PATTERN RECOGNITION AND MACHINE LEARNING

CHAPTER 5: NEURAL NETWORKS

Outlines

- Feedforward Network Functions
- Network Training
- Error Backpropagation
- Jacobian Matrix
- Hessian Matrix
- CNN and GAN

Feed-forward Network Functions

- Goal: to extend linear model by making the basis functions depend on parameters, allow these parameters to be adjusted.

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

First layer

activations

weights

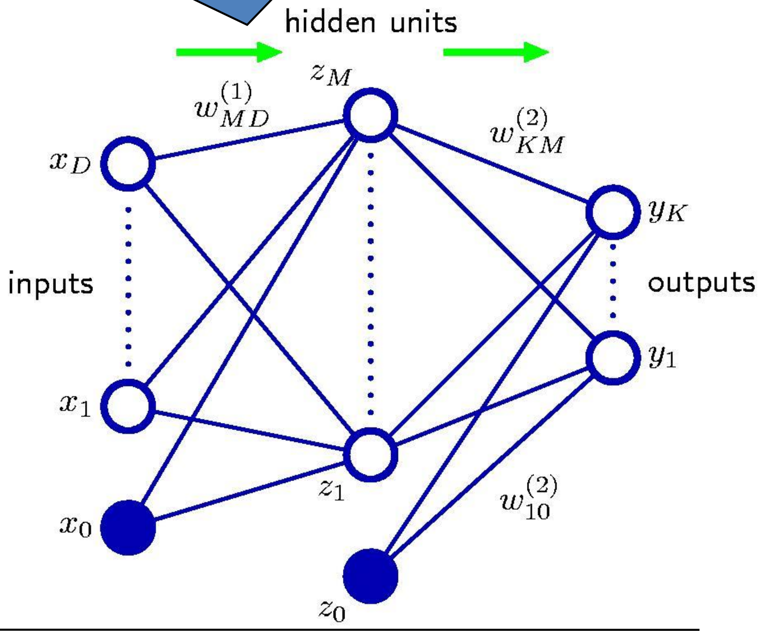
biases

$$z_j = h(a_j)$$

nonlinear activation function

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

Forward propagation of information



Outlines

- Feedforward Network Functions
- Network Training
- Error Backpropagation
- Jacobian Matrix
- Hessian Matrix
- CNN and GAN

Network Training

- t has a Gaussian distribution with an x-dependent mean

$$p(t \mid x, w) = N(t \mid y(x, w), \beta^{-1})$$

$$p(\mathbf{t} \mid \mathbf{x}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n \mid x_n, w, \beta)$$ (likelihood function)

$$\frac{\beta}{2} \sum_{n=1}^{N} \{y(X_n, w) - t_n\}^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi)$$ (negative log)

- Maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function

(C) 2007, SNU Biointelligence Lab, http://bi.snu.ac.kr/

Network Training

- The choice of output unit activation function and matching error function

  Standard regression problems:

    Error function: Negative log-likelihood function

    Output: identity

  For multiple binary classification problems:

    Error function: cross-entropy error function

$$E(w) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln (1 - y_n)\}$$

    Output: logistic sigmoid

  For multiclass problems:

    Multiclass cross-entropy error function

$$E(w) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

    a softmax activation function

$$\frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

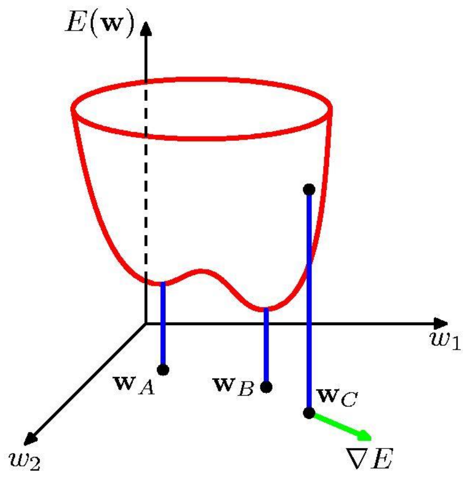
Parameter Optimization

- Error E(w) is a smooth continuous function of w and smallest value will occur at a point in weight space

$\nabla E(w) = 0$

Global minimum

Local minima



- Most techniques involve choosing some initial value for weight vector and then moving through weight space in a succession of steps of the form $w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}$
(Many algorithms make use of gradient info)

Local Quadratic Approximation

- Taylor expansion of E(w) around some point

$$E(w) \simeq E(\hat{w}) + (w-\hat{w})^T \mathbf{b} + \frac{1}{2}(w-\hat{w})^T \mathbf{H}(w-\hat{w})$$

$$\mathbf{b} \equiv \nabla E \,|_{w=\hat{w}}$$ (gradient)

$$(\mathbf{H})_{ij} \equiv \frac{\partial E}{\partial w_i \partial w_j} \,|_{w=\hat{w}}$$ (Hessian Matrix)

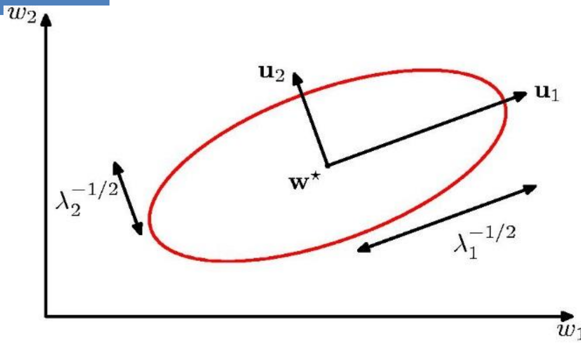
- Local approximation to the gradient

$$\nabla E \simeq \mathbf{b} + \mathbf{H}(w-\hat{w})$$

- Local quadratic approximation when $\nabla E = 0$ at w*

$$E(w) = E(w^*) + \frac{1}{2}(w-w^*)^T \mathbf{H}(w-w^*)$$

$$E(w) = E(w^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2$$



Use of Gradient Information

- In the quadratic approximation, computational cost to find minimum is O(W^3)

  W is the dimensionality of w

  - perform O(W^2) evaluations, each of which would require O(W) steps.

- In an algorithm that makes use of the gradient information, computational cost is O(W^2)
  - By using error backpropagation, O(W) gradient evaluations and each such evaluation takes only O(W) steps.

Gradient Descent Optimization

- Weight update to comprise a small step in the direction of the negative gradient

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)})$$

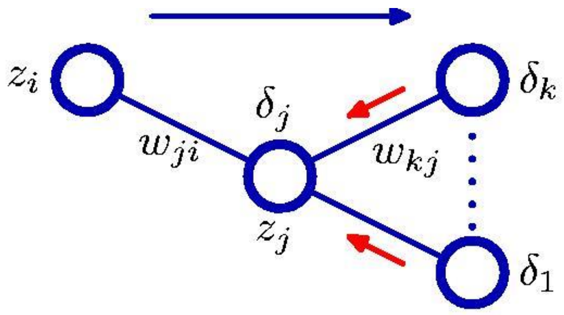
- Batch method
  - Techniques that use the whole data set at once.
  - The error function always decreases at each iteration unless the weight vector has arrived at a local or global minimum.

- On-line version of gradient descent
  - Sequential gradient descent or stochastic gradient descent
  - Update to the weight vector based on one data point at a time.
  - Can handle redundancy in the data much more efficiently.
  - The possibility of escaping from local minima.

Outlines

- ➢ Feedforward Network Functions
- ➢ Network Training
- ➢ Error Backpropagation
- ➢ Jacobian Matrix
- ➢ Hessian Matrix
- ➢ CNN and GAN

Error Backpropagation



Error Backpropagation

Apply an input vector X$_n$ to the network and forward propagate through the network using equations below to find the activations of all the hidden and output units.

$$a_j = \sum_i w_{ji} z_i \qquad z_j = h(a_j)$$

Evaluate the δ_k for all the output units using $\delta_k = y_k - t_k$

Backpropagate the δ 's using

to obtain δ_j for each hidden unit in the network $\delta_j = h'(a_j) \sum_k w_{ji} \delta_k$

Use $\dfrac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$ to evaluate the required derivatives.

A Simple Example

$$h(a) \equiv \tanh(a)$$

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

$$h'(a) = 1 - h(a)^2$$

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$

(y_k: output unit k, t_k: the corresponding target)

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i$$

$$z_j = \tanh(a_j)$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$

$$\delta_j = (1 - z_j^2)\sum_{k=1}^{K} w_{kj}\delta_k$$

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i$$

$$\frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

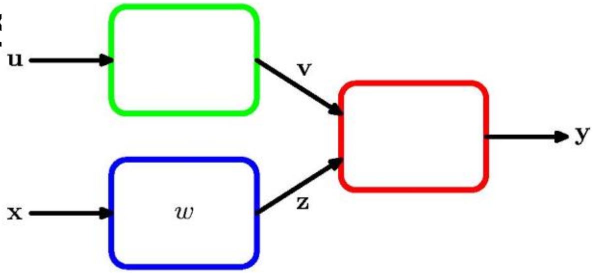
Outlines

- Feedforward Network Functions
- Network Training
- Error Backpropagation
- Jacobian Matrix
- Hessian Matrix
- CNN and GAN

The Jacobian Matrix

- The technique of backpropagation can also be applied to the calculation of other derivatives.
- The Jacobian matrix
  - Elements are given by the derivatives of the network outputs w.r.t. the inputs

$$J_{ki} = \frac{\partial y_k}{\partial x_i}$$



  - Minimizing an error function E w.r.t. the parameter

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w}$$

The Jacobian Matrix

- A measure of the *local sensitivity* of the outputs to change in each of the input variables.
  - In general, the network mapping is nonlinear, and so the elements will not be constants. This is valid provided $|\Delta x_i|$ are small.

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_j} \Delta x_i$$

$$\frac{\partial y_k}{\partial a_l} = \delta_{kj} \sigma'(a_j)$$

  - The evaluation of the Jacobian Matrix

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} = \sum_j w_{ji} \frac{\partial y_k}{\partial a_j}$$

$$\frac{\partial y_k}{\partial a_j} = \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} = h'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l}$$

(Sigmoidal activation function)

Outlines

- Feedforward Network Functions
- Network Training
- Error Backpropagation
- Jacobian Matrix
- Hessian Matrix
- CNN and GAN

Hessian: Diagonal approximation

- Inverse of the Hessian is useful, and inverse of diagonal matrix is trivial to evaluate.
- Consider an error function
  - Replaces the off-diagonal elements with zeros
  - The diagonal elements of the Hessian:

$$\frac{\partial^2 E_n}{\partial w_{ji}^2} = \frac{\partial^2 E_n}{\partial a_j^2} z_i^2$$

$$\frac{\partial^2 E_n}{\partial a_j^2} = h'(a_j)^2 \sum_k \sum_{k'} w_{kj} w_{k'j} \frac{\partial^2 E_n}{\partial a_k \partial a_{k'}} + h''(a_j) \sum_k w_{kj} \frac{\partial E_n}{\partial a_k}$$

$$\frac{\partial^2 E_n}{\partial a_j^2} = h'(a_j)^2 \sum_k w_{kj}^{\ 2} \frac{\partial^2 E_n}{\partial a_k^{\ 2}} + h''(a_j) \sum_k w_{kj} \frac{\partial E_n}{\partial a_k}$$ (neglect off-diagonal)

Outer Product Approximation

$$\mathbf{H} = \nabla\nabla E = \sum_{n=1}^{N} \nabla y_n \nabla y_n + \sum_{n=1}^{N} (y_n - t_n)\nabla\nabla y_n$$

- Output *y* happen to be very close to the target values t, the second term will be small and can be neglected.
    - Or the value of y – t is uncorrelated with the value of the second derivative term, then the whole term will average to zero.

$$\mathbf{H} \simeq \sum_{n=1}^{N} \mathbf{b}_n \mathbf{b}_n^T$$

$$\mathbf{b}_n = \nabla y_n = \nabla a_n$$

Inverse Hessian

- A procedure for approximating the inverse of the Hessian
  - First, we write the outer-product approximation
  - Derive a sequential procedure for building up the Hessian by including data points one at a time.

$$\mathbf{H}_{L+1} = \mathbf{H}_L + \mathbf{b}_{L+1}\mathbf{b}_{L+1}^T$$

$$\mathbf{H}_{L+1}^{-1} = \mathbf{H}_L^{-1} - \frac{\mathbf{H}_L^{-1}\mathbf{b}_{L+1}\mathbf{b}_{L+1}^T\mathbf{H}_L^{-1}}{1+\mathbf{b}_{L+1}^T\mathbf{H}_L^{-1}\mathbf{b}_{L+1}}$$

- The initial matrix $H_0 = H + \alpha I$

  α is a small quantity, not sensitive to the precise value of α.

Finite Differences

- By using a symmetrical central differences formulation

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}} = \frac{1}{4\varepsilon}\{E(w_{ji}+\varepsilon, w_{lk}+\varepsilon) - E(w_{ji}+\varepsilon, w_{lk}-\varepsilon)$$
$$-E(w_{ji}-\varepsilon, w_{lk}+\varepsilon) + E(w_{ji}-\varepsilon, w_{lk}-\varepsilon)\} + O(\varepsilon^2)$$

Require O(W^3) operations to evaluate the complete Hessian

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}} = \frac{1}{2\varepsilon}\{\frac{\partial E}{\partial w_{ji}}(w_{lk}+\varepsilon) - \frac{\partial E}{\partial w_{ji}}(w_{lk}-\varepsilon)\} + O(\varepsilon^2)$$

By applying central differences to the first derivatives of the error function. Costs: O(W^2)

Evaluation of the Hessian

The Hessian can also be evaluated exactly.

- Using extension of the technique of backpropagation used to evaluate first derivatives.

$$M_{kk'} = \frac{\partial^2 E_n}{\partial a_k \partial a_{k'}}$$

$$\frac{\partial^2 E_n}{\partial w_{kj}^{(2)} \partial w_{k'j'}^{(2)}} = z_j z_{j'} M_{kk'}$$

$$\delta_k = \frac{\partial E_n}{\partial a_k}$$

(Both weights in the second layer)

$$\frac{\partial^2 E_n}{\partial w_{ji}^{(1)} \partial w_{j'i'}^{(1)}} = x_i x_{i'} h''(a_{j'}) I_{jj'} \sum_k w_{kj'}^{(2)} \delta_k$$

$$+ x_i x_{i'} h'(a_{j'}) h'(a_j) \sum_k \sum_{k'} w_{k'j'}^{(2)} w_{kj}^{(2)} M_{kk'}$$

(Both weights in the first layer)

$$\frac{\partial^2 E_n}{\partial w_{ji}^{(1)} \partial w_{kj'}^{(2)}} = x_i h'(a_{j'}) \{ \delta_k I_{jj'} + z_j \sum_{k'} w_{k'j'}^{(2)} H_{kk'} \}$$

(Both weights in each layer)

Fast Multiplication by the Hessian

- Try to find an efficient approach to evaluating $v^T H$ directly.

  O(W) operations

  - $v^T H = v^T \nabla (\nabla E)$ (∇: the gradient operator in weight space)
  - Introducing new notation $R\{w\} = v$， to denote the operator $v^T \nabla$

$$R\{a_j\} = \sum_i v_{ji} x_i \qquad R\{z_j\} = h'(a_j) R\{a_j\}$$

$$R\{\delta_j\} = h''(a_j) R\{a_j\} \sum_k w_{kj}\delta_k + h'(a_j) \sum_k v_{kj}\delta_k + h'(a_j) \sum_k w_{kj} R\{\delta_k\}$$

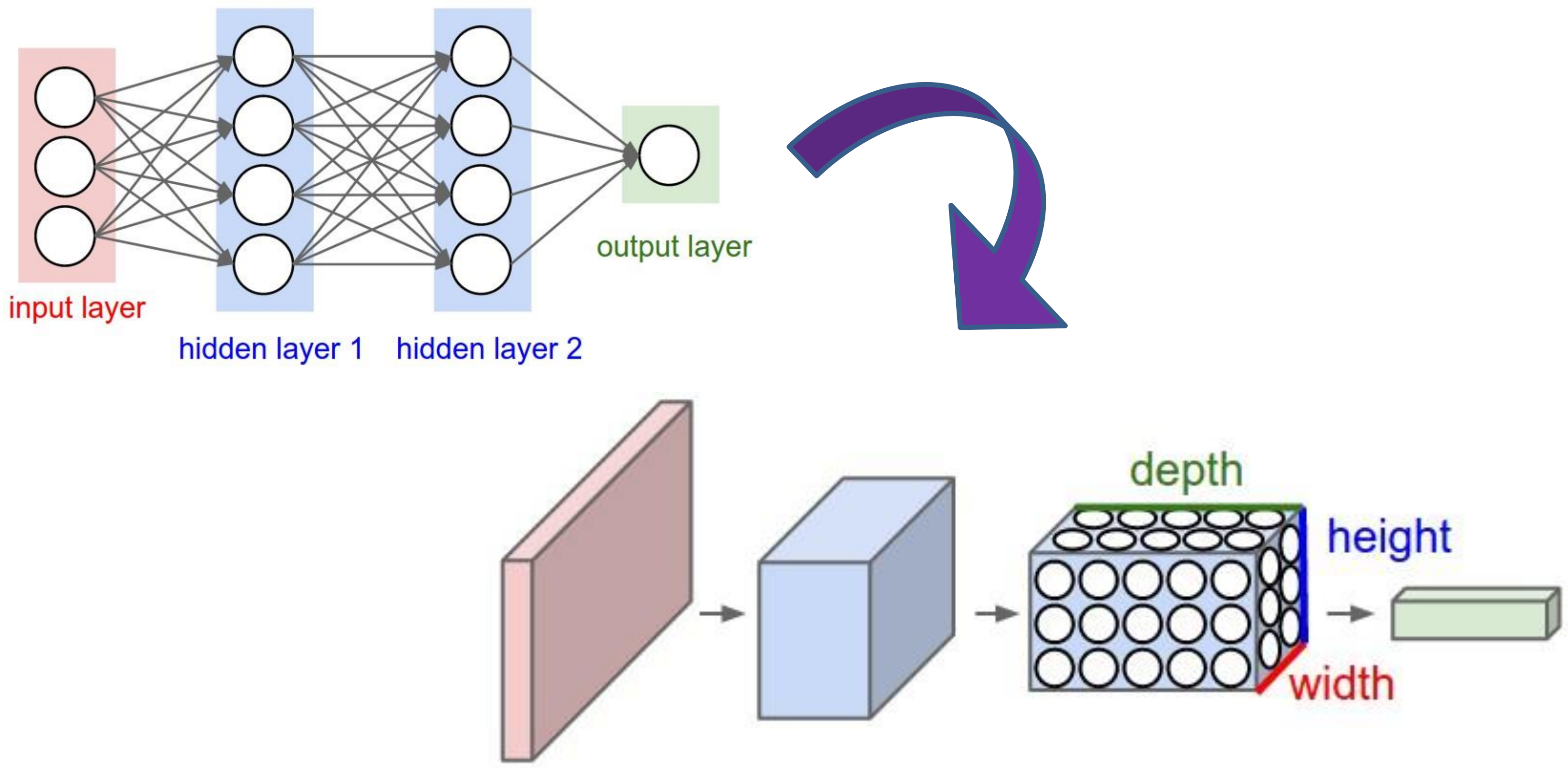
  - The Implementation of this algorithm involves the introduction of additional variables $R\{a_j\}$ $R\{z_j\}$ $R\{\delta_j\}$ for the hidden units and $R\{\delta_k\}$ $R\{y_k\}$ for the output units.
  - For each input pattern, the values of these quantities can be found.

Outlines

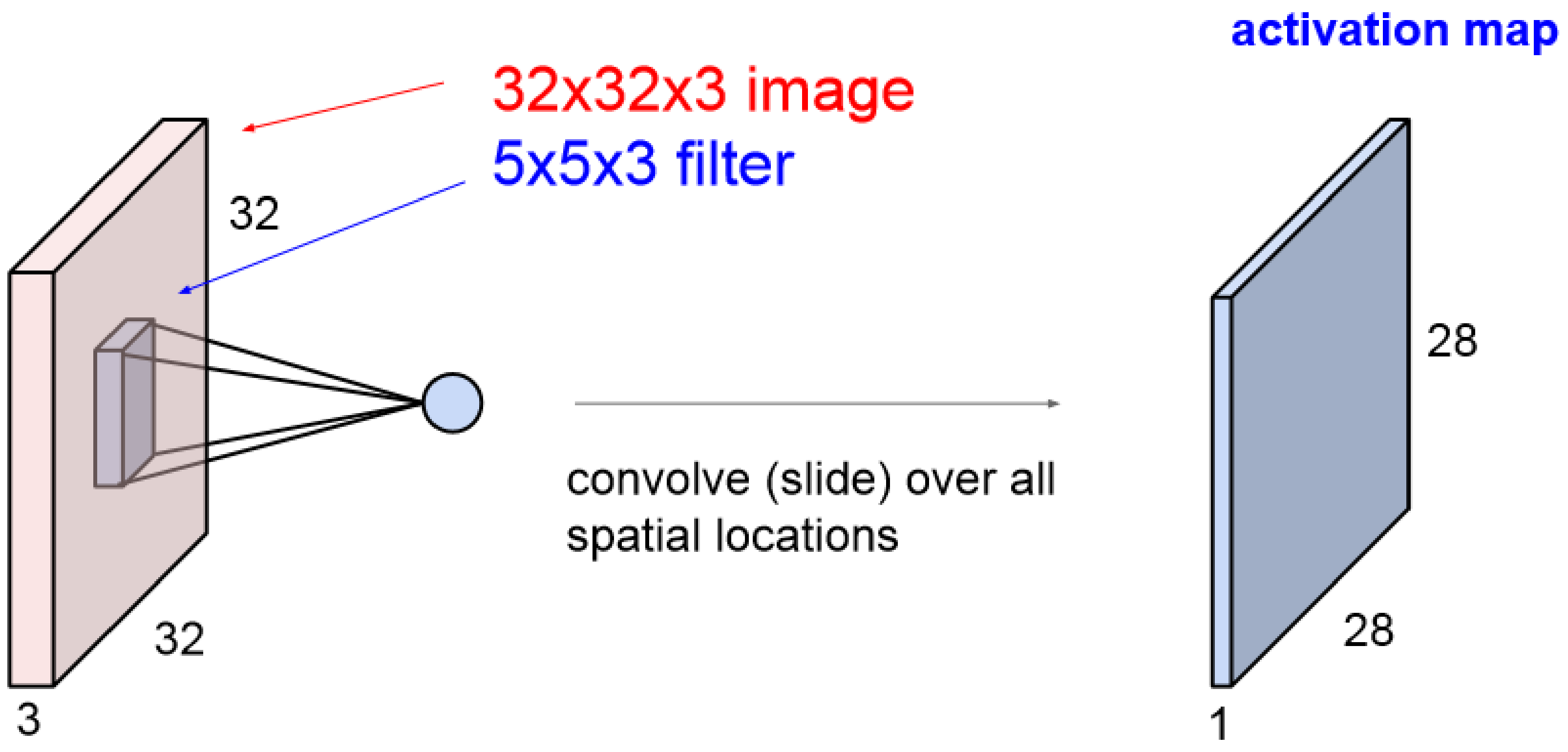
- Feedforward Network Functions
- Network Training
- Error Backpropagation
- Jacobian Matrix
- Hessian Matrix
- CNN and GAN

CNN Architectures

What's the same and difference between a normal full-connected network and convolution network



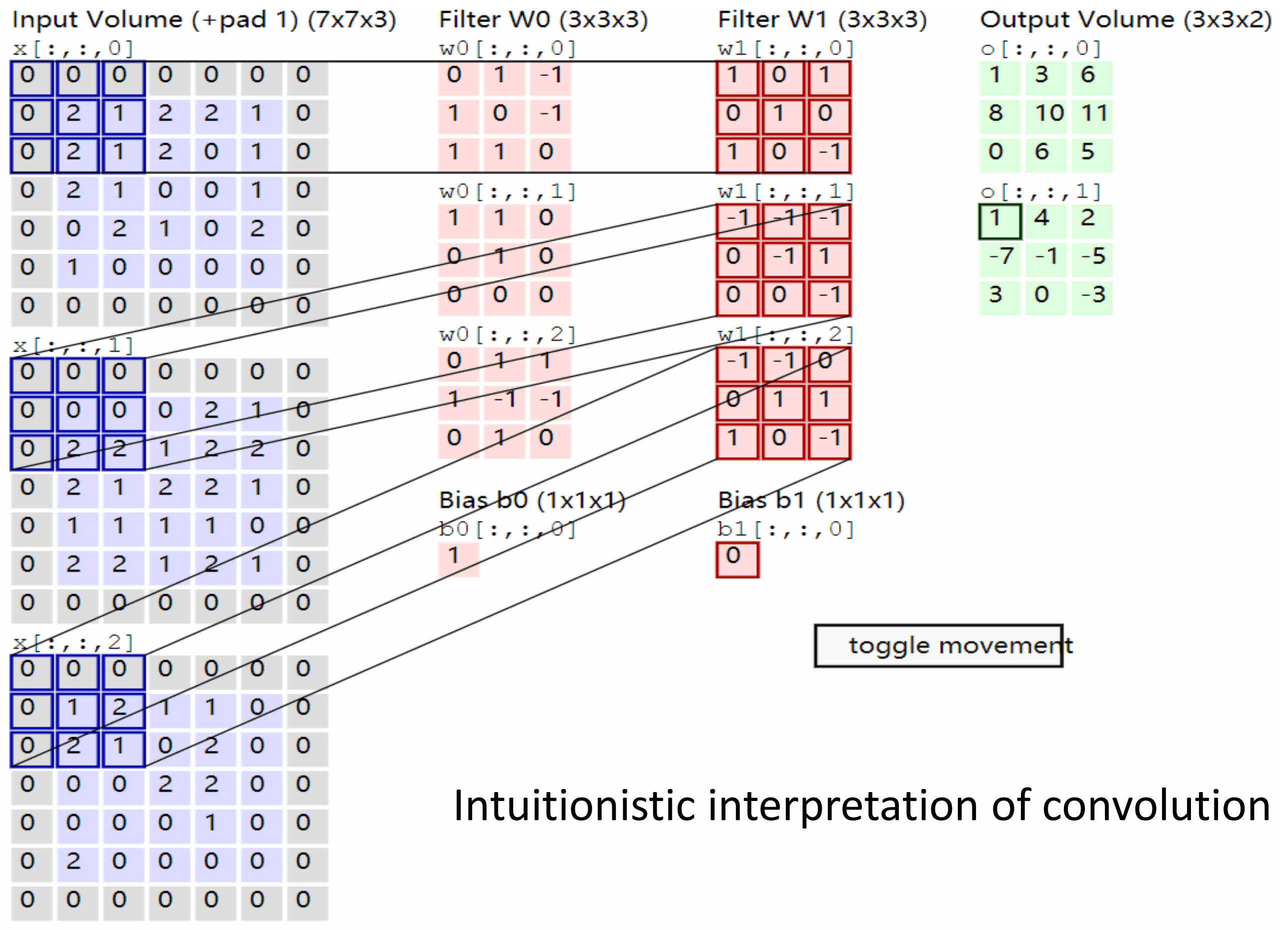
Convolution Layer



CNN Architectures

- Convolution layer
- Pooling layer
- Full-connected layer
- Rectified Linear Units (ReLu) layer

CNN Architectures



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	1	2	2	1	0
0	2	1	2	0	1	0
0	2	1	0	0	1	0
0	0	2	1	0	2	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	0	0	0	2	1	0
0	2	2	1	2	2	0
0	2	1	2	2	1	0
0	1	1	1	1	0	0
0	2	2	1	2	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	1	2	1	1	0	0
0	2	1	0	2	0	0
0	0	0	2	2	0	0
0	0	0	0	1	0	0
0	2	0	0	0	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

0	1	-1
1	0	-1
1	1	0

w0[:,:,1]

1	1	0
0	1	0
0	0	0

w0[:,:,2]

0	1	1
1	-1	-1
0	1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

1	0	1
0	1	0
1	0	-1

w1[:,:,1]

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

w1[:,:,2]

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

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

1	3	6
8	10	11
0	6	5

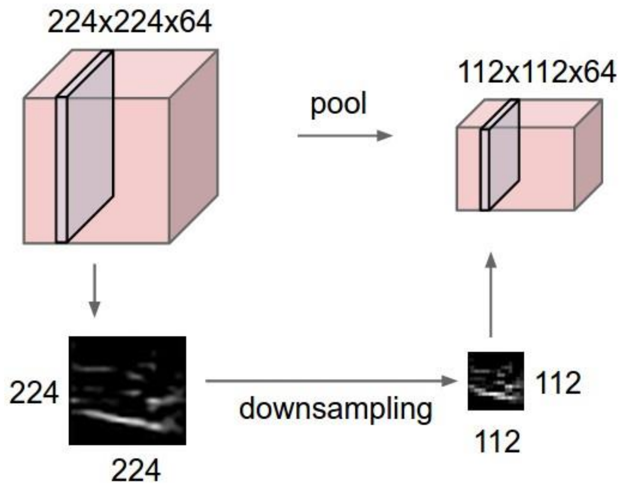
o[:,:,1]

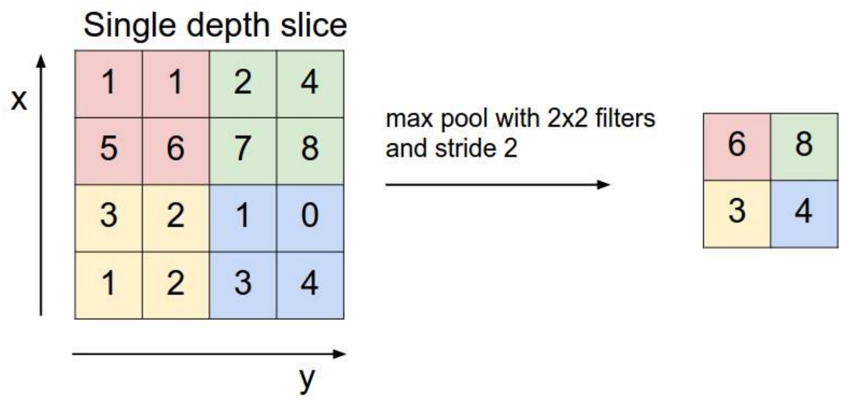
1	4	2
-7	-1	-5
3	0	-3

Intuitionistic interpretation of convolution

CNN Architectures

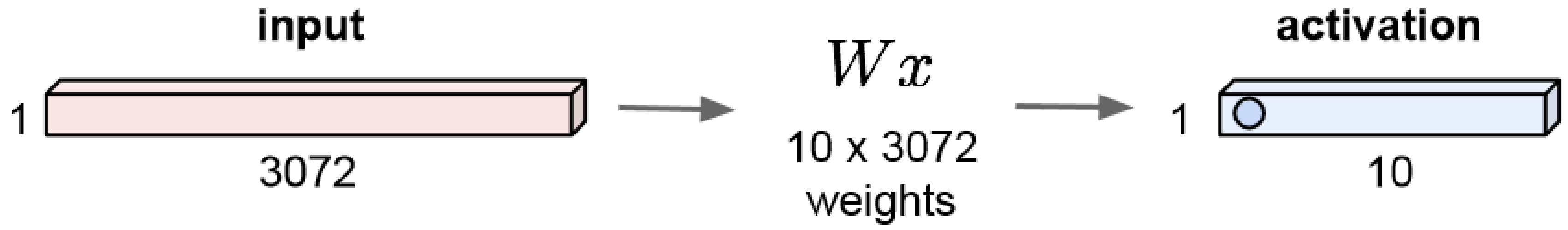
Pooling layer: to reduce the size of the input
Usually using the max value in the block(2*2 most usually) to replace the block itself



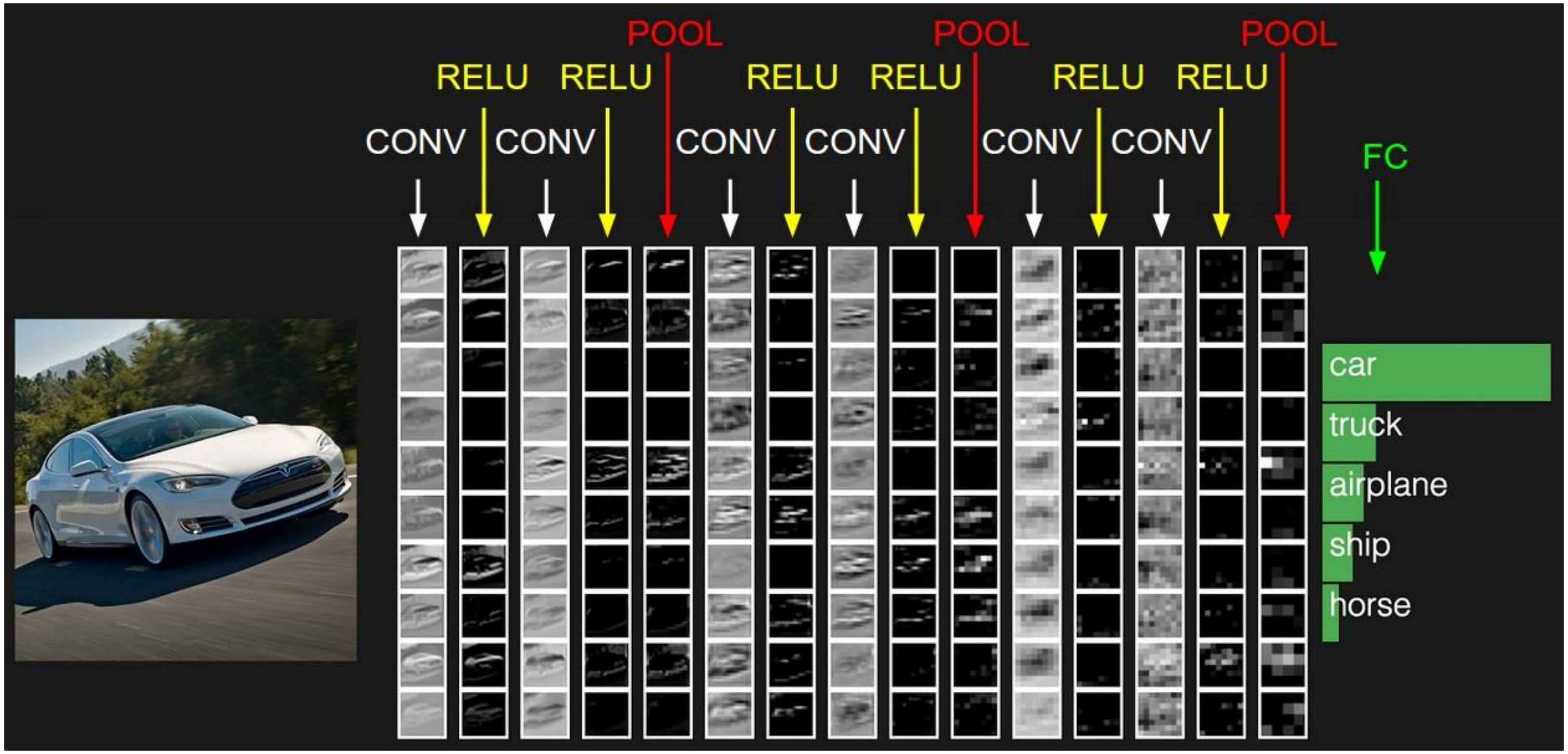


Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



CNN Architectures



CNN

Useful page

CS231n: Convolutional Neural Networks for Visual Recognition:
http://cs231n.github.io/

Neural Networks and Deep Learning
http://neuralnetworksanddeeplearning.com/

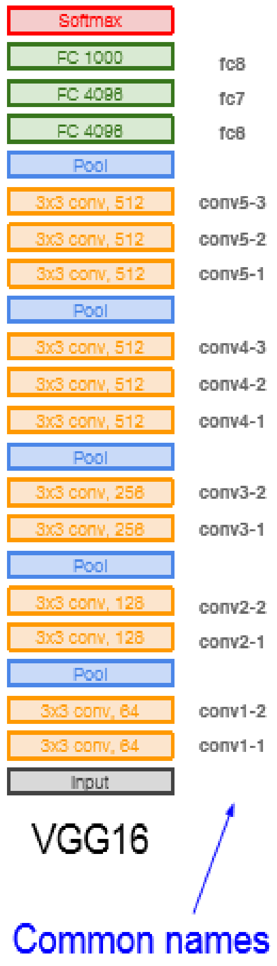
CVPR 2017 Paper interpretation
http://cvmart.net/community/article/detail/69

VGG

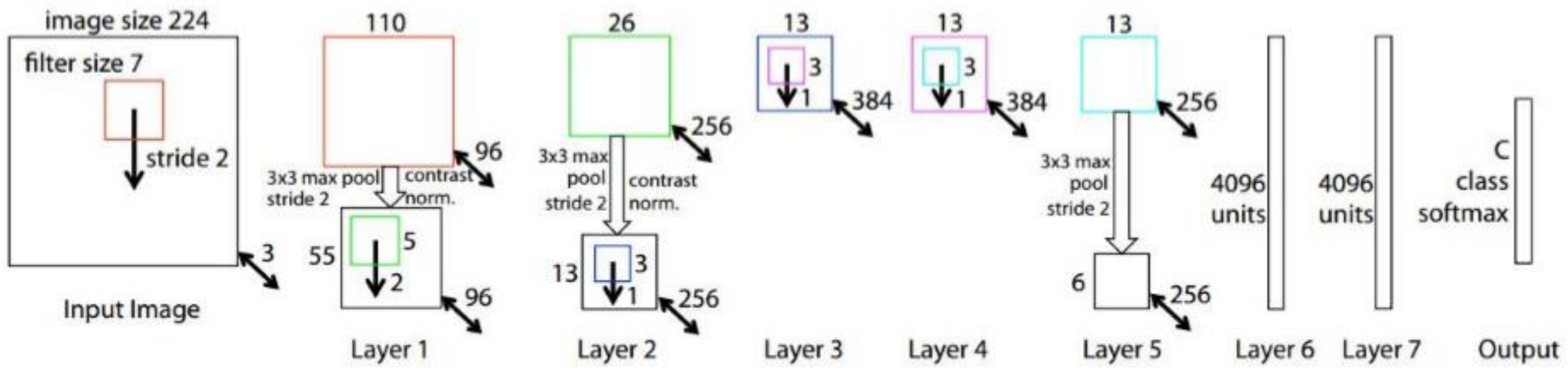
(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

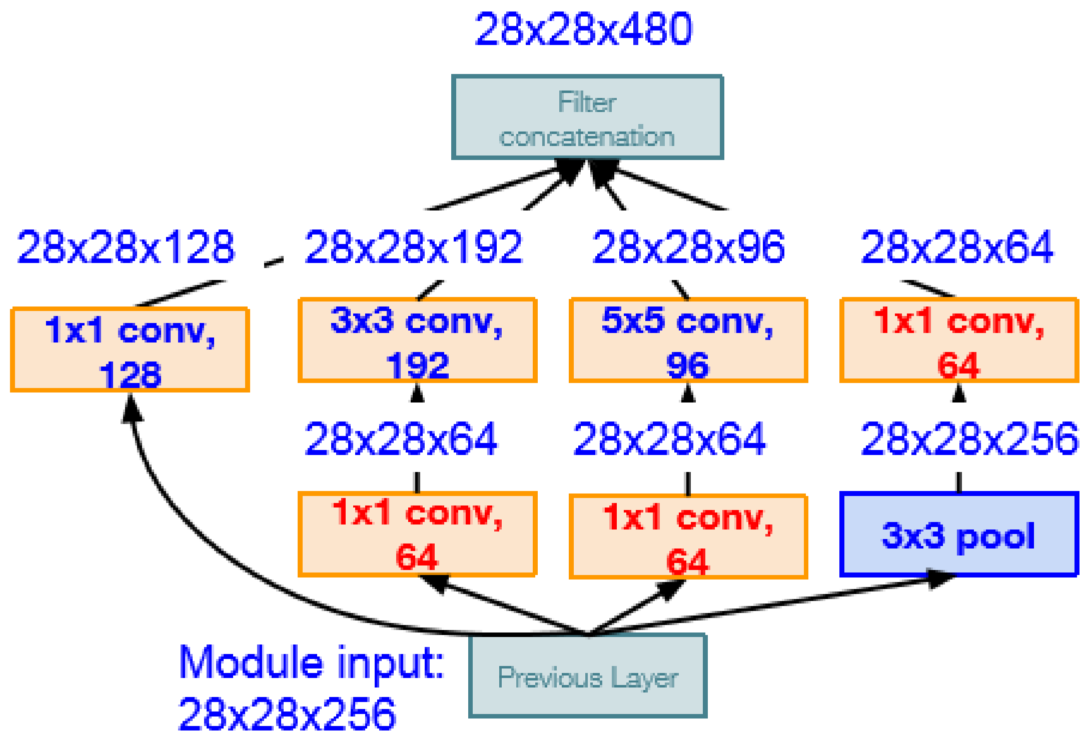


AlexNet/ZFNet



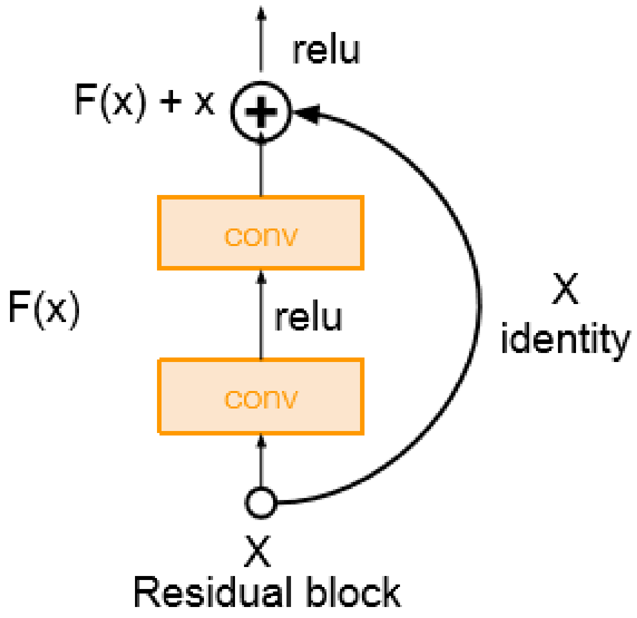
a large, deep convolutional neural network to classify the 1.3 million high-resolution images in the LSVRC-2010 ImageNet training set into the 1000 different classes
http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

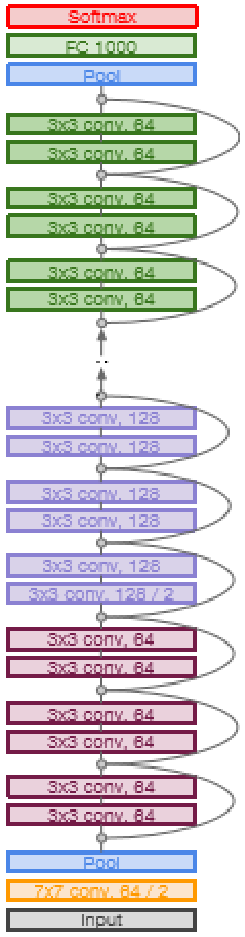
GoogleNet



Inception Module reduced a huge number of parameters in the network (4M, compared to AlexNet with 60M). using Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. https://arxiv.org/pdf/1409.4842.pdf

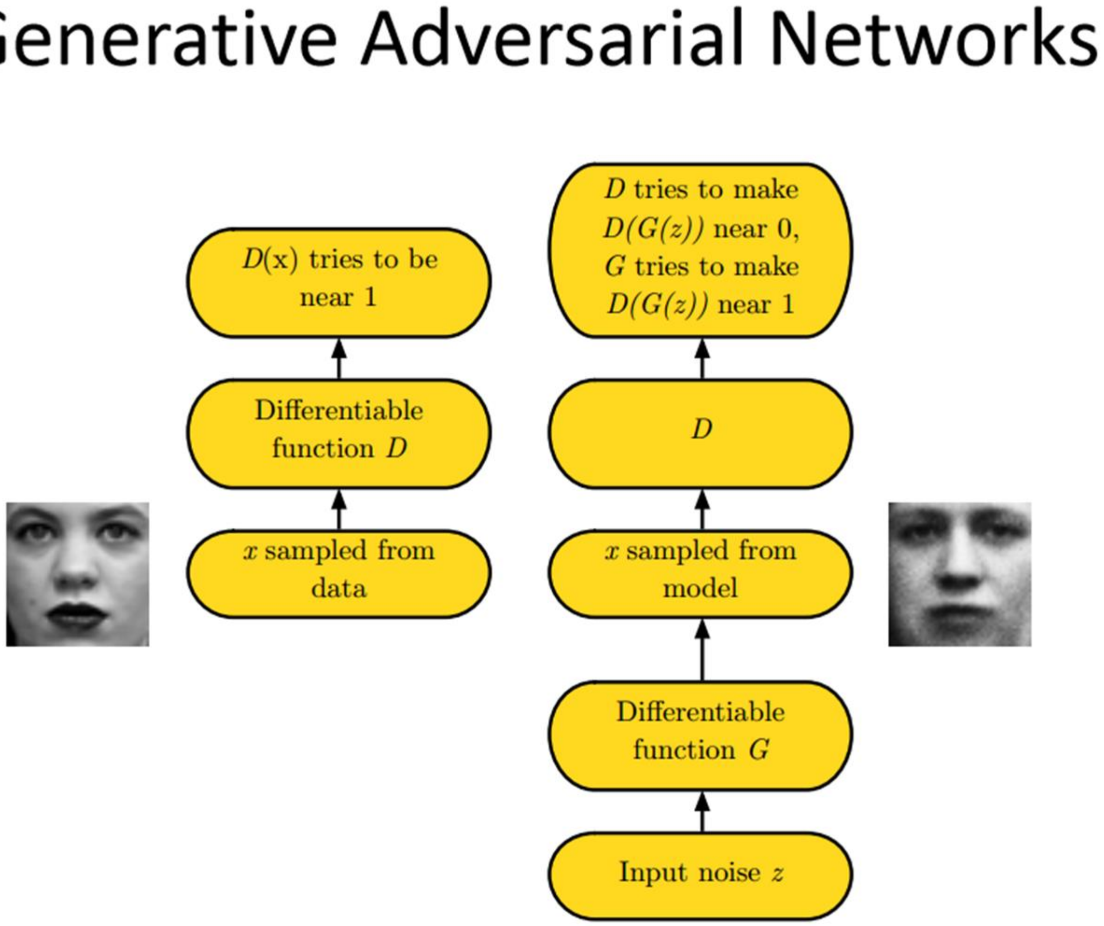
ResNet





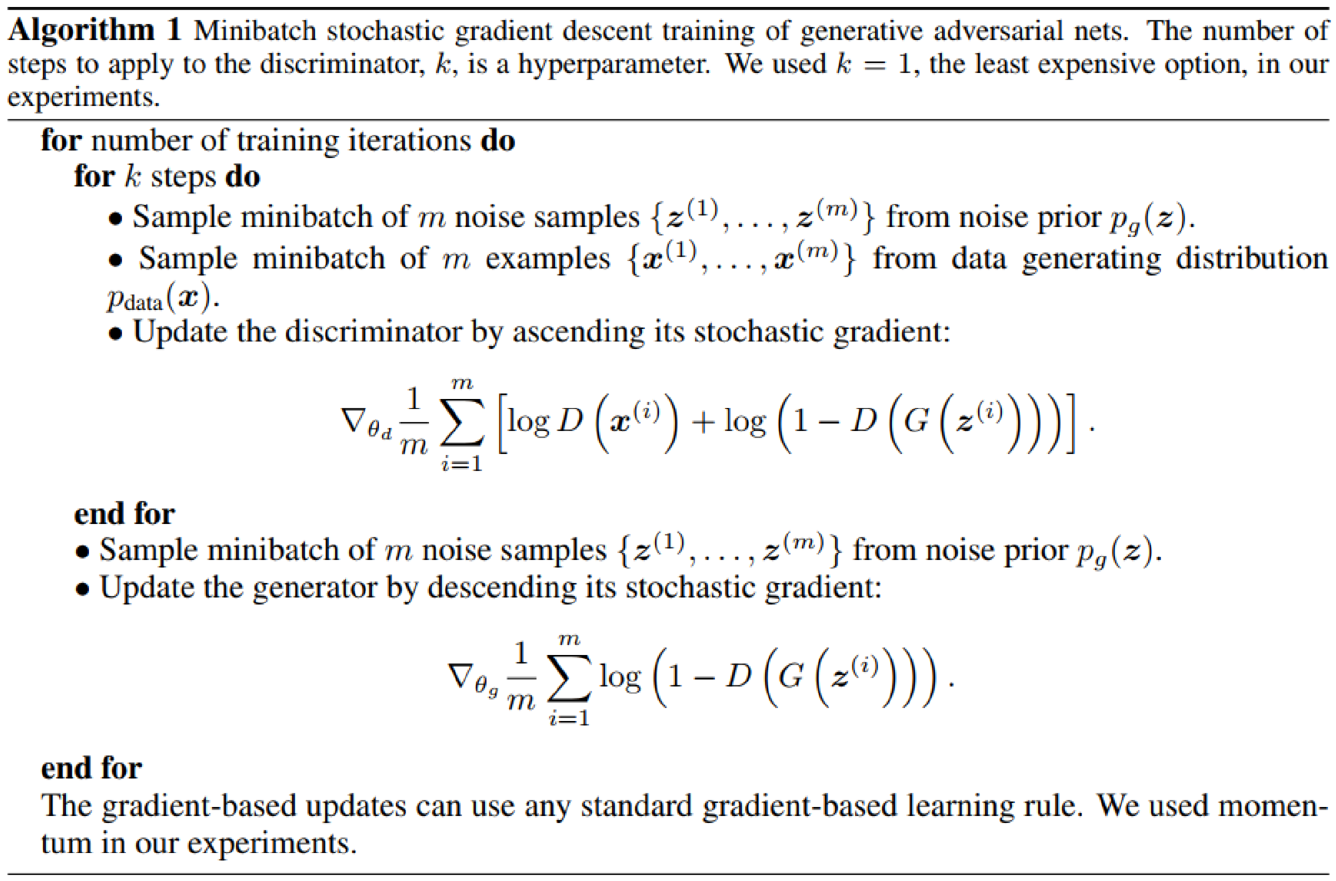
Deep Residual Learning for Image Recognition
https://arxiv.org/pdf/1512.03385.pdf

GAN



Generative Adversarial Networks
D(x) tries to be near 1
Differentiable function D
x sampled from data
D tries to make $D(G(z))$ near 0, G tries to make $D(G(z))$ near 1
D
x sampled from model
Differentiable function G
Input noise z

GAN



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
        • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
        • Update the discriminator by ascending its stochastic gradient:
            $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$
    end for
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:
        $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$
end for
```

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Generative Adversarial Nets
https://arxiv.org/pdf/1406.2661.pdf

HW5

NN Fundamentals: 5.1   5.2   5.4   5.9

Hessian: 5.16 5.19

Extensions: 5.34 5.37 5.38 5.39