


CNN: Training and Testing

- **Training**
  - Do forward and backward phase for several **iterations** to converge on the training set
    - Converge = acceptable error for all samples
- **Testing**
  - Do forward propagation on test set to see how the network generalizes well to the test set
  - No parameter update happens here (i.e., no backpropagation)

CNN in PyTorch – Convolutional Layer

***Convolutional layer*: convolution filter + activation (ReLU) + Pooling (Max-Pooling)**



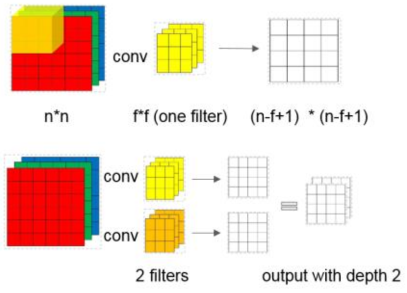
The fundamental cog of CNNS: multiple edge detection filters except the filter coefficients are learned during training in contrast to image processing

input —convolve→

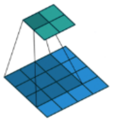
1	0	-1
1	0	-1
1	0	-1

→ Filtered output (input edge image)

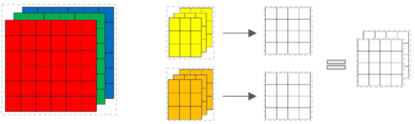
Convolution over RGB Images



See it in action:

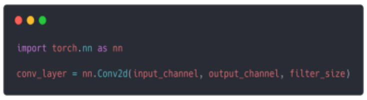


A Sample Convolution Filter



Input channel = 3 **filter_size = 3 x 3** **output_channel = 2 = number of filters**

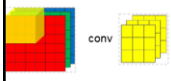
Stack multiple filters and you get one convolution filter layer



```
import torch.nn as nn

conv_layer = nn.Conv2d(input_channel, output_channel, filter_size)
```

Activation Layer: ReLU

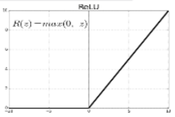
- To understand the nonlinearity, pass through a non-linear function (*f*):
  - E.g. tanh, sigmoid, relu, …
- ReLU (Rectified Linear Unit) is the most common

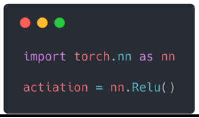


9	1	-4	3
5	6	4	5
-5	0	8	6
7	5	-6	2

activation relu →

9	1	0	3
5	6	4	5
0	0	8	6
7	5	0	2

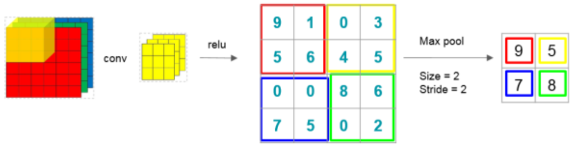




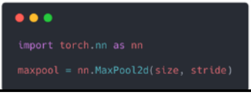
```
import torch.nn as nn

actiation = nn.Relu( )
```

Pooling Layer

- Reduce the dimension in between layers of the network



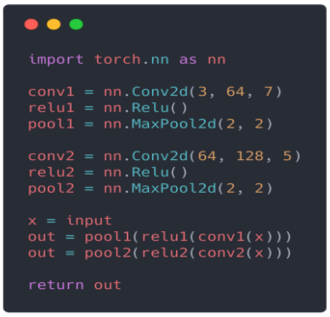
- Intuition: a high number in the output of max pooling means a particular feature is detected in a specific region -> only pick the most important feature in each region



```
import torch.nn as nn

maxpool = nn.MaxPool2d(size, stride)
```

Sample PyTorch Code

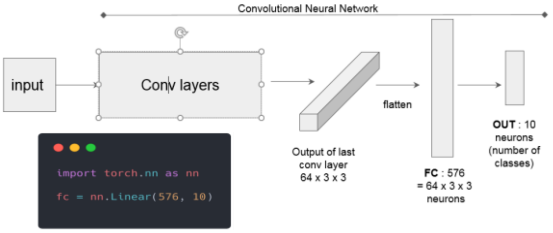
- Build the convolutional part of the network



```
import torch.nn as nn

conv1 = nn.Conv2d(3, 64, 7)
relu1 = nn.Relu()
pool1 = nn.MaxPool2d(2, 2)

conv2 = nn.Conv2d(64, 128, 5)
relu2 = nn.Relu()
pool2 = nn.MaxPool2d(2, 2)

x = input
out = pool1(relu1(conv1(x)))
out = pool2(relu2(conv2(x)))

return out
```

CNN in PyTorch: Linear Layers

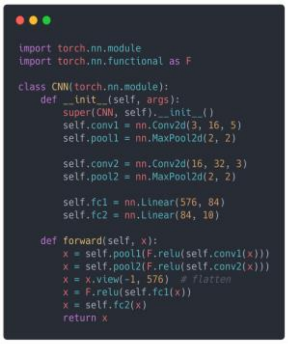
- To actually classify the output, put few Fully Connected (FC) layers after convolution layers.


Convolutional Neural Network
input
Conv layers
Output of last conv layer 64 x 3 x 3
flatten
FC : 576 = 64 x 3 x 3 neurons
OUT : 10 neurons (number of classes)

```
import torch.nn as nn

fc = nn.Linear(576, 10)
```

Sample PyTorch Code

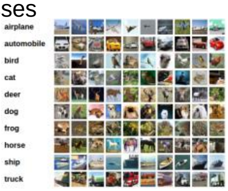
- To build a CNN in PyTorch to classify images in 10 classes
  - Define the network in the ***__init__*** method
  - Define the forward propagation in the ***forward*** method
  - ***Backward*** is automatically handled by PyTorch
    - No need to implement backward



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

class CNN(torch.nn.module):
    def __init__(self, args):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(16, 32, 3)
        self.pool2 = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(576, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 576)  # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Example 1: Training a Classifier in PyTorch

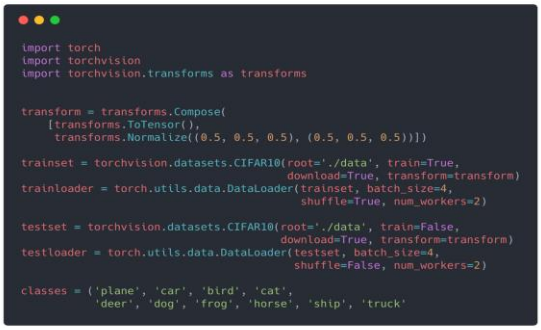
- Dataset: CIFAR10 with 10 classes
  - RGB images: 32 x 32 x 3
  - Training set: 50K images
    - 5K in each class
  - Test set: 10K images



- We will do the following:

1. Load and normalize CIFAR10 train and test set
2. Define a CNN
3. Define a loss function
4. Train the network on *train set*
5. Test the network on *test set*

Step 1: Loading and Normalizing CIFAR10

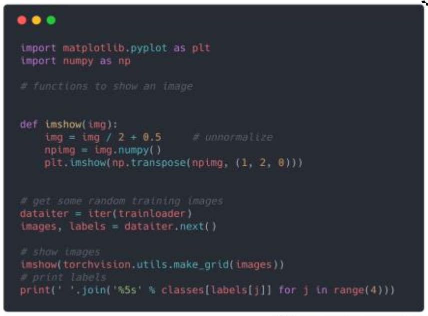


```
import torch
import torchvision
import torchvision.transforms as transforms


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck'
```

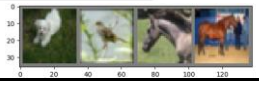
Three Notes about Step 1

- Data is represented as *tensors* in deep learning frameworks.
- Loaded data are images with values in the range [0, 1]. Everything is converted to tensors and then normalized to range [-1, 1].
- Instead of forward propagating each sample one at a time, for efficiency we pass batches of samples, in this case ***batch_size* = *4***, to the network.

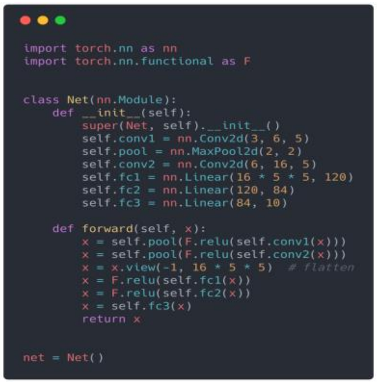
Review Loaded Images



```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

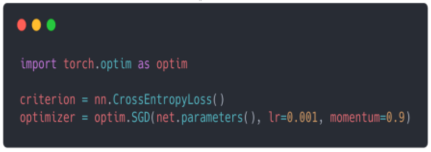


Step 2: Define a CNN



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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)  # flatten
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

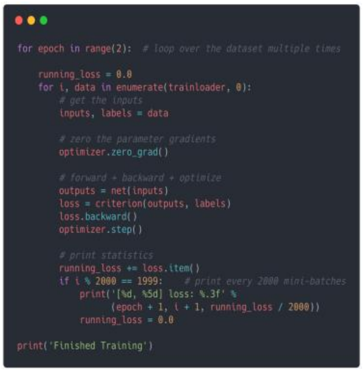
Step 3: Define a Loss Function and Optimizer



```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

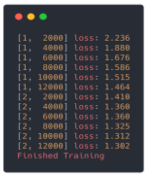
- *Cross Entropy Loss* function measures the error of the network.
- ***optim.SGD*** is the Gradient Descent algorithm which updates the network parameters to minimize the ***criterion***.

Step 4: Train CNN



```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

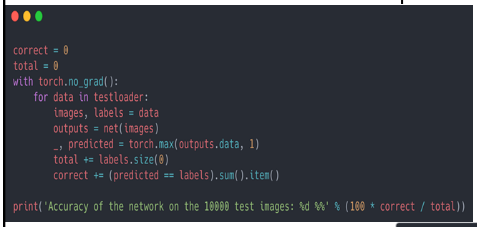
- See the whole dataset two times (i.e. 2 epochs).
- See all samples in each epoch in batches of 4 images.
- Loss (error) decreases for each iteration



```
[1,  2000] loss: 2.236
[1,  4000] loss: 1.880
[1,  6000] loss: 1.676
[1,  8000] loss: 1.586
[1, 10000] loss: 1.515
[1, 12000] loss: 1.464
[2,  2000] loss: 1.410
[2,  4000] loss: 1.360
[2,  6000] loss: 1.360
[2,  8000] loss: 1.325
[2, 10000] loss: 1.312
[2, 12000] loss: 1.302
Finished Training
```

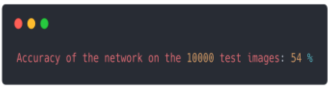
Step 5: Test CNN

- Iterate over data and feed the inputs to CNN

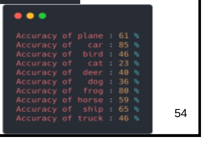


```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 * correct / total))
```

- Out:



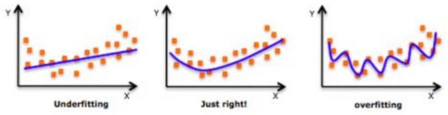
```
Accuracy of the network on the 10000 test images: 54 %
```



```
Accuracy of plane : 61 %
Accuracy of   car : 85 %
Accuracy of  bird : 46 %
Accuracy of   cat : 23 %
Accuracy of  deer : 40 %
Accuracy of   dog : 36 %
Accuracy of  frog : 80 %
Accuracy of horse : 59 %
Accuracy of  ship : 65 %
Accuracy of truck : 46 %
```

How to Improve Accuracy

- Deeper CNNs (more layers)
- Regularization to address overfitting problem



- Different optimizers
- Data augmentation
- Train for more epochs

- Reference URL: https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

Example 2: Training a Classifier in Matlab

- Dataset: MNIST database of handwritten digits, 10 classes
  - RGB images: 28 x 28
  - Training set: 60K images
    - 6K in each class
  - Test set: 10K images
- Five Steps:
  1. Load and explore image data.
  2. Define the network architecture.
  3. Specify training options.
  4. Train the network.
  5. Predict the labels of new data and calculate the classification accuracy.

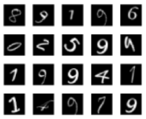
Step 1: Load Data

– Load and explore image data.

```
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
'nndatasets','DigitDataset');

 imds = imageDatastore(digitDatasetPath, ...
'IncludeSubfolders',true,'LabelSource','foldernames');
```

– Display some random images of the dataset:

```
figure;
perm = randperm(10000,20);
for i = 1:20
     subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end
```



Organize Data

- Calculate the number of images in each category:

```
labelCount = countEachLabel(imds);
```

  – The datastore contains 1000 images for each of the digits 0-9, for a total of 10000 images

- Specify Training and Validation Sets:
  – Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label.

```
numTrainFiles = 750;
[imdsTrain,imdsValidation] =
splitEachLabel(imds,numTrainFiles,'randomize');
```

Step 2: Define Network Architecture

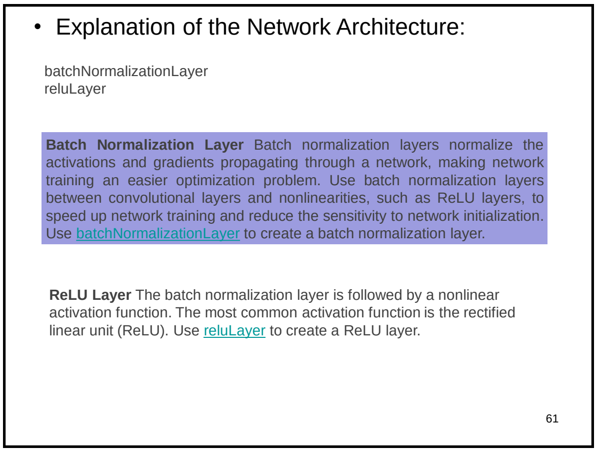
```
layers = [ imageInputLayer([28 28 1])
convolution2dLayer(3,8,'Padding','same')
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,16,'Padding','same')
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,32,'Padding','same')
 batchNormalizationLayer
 reluLayer

 fullyConnectedLayer(10)
 softmaxLayer
 classificationLayer];
```

- Explanation of the Network Architecture:

```
imageInputLayer([28 28 1])
convolution2dLayer(3,8,'Padding','same')
```

Image Input Layer An imageInputLayer is where you specify the image size, which, in this case, is 28-by-28-by-1. These numbers correspond to the height, width, and the channel size. The digit data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values.

Convolutional Layer In the convolutional layer, the first argument is filterSize, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, numFilters, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the 'Padding' name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, 'same' padding ensures that the spatial output size is the same as the input size.

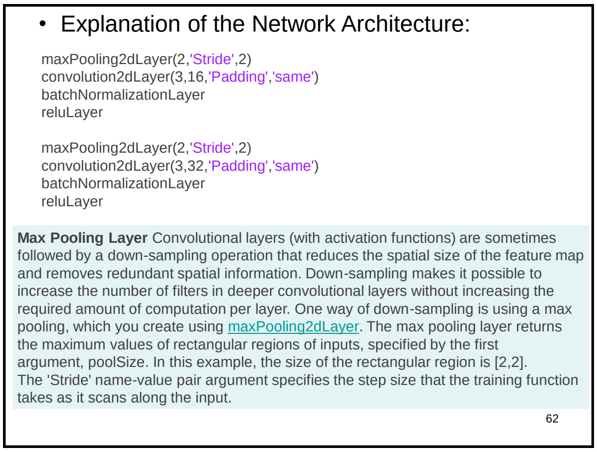


- Explanation of the Network Architecture:

```
batchNormalizationLayer
reluLayer
```

Batch Normalization Layer Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Use batchNormalizationLayer to create a batch normalization layer.

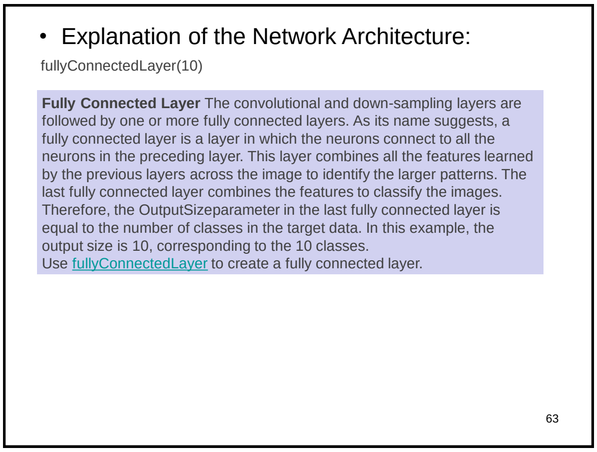
ReLU Layer The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use reluLayer to create a ReLU layer.



- Explanation of the Network Architecture:

```
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,16,'Padding','same')
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,32,'Padding','same')
batchNormalizationLayer
reluLayer
```

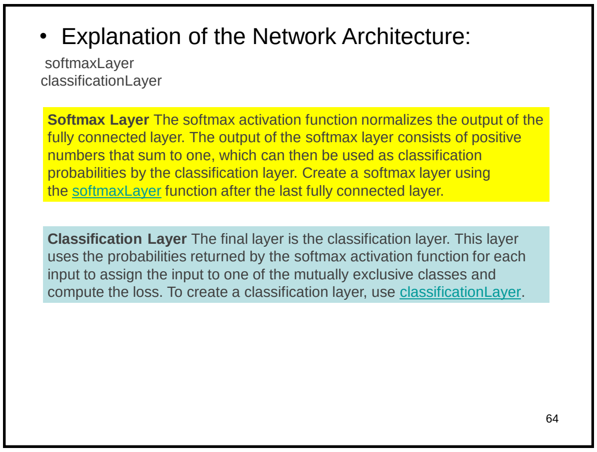
Max Pooling Layer Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using maxPooling2dLayer. The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, poolSize. In this example, the size of the rectangular region is [2,2]. The 'Stride' name-value pair argument specifies the step size that the training function takes as it scans along the input.



- Explanation of the Network Architecture:

```
fullyConnectedLayer(10)
```

Fully Connected Layer The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore, the OutputSizeparameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes. Use fullyConnectedLayer to create a fully connected layer.

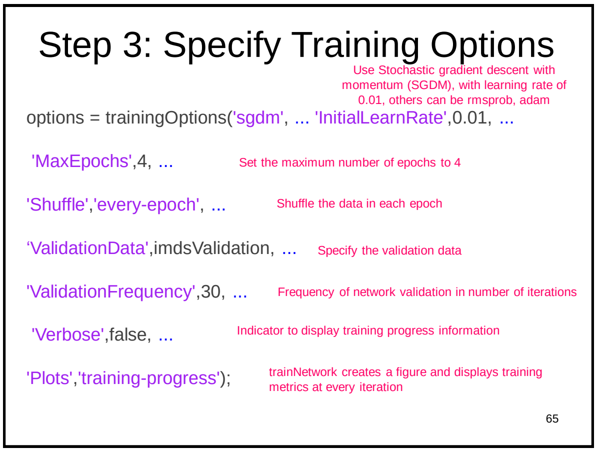


- Explanation of the Network Architecture:

```
softmaxLayer
classificationLayer
```

Softmax Layer The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the softmaxLayer function after the last fully connected layer.

Classification Layer The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer, use classificationLayer.



Step 3: Specify Training Options

Use Stochastic gradient descent with momentum (SGDM), with learning rate of 0.01, others can be rmsprob, adam

```
options = trainingOptions('sgdm', ... 'InitialLearnRate',0.01, ...
 'MaxEpochs',4, ...
'Shuffle','every-epoch', ...
'ValidationData',imdsValidation, ...
'ValidationFrequency',30, ...
 'Verbose',false, ...
'Plots','training-progress');
```

Set the maximum number of epochs to 4

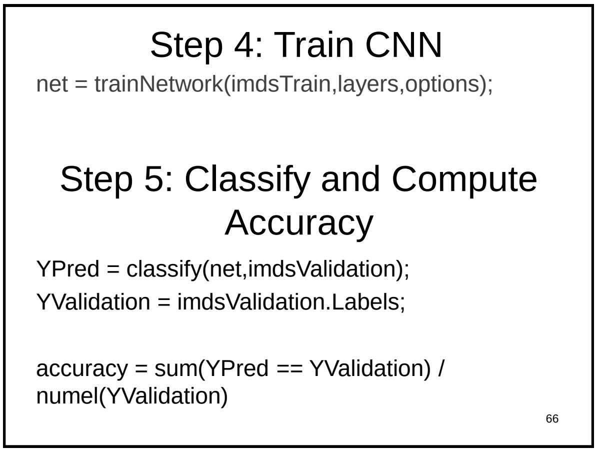
Shuffle the data in each epoch

Specify the validation data

Frequency of network validation in number of iterations

Indicator to display training progress information

trainNetwork creates a figure and displays training metrics at every iteration



Step 4: Train CNN

```
net = trainNetwork(imdsTrain,layers,options);
```

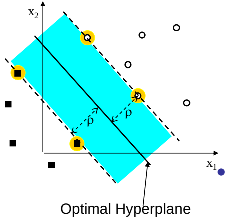
Step 5: Classify and Compute Accuracy

```
YPred = classify(net,imdsValidation);
YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation) /
numel(YValidation)
```

Support Vector Machines (SVMs)

- SVM is a classifier derived from statistical learning theory by Vapnik and Chervonenkis
- SVM becomes famous when, using pixel maps as input, it gives accuracy comparable to sophisticated neural networks with elaborated features in a handwriting recognition task.
- Currently, SVM is closely related to:
  - Kernel methods
  - Large margin classifiers
  - Reproducing kernel Hilbert space
  - Gaussian process

Optimal Hyper-plane and Support Vectors -- Linearly Separable Case



- Optimal hyper-plane should be in the center of the gap. It gives the largest margin (ρ) of separation between the classes. This optimal hyperplane bisects the shortest line between the convex hulls of the two classes. It is also orthogonal to this line.
- Support Vectors – A relative small subset of the data points on the boundaries (margins).

Support vectors alone can determine the optimal hyper-plane. In other words, the optimal hyper-plane is completely determined by these support vectors.

Optimal Hyper-Plane Formulation -- Mathematical Point of View

- The goal is to find w and b to define the optimal hyper-plane, from the training data

Given $\{(x_i, d_i)\}_{i=1}^{N}$

we want to find the optimal hyperplane such that

$d_i(w^T x_i + b) \geq 1$ for $i = 1, 2, \ldots, N$

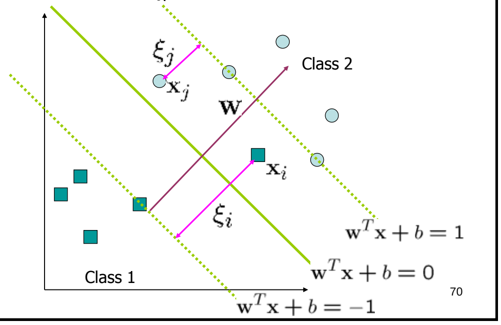
and the weight ve ctor minimizes $\|w\|$ or

$\phi(w) = \frac{1}{2} w^T w$ the 1/2 is for convenienc e

- ➔ This is equivalent to: $\min_{w,b} \left(\frac{1}{2} w^T w \right)$ subject to $d_i(w^T x_i + b) \geq 1$

Soft Margin SVMs: Illustration

- We allow “error” ξ_i in classification


ξ_j
$\mathbf{x}_j$
Class 2
$\mathbf{w}$
$\mathbf{x}_i$
ξ_i
$\mathbf{w}^T\mathbf{x} + b = 1$
Class 1
$\mathbf{w}^T\mathbf{x} + b = 0$
$\mathbf{w}^T\mathbf{x} + b = -1$

Soft Margin SVMs

Since $\zeta_i > 1$ implies misclassification, the cost function must include a term to minimize the number of samples that are misclassified. An approximated cost function can be used:

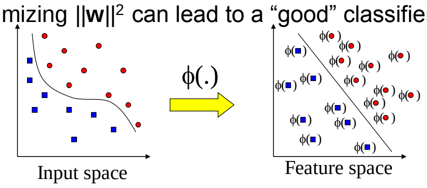
$$\Phi(W, \zeta) = \frac{1}{2} W^T W + C \sum_{i=1}^{N} \zeta_i$$

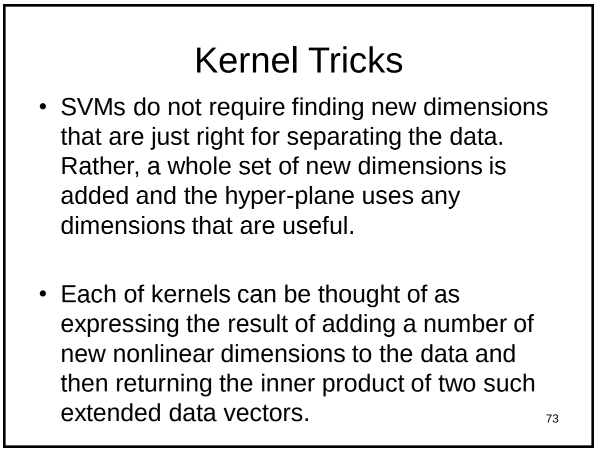
Where C is the tradeoff parameter between error and margin. It is also called **slack penalty**. That is:

$$\min_{w,b,\zeta} \left\{ \frac{1}{2} w^T w + C \sum_{i=1}^{N} \zeta_i \right\} \text{ subject to } d_i(w^T x_i + b) \geq 1 - \zeta_i \text{ and } \zeta_i > 0$$

Non-Linear SVMs (Cont.)

- Possible problem of the transformation
  - High computation burden and hard to get a good estimate
- SVM solves the following two issues simultaneously
  - Kernel tricks for efficient computation
  - Minimizing $\|\mathbf{w}\|^2$ can lead to a “good” classifier

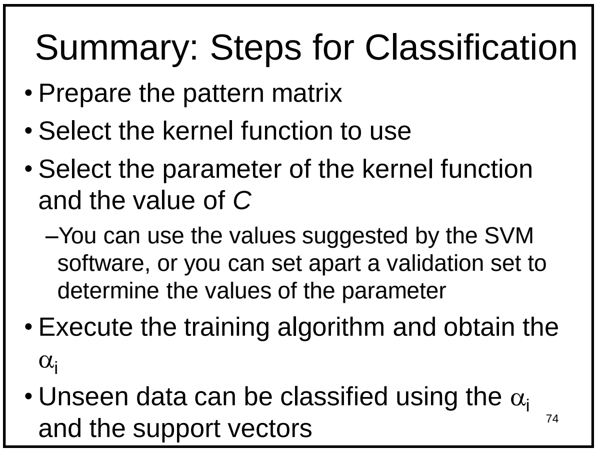




Kernel Tricks

- SVMs do not require finding new dimensions that are just right for separating the data. Rather, a whole set of new dimensions is added and the hyper-plane uses any dimensions that are useful.
- Each of kernels can be thought of as expressing the result of adding a number of new nonlinear dimensions to the data and then returning the inner product of two such extended data vectors.

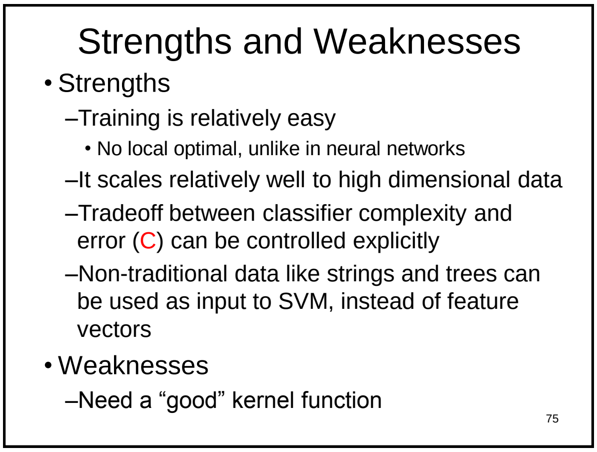
73



Summary: Steps for Classification

- Prepare the pattern matrix
- Select the kernel function to use
- Select the parameter of the kernel function and the value of *C*
  - –You can use the values suggested by the SVM software, or you can set apart a validation set to determine the values of the parameter
- Execute the training algorithm and obtain the α_i
- Unseen data can be classified using the α_i and the support vectors

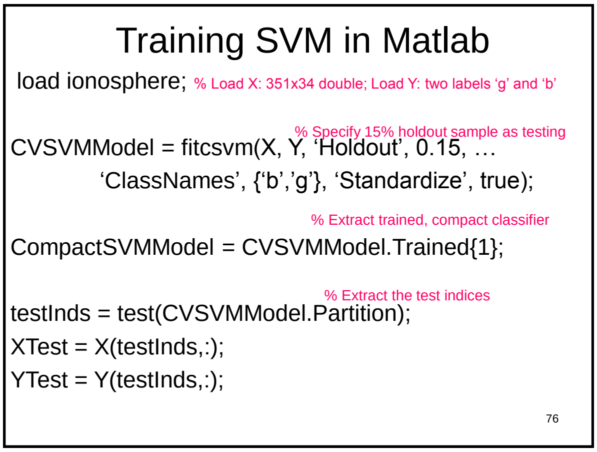
74



Strengths and Weaknesses

- Strengths
  - –Training is relatively easy
    - No local optimal, unlike in neural networks
  - –It scales relatively well to high dimensional data
  - –Tradeoff between classifier complexity and error (C) can be controlled explicitly
  - –Non-traditional data like strings and trees can be used as input to SVM, instead of feature vectors
- Weaknesses
  - –Need a “good” kernel function

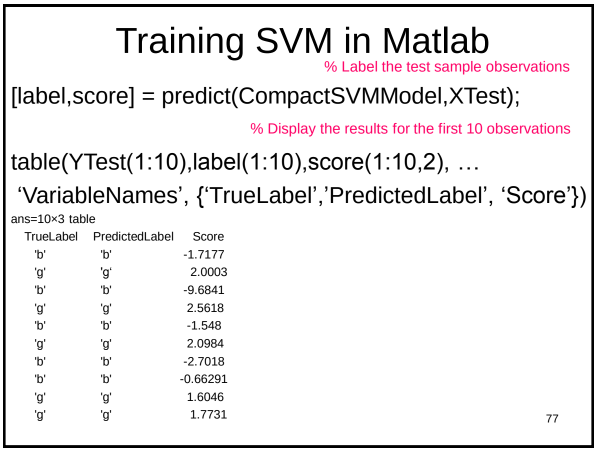
75



Training SVM in Matlab

```
load ionosphere; % Load X: 351x34 double; Load Y: two labels ‘g’ and ‘b’

% Specify 15% holdout sample as testing
CVSVMModel = fitcsvm(X, Y, ‘Holdout’, 0.15, …
        ‘ClassNames’, {‘b’,’g’}, ‘Standardize’, true);

% Extract trained, compact classifier
CompactSVMModel = CVSVMModel.Trained{1};

% Extract the test indices
testInds = test(CVSVMModel.Partition);
XTest = X(testInds,:);
YTest = Y(testInds,:);
```

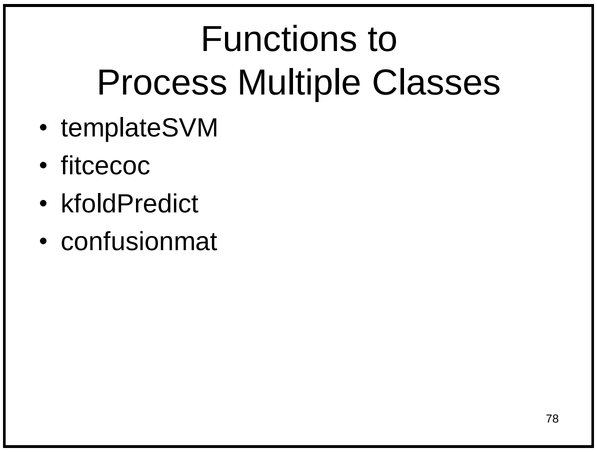
76



Training SVM in Matlab

```
% Label the test sample observations
[label,score] = predict(CompactSVMModel,XTest);

% Display the results for the first 10 observations
table(YTest(1:10),label(1:10),score(1:10,2), …
 ‘VariableNames’, {‘TrueLabel’,’PredictedLabel’, ‘Score’})
ans=10×3 table
   TrueLabel   PredictedLabel    Score
     'b'           'b'          -1.7177
     'g'           'g'           2.0003
     'b'           'b'          -9.6841
     'g'           'g'           2.5618
     'b'           'b'          -1.548
     'g'           'g'           2.0984
     'b'           'b'          -2.7018
     'b'           'b'          -0.66291
     'g'           'g'           1.6046
     'g'           'g'           1.7731
```

77



Functions to Process Multiple Classes

- templateSVM
- fitcecoc
- kfoldPredict
- confusionmat

78

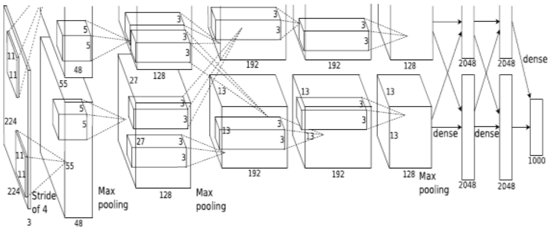
Representative Deep Convolutional Neural Networks (DCNN)

The following is FYI only

- AlexNet (2012)
- ZFNet (2014)
- GoogLeNet (2015)
- VGGNet (2015)
- ResNet (2016)
- DenseNet (2017)

AlexNet (2012)

Trained a large, DCNN to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes.



The NN, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax.

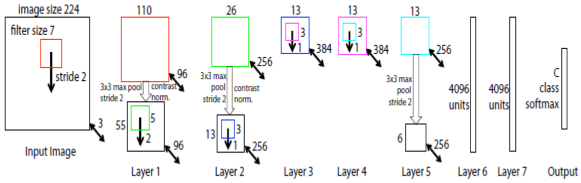
A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," LSVRC,

AlexNet Implementation

- Input: Centered (zero-mean) RGB values
- Data Augmentation: Train on 224x224 patches extracted randomly from 256x256 images, and also their horizontal reflections.
- Drop-out: Independently set each hidden unit activity to zero with 0.5 probability; do this in the two globally-connected hidden layers at the net's output

ZFNet (2014)

- Introduce a novel visualization technique that gives insight into the function of intermediate feature layers and the operation of the classifier.
- Use these visualizations to find model architectures that outperform AlexNet on the ImageNet classification benchmark.



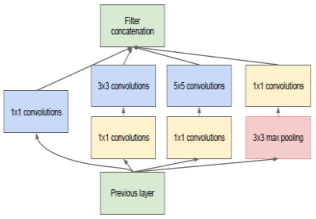
M. Zeiler & R. Fergus, “Visualizing and Understanding Convolutional Networks,” ECCV, 2014.

GoogLeNet (2015)

- Improve utilization of the computing resources inside the network.
- Increase the depth and width of the network while keeping the computational budget constant.
- Employ the Hebbian principle (neurons that fire together, wire together) and the intuition of multi-scale processing to make the architectural decisions.
- GoogLeNet is a 22 layers deep network. Its codename is Inception.

C. Szegedy, W. Liu, and Y. Jia, et al., “Going Deeper with Convolutions,” CVPR, 2015.

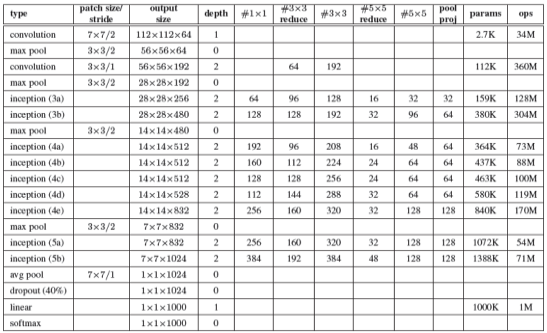
GoogLeNet (2015)



Inception module with dimensionality reduction

In order to avoid patch-alignment issues, current incarnations of the Inception architecture are restricted to filter sizes 1x1, 3x3 and 5x5; It also means that the suggested architecture is a combination of all those layers with their output filter banks concatenated into a single output vector forming the input of the next stage. Additionally, since pooling operations have been essential for the success of current convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too.

GoogLeNet (2015): Inception



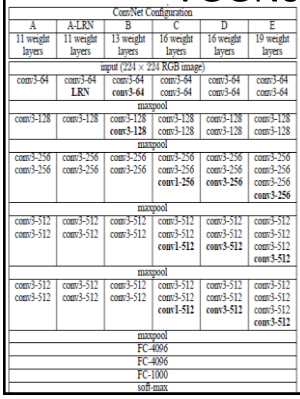
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

VGGNet (2015)

- Thoroughly evaluate networks of increasing depth using an architecture with very small (3×3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16–19 weight layers.
- Two best-performing models are shown in D and E column in the table shown in the next slide.

K. Simonyan & A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR, 2015.

VGGNet (2015)



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

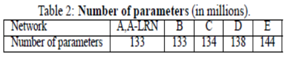


Table 2: **Number of parameters (in millions).**

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

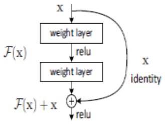
ConvNet configurations (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold).

ResNet (2016)

- Use a residual learning framework to ease the training of networks that are substantially deeper than those used previously.
- Explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions.
- These residual networks are easier to optimize, and can gain accuracy from considerably increased depth.
- ResNet has a depth of up to 152 layers--8x deeper than VGG nets but still having lower complexity.

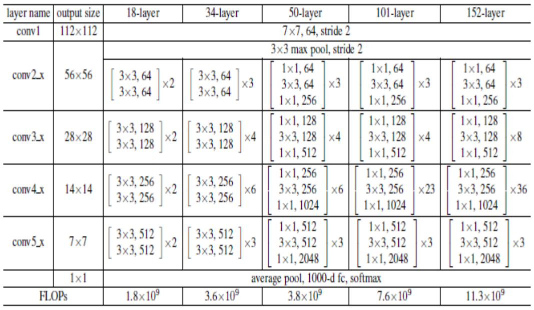
K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, 2016.

Residual Learning: Building Block



- The formulation of F(x)+x can be realized by feedforward NN with “shortcut connections”.
- Shortcut connections are those skipping one or more layers. In ResNet, they simply perform identity mapping, and their outputs are added to the outputs of the stacked layers.
- Identity shortcut connections add neither extra parameter nor computational complexity.
- The entire network can still be trained end-to-end by stochastic gradient descent with backpropagation, and can be easily implemented using common libraries (e.g., Caffe) without modifying the solvers.

ResNet (2016)



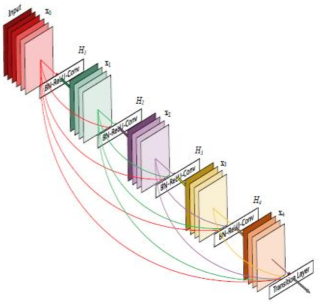
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

DenseNet (2017)

- Connect each layer to every other layer in a feed-forward fashion.
- L-layer network has L(L+1)/2 connections
- For each layer, the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers.
- Advantages:
  1. alleviate the vanishing-gradient problem
  2. strengthen feature propagation
  3. encourage feature reuse
  4. substantially reduce the number of parameters

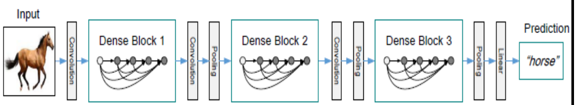
G. Huang, Z. Liu, and L. Maaten, "Densely Connected Convolutional Networks,“ CVPR, 2017.

Dense Block Illustration



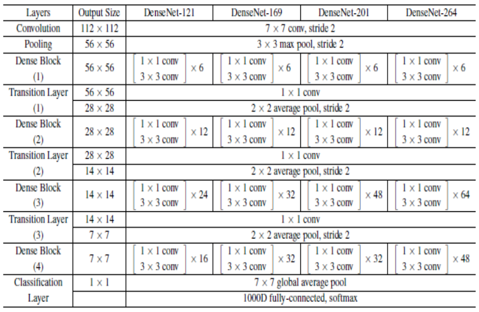
A 5-layer dense block with a growth rate of k = 4.
Each layer takes all preceding feature-maps as input.

Dense Block Illustration



A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

DenseNet (2017)



Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 24$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 48$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 16$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			