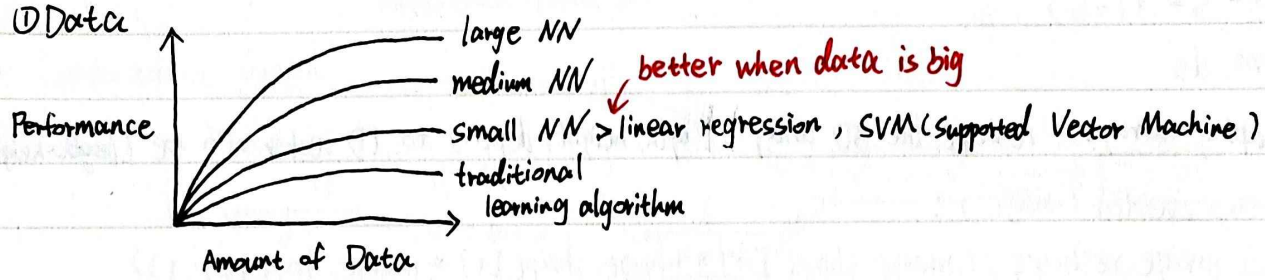
2019

Deep Learning.ai Summary

1- Neural Network and Deep Learning

1.1 Deep Learning is taking off because:

① Data



② Computation

GPUs, powerful CPUs, Distributed Computing, ASICs

③ Algorithm

Creative algorithms has appeared that changed the way NN works

Ex: RELU is much better than SIGMOID

1.2 Logistic Regression

① A most popular binary classification technique.

② We have an input x (image) and we want to know the probability that the image belongs to class 1 (i.e. a cat). For a given x vector, the output will be:

$y = w^T x + b$ * w is n_x dimensional vector, b is a real number

To let the result y make sense as a probability between 0 and 1, Using sigmoid function

$\hat{y} = \sigma(w^T x + b)$ $\quad \sigma(z) = \frac{1}{1+e^{-z}}$ if $z \to \infty$, $e^{z} \to 0 \Rightarrow \sigma(z) = 1$; if $z \to -\infty$, $e^{z} \to \infty \Rightarrow \sigma(z) = 0$

③ Logistic Regression Cost Function

To train the parameters w and b, we need a cost function

$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ * Problem: non-convex, resulting in multiple local optima

a convex function: $L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$ * This is a loss function that measures a single training example, to measure the entire training set, we need a cost function:

$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$

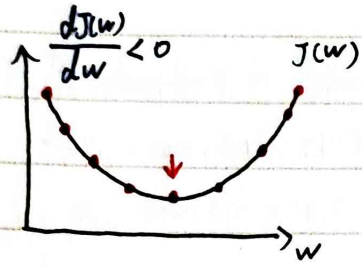
1.3. Gradiend Descent

To train / learn w and b in such a way that the cost function is minimized

Steps: ① Initialize w and b (usually "0")

② Take a step in the steepest downhill direction

③ Repeat step ② until global optimum is achieved



The updated equations for the parameters of logistic regression are:

$$w := w - \alpha \frac{dJ(w,b)}{dw} \qquad b := b - \alpha \frac{dJ(w,b)}{db}$$

* α is the learning rate that controls how big a step we should take after each iteration

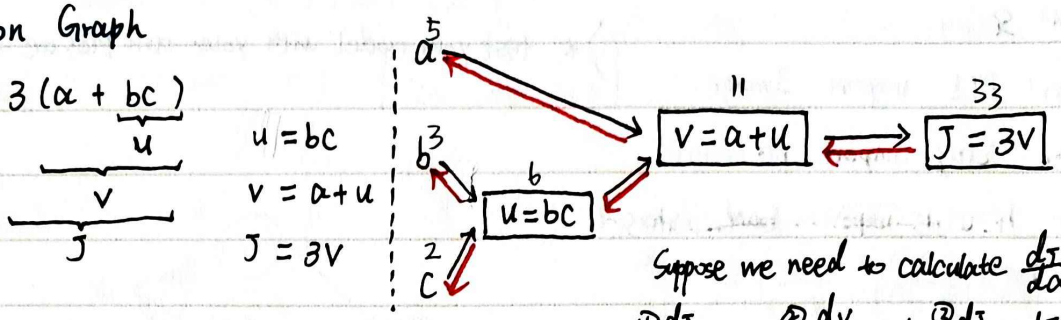
1.4. Computation Graph

$J(a,b,c) = 3(a + bc)$

$u = bc$

$v = a + u$

$J = 3v$



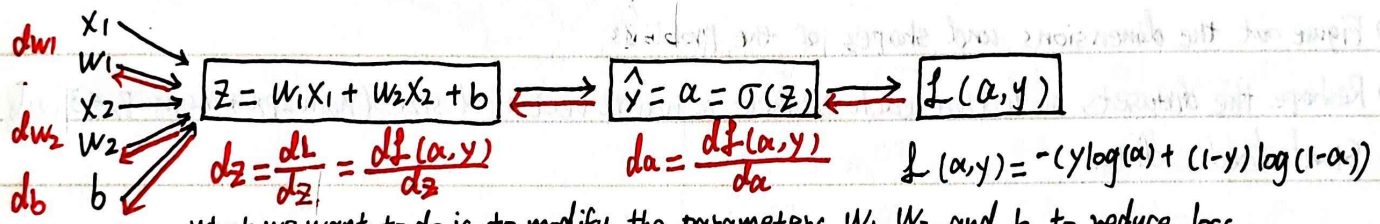
(chain rule)

Suppose we need to calculate $\frac{dJ}{da}$:

① $\frac{dJ}{dv} = 3$ ② $\frac{dv}{da} = 1$ ③ $\frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da} = 3$

1.5. Logistic Regression Gradient Descent (a single training example)

For two features (x_1, x_2), the computation graph:



what we want to do is to modify the parameters w_1, w_2 and b to reduce loss

Steps to calculate dw_1: ① $da = \frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

(Similarly, we can calculate dw_2 and b)

② $dz = \frac{dL}{da} \times \frac{da}{dz} = \left[-\frac{y}{a} + \frac{1-y}{1-a}\right] \times [a(1-a)] = a - y$

③ $dw_1 = \left[\frac{dL}{da} \times \frac{da}{dz}\right] \times \frac{dz}{dw_1} = (a-y) \times \frac{dz}{dw_1}$

$w_1 := w_1 - \alpha dw_1 \qquad w_2 := w_2 - \alpha dw_2 \qquad b := b - \alpha db$

1.6 Gradient Descent on 'm' Examples

Using logistic regression in Python would be slow because you have to run the loop 'm' times to get the output

To get rid of 'for' loops in codes, we use vectorization

① $Z = \underbrace{np.dot(W.T, X)}_{W^T X} + b$ * X contains the features for all the training examples while W is the coefficient matrix for these examples

② $A = np.exp(-Z)$ * A is the sigmoid of Z

③ $dZ = A - Y$ * Calculate the loss and then use backpropagation to minimize the loss

④ $dw = np.dot(X, dZ.T)/m$

$db = dZ.sum()/m$

$w = w - \alpha dw$

$b = b - \alpha db$

* Finally, calculate the derivative of the parameters and update them.

1.7. Broadcasting in Python

- obj.sum(axis=0) sums the columns

  obj.sum(axis=1) sums the rows

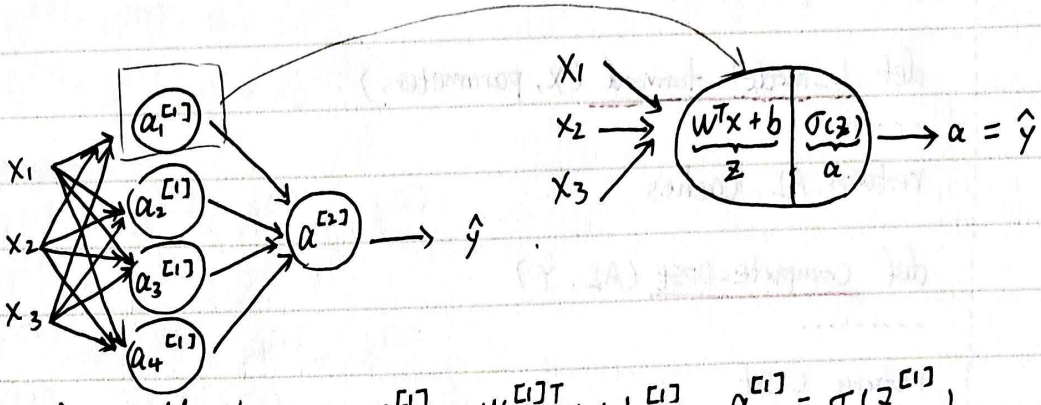
General principle: (m, n) $\overset{+}{-}$ (1, n) $\Rightarrow$ (m, n)

matrix $\overset{\times}{/}$ (m, 1) $\Rightarrow$ (m, n)

Example: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$

1.8 Neural Network

2 layers



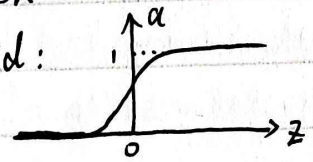
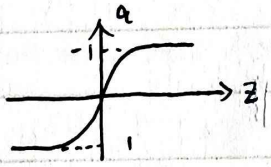
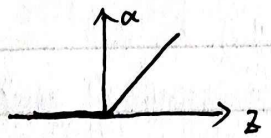
The equations for the first hidden layer

$$\begin{cases} z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, & a_1^{[1]} = \sigma(z_1^{[1]}) \\ z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, & a_2^{[1]} = \sigma(z_2^{[1]}) \\ z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, & a_3^{[1]} = \sigma(z_3^{[1]}) \\ z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, & a_4^{[1]} = \sigma(z_4^{[1]}) \end{cases}$$

$$\left.\begin{aligned} z^{[1]} &= w^{[1]} x + b^{[1]} \\ A^{[1]} &= \sigma(z^{[1]}) \\ z^{[2]} &= w^{[2]} x + b^{[2]} \\ A^{[2]} &= \sigma(z^{[2]}) \end{aligned}\right\}$$ The vectorized form for calculating the output

* this will significantly reduce the computation time

1.9. Activation Function

		Pros	Cons
① Sigmoid: $a = \frac{1}{1+e^{-z}}$		Used in the output layer for binary classification	If z is either very large or very small, the gradient of the derivative or the slope of this function becomes very small
② tanh $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$		Better than Sigmoid ∵ effectively center your data	
③ ReLu $a = \max(0, z)$		Update parameters faster as slope is 1 when $z > 0$	slope = 0 when $x < 0$

✓ (most time choose softmax)

* Softmax function : multiclass logistic regression

Sigmoid function : two-class logistic regression

2.1

Gradient Descent for NN:

Steps: Repeat:

Compute predictions $(y'^{(i)}, i=1, \cdots m)$

Get derivatives: $dW^{[1]}, db^{[1]}, dW^{[2]}, db^{[2]}$

Update: $W^{[1]} = W^{[1]} - \alpha * dW^{[1]}$

$b^{[1]} = b^{[1]} - \alpha * db^{[1]}$

$W^{[2]} = W^{[2]} - \alpha * dW^{[2]}$

$b^{[2]} = b^{[2]} - \alpha * db^{[2]}$

Forward propagation:

$$Z^{[1]} = W^{[1]} * A^{[0]} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} * A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

Backpropagation:

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = (dZ^{[2]} * A^{[1]}.T)/m$$
$$db^{[2]} = \text{Sum}(dZ^{[2]})/m$$
$$dZ^{[1]} = (W^{[2]}.T * dZ^{[2]}) * g'^{[1]}(Z^{[1]})$$
$$dW^{[1]} = (dZ^{[1]} * A^{[0]}.T)/m$$
$$db^{[1]} = \text{Sum}(dZ^{[1]})/m$$

2.2

Random Initialization

$W^{[1]} = np.random.randn((2,2)) * 0.01$ * Should NOT initialize to 0

$b^{[1]} = np.zero((2,1))$

2.3. Forward Propagation in a Deep Neural Network

$Z^{[l]} = W^{[l]} A^{[l-1]} + B^{[l]}$

$A^{[l]} = g^{[l]}(Z^{[l]})$

* have been vectorized for 'm' training examples

2.4. The generalized form of dimensions of W, b and their derivatives:

$W^{[l]} = (n^{[l]}, n^{[l-1]})$

$b^{[l]} = (n^{[l]}, 1)$

$dW^{[l]} = (n^{[l]}, n^{[l-1]})$

$db^{[l]} = (n^{[l]}, 1)$

Dimension of $Z^{[l]}, A^{[l]}, dZ^{[l]}, dA^{[l]} = (n^{[l]}, m)$

* m is the number of training examples

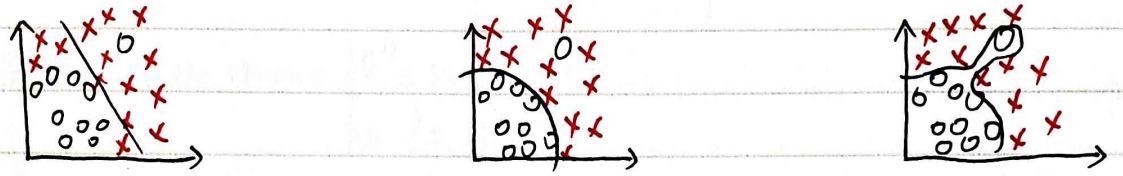
2.5 Vectorized equations for backward propagation in deep NN

$$dZ^{[l]} = dA^{[l]} * g'^{[l]}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} * (dZ^{[l]} * A^{[l-1].T})$$
$$db^{[l]} = \frac{1}{m} * np.sum(dZ^{[l]}, axis=1, keepdims = True)$$
$$dA^{[l-1]} = W^{[l].T} * dZ^{[l]}$$

(Course 2)

3. Improving Neural Network - Hyperparameter Tuning, Regularization, and More

3.1.



high bias | just right | high variance

training set error (%)	1	15	15	0.5
dev set error (%)	11	16	30	1
result	High Variance (overfit)	High Bias	High Variance and High Bias (Underfit)	Low Bias and Low Variance (Reasonable)

Q: Does the model have high bias?

High training error results in high bias.

Solutions: Try bigger networks, try different NN architechtures, train models for a longer period of time

Q: Does the model have high variance?

High dev set error results in high variance

Solutions: Get more data, use regularization, try different NN architechtures

3.2 Regularization

The cost function for a NN: $J(w^{[1]}, b^{[1]}, \dots w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Adding a regularization term: $J(w^{[1]}, b^{[1]}, \dots w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{i=1}^{L} \|w^{[l]}\|^2$

* L_2 regularization $\quad * \|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{i=1}^{n^{[l]}} (w_{ij}^{[l]})^2$

Regularized form of update equations:

$$dw^{[l]} = (\text{from backpropagation}) + \frac{\lambda}{m} w^{[l]}$$
$$w^{[l]} = w^{[l]} - \alpha[dw^{[l]} + \frac{\lambda}{m} w^{[l]}]$$

⇒ * That is why L_2 regularization is also known as weight decay

Other regularization methods:

Dropout Regularization only use during training time, both during forward and backward propagation

Data Augmentation

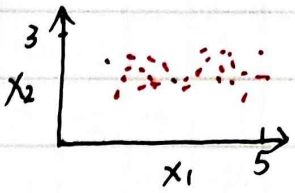
Early Stopping

3.3. Setting up your optimization Problem

① Normalizing inputs

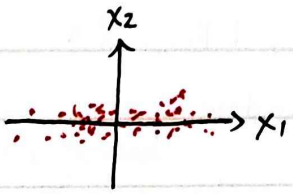
What we want to do is to:

Ex: Changing the 2 input features with scatter plot looks like:



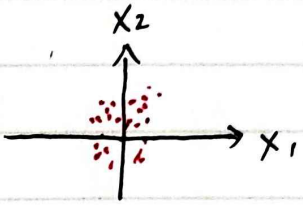
① Subtract the mean from the input features:

$$\begin{cases} \mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ x := x - \mu \end{cases}$$



② normalize the variance:

$$\begin{cases} \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2 \\ x \mathrel{/}= \sigma^2 \end{cases}$$



Normalizing the inputs makes the cost function symmetric. This makes it easier for the gradient descent algorithm to find the global minima more quickly

3.4. How to initialize weights in deep NN to avoid vanishing / exploding gradients?

Initialize weights randomly.

Reason: Break symmetry, making sure different hidden units learn different patterns

3.5. Optimization Algorithm

1). Mini-Batch Gradient Descent (make faster)

For a large training set, typical mini-batch sizes are: 64, 128, 256, 512

2). Exponentially weighted averages

$$V_t = [\beta * V_{(t-1)} + (1-\beta) * \theta_t] \,/\, (1-\beta_t)$$

($1-\beta_t$: a bias correction term)

* This takes a lot less memory as we are overwriting the previous values

3). Gradient Descent with Momentum (reduce oscillations of Mini-Batch gradient descent)

Momentum takes into account the past gradients to smooth out the update.

The stored 'direction' of the previous gradients is the exponentially weighted average of the gradient on the previous steps

* Choose of β: 0.9 but the optimal β should be tuned by several values

4). Adam

① Most fast

② Low memory requirements

③ Work well even with little tuning of hyperparameters (except α)

Convolutional Neural Network

1. Edge Detection Example 灰度图，每一格代表灰度值

$3\times1+1\times1+2\times1+0\times0+5\times0+7\times0+1\times(-1)+8\times(-1)+2\times(-1)=-5$

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

6×6

*

filter

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

3×3

=

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

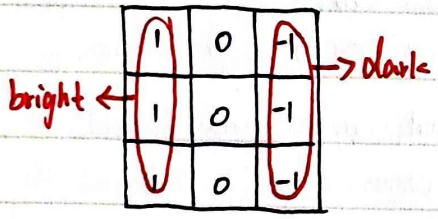
4×4

Convolution, Not multiplication

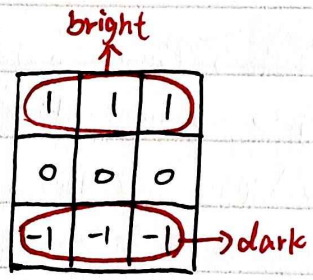
Python : conv-forward

tensorflow: tf.nn.conv2d

2. Vertical and Horizontal Edge Detection



vertical



Horizontal

1	0	-1
2	0	-2
1	0	-1

Sobel filter Adv: put a little more weight to the central row, which makes it more robust

3	0	-3
10	0	-10
3	0	-3

Scharr filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Just learn them using back propagation

3. Padding : to solve two problems :

① Shrink output

② Throw away a lot of information from the edges of the image.

Padding: from 6×6 to 8×8

Input : $n \times n$	Padding=p →	Input : $n \times n$
Filter : $f \times f$		Filter : $f \times f$
Output : $(n-f+1) \times (n-f+1)$		Output : $(n+2p-f+1) \times (n+2p-f+1)$

Valid convolutions : no padding

Same convolutions : the output size is the same as the input size

$$\Rightarrow n+2p-f+1 = n \quad \Rightarrow \quad p = \frac{f-1}{2}$$

* f is usually odd

4. Strided Convolutions

Suppose we choose a stride of 2. While convoluting through the image, we will take two steps - both in the horizontal and vertical directions separately, the dimensions for stride s will be:

• Input : $n \times n$	Stride : s	
• Padding : p	Filter size : $f \times f$	Output : $[(n+2p-f)/s+1] \times [(n+2p-f)/s+1]$

Stride helps to reduce the size of the image

5. Convolutions over volume

① Instead of 2-D image, we have a 3-D image of shape $6 \times 6 \times 3$

- Input : $6 \times 6 \times \underbrace{3}_{\text{channel}}$
- Output : $3 \times 3 \times 3$

* The number of channels in input and output should be the same

② Instead of a single filter, we can use multiple filters as well.

Ex: use two $3 \times 3 \times 3$ filters, then the output should be : $4 \times 4 \times 2$

Input : $6 \times 6 \times 3$

Generalized dimensions :

Input : $n \times n \times n_c$	* n_c : num of channels
Filter : $f \times f \times n_c$	* n_c' : num of filters
Padding : p	
Stride : s	
Output : $[(n+2p-f)/s+1] \times [(n+2p-f)/s+1] \times n_c'$	

6. One layer of a Convolutional network

• No matter how big the image is, the parameters only depend on the filter size.

Summary of notation :

If layer l is a convolution layer :

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = Stride

$n_c^{[l]}$ = num of filters

Input : $n_H^{[l-1]} \times n_w^{[l-1]} \times n_c^{[l-1]}$

Output : $n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

$$n_{H/w}^{[l]} = \frac{n_{H/w}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$$

Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

Bias: $n_c^{[l]} \rightarrow (1, 1, 1, n_c^{[l]})$

when using mini-batch gradient descent

↓ num of training examples

$A^{[l]} \rightarrow m \times n_H^{[l]} \times n_w^{[l]} \times n_c^{[l]}$

Pooling Layers

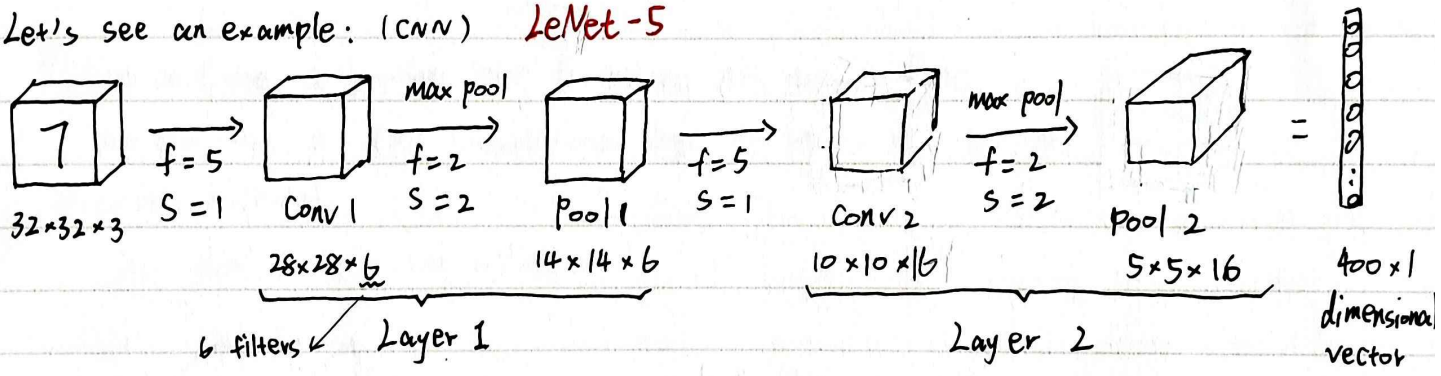
To reduce the size of the inputs and hence speed up the computation

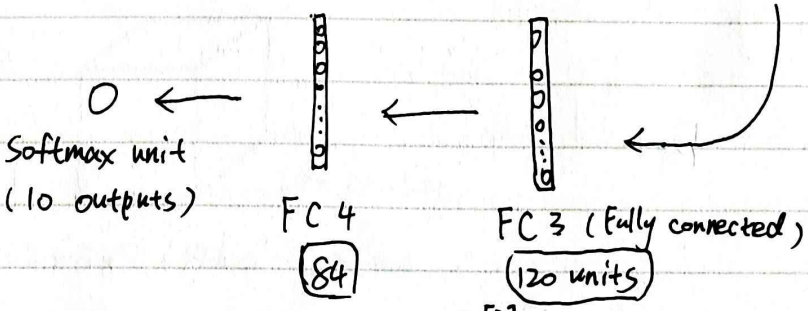
① Max pooling ② Average pooling

Why Convolutions?

1. Parameter sharing → speed up the training of the model
2. Sparsity of connections

Let's see an example: (CNN) LeNet-5





$W^{[3]}$ (120, 400)

$b^{[3]}$ (120)

* Choose of hyperparameters:

Do some research on literature

* Deeper NN, n_w & n_H ↓, while n_c ↑

Three classic architectures:

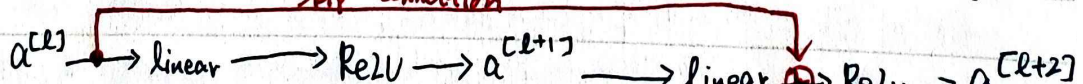
- LeNet-5
- AlexNet
- VGG-16

(much larger network; simplify hyperparameters)

More advanced architectures:

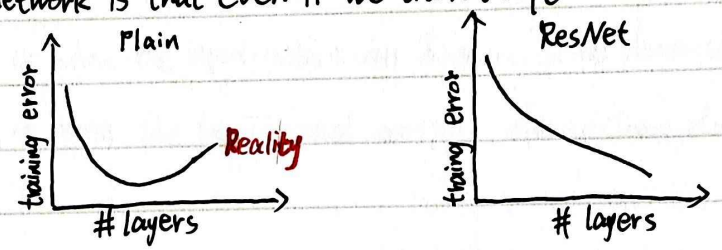
- ResNet

ResNet: Using residual blocks. Skip connections where we take activations from one layer and feed it to another layer that is even more deeper in the network. ⇒ Solving the problem of vanishing and exploding gradients.



$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

The benefit of training a residual network is that even if we train deeper networks, the training errors does not increase



To calculate activation using a residual block: $a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

$$a^{[l+2]} = g(W^{[l+2]} * a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

Now, say $W^{[l+2]} = 0$ and the bias $b^{[l+2]}$ is also 0, then:

$$a^{[l+2]} = g(a^{[l]})$$

Networks in Networks and 1×1 Convolutions

- The basic idea of using 1×1 convolution is to reduce the number of channels from the image

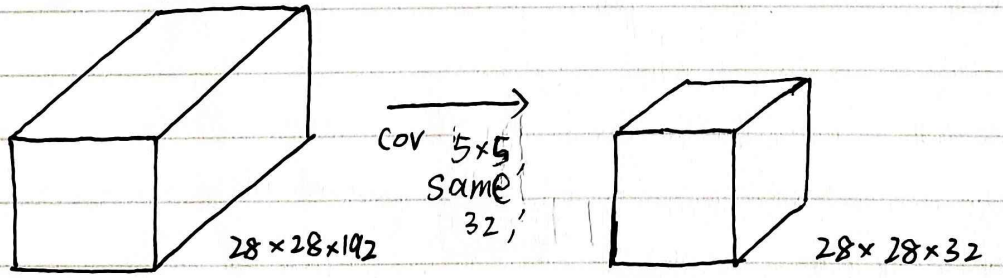
We use a pooling layer to shrink the height and width of the image

☆ You can use a pooling layer to reduce n_H, n_W, and n_C.

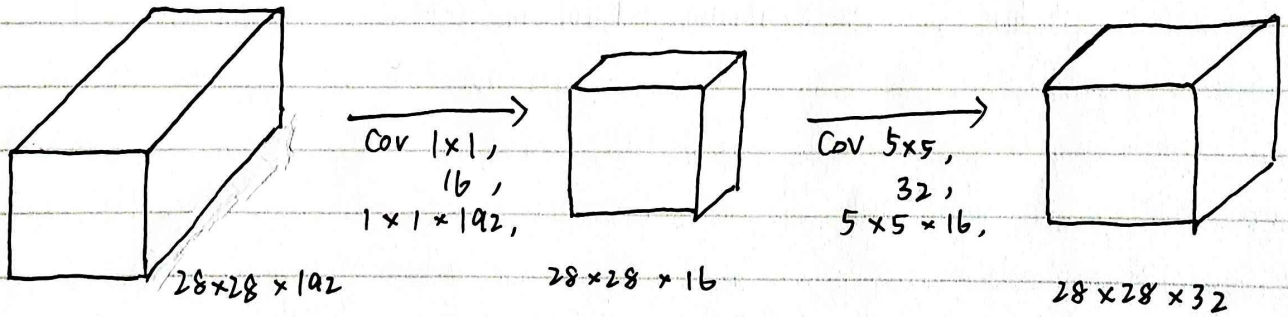
You can use a 1×1 convolutional layer to reduce n_C, but not n_H, n_W

Inception Network

Key idea: significant reduction



Number of multipliers = $28 \times 28 \times 32 \times 5 \times 5 \times 192 = 120$ million



Number of multiplies for first convolution: $28 \times 28 \times 16 \times 1 \times 1 \times 192 = 2.4$ million

for second convolution: $28 \times 28 \times 32 \times 5 \times 5 \times 16 = 10$ million

Total: 2.4 million + 10 million = 12.4 million

GANs

1. Generative Algorithm vs. Discriminative Algorithm

eg. Assuming this email is spam, how likely are these features?

⇒ Predict features given a certain label

⇒ Model the distribution of individual classes

(MNIST)

Measure the probability that an email is spam given the words it contains

⇒ Predict a label given certain features

⇒ learn the boundary between classes

2. One neural network, called the generator, generates new data instances, while the discriminator, evaluates them for authenticity.

The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake

3. For MNIST, the discriminator network is a standard convolutional network that can categorize the images fed to it, a binomial classifier labeling images as real or fake.

The generator is an inverse convolutional network.

A standard convolutional classifier takes an image and downsamples it to produce a probability, the generator takes a vector of random noise and upsamples it to an image.

4. Autoencoders encode input data as vectors. They create a hidden, or compressed, representation of the raw data. They are useful in dimensionality reduction.

5. Steps: ① Given a label, they predict the associated features (Naive Bayes)

② Given a hidden representation, they predict the associated features (VAE, GAN)

③ Given some features, they predict the rest (inpainting, imputation)

6. Tips: 1). When you train the discriminator, hold the generator values constant

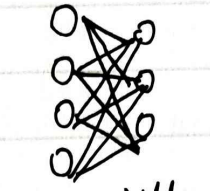
When you train the generator, hold the discriminator values constant

2). The two NNs must have a similar "skill level"

RBMs (Restricted Boltzmann Machines) — Geoffrey Hinton

1. Definition: ~ is an algorithm useful for dimensionality reduction, classification, regression, collaborative filtering, feature learning and topic modeling

2. Structure: RBMs are shallow, two-layer NNs



Entropy: a common way to measure impurity

Entropy $H(x)$ of a random variable X:

$$H(X) = -\sum_{i=1}^{n} P(X=i) \log_2 P(X=i)$$

*n: # of possible values for X

- $H(x)$ is the expected number of bits needed to encode a randomly drawn value of X
- Most efficient code assigns $-\log_2 P(X=i)$ bits to encode the message $X=i$

Information Gain:

1). We want to determine which attribute in a given set of training feature vectors is most useful for discriminating between the classes to be learned

2). Information gain tells us how important a given attribute of the feature vectors is

3) We will use it to decide the ordering of attributes in the nodes of a decision tree.

$$\text{Gain}(S, A) = I_S(A, Y) = H_S(Y) - H_S(Y|A) \quad \text{ii)}$$

the same as: $I(X, Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$ i)

i). Information gain is the mutual information between input attribute A and target variable Y

ii). Information gain is the expected reduction in entropy of target variable Y for data sample S, due to sorting on variable A.

Using Information Gain to Construct a Decision Tree:

Choose the attribute A with highest information gain for the full training set at the root of the tree

Summary: $\text{Info Gain}(X, A) = H(X) - H(X|A)$

$$= H(X) - \sum_{v \in \text{Values}(A)} \underbrace{\frac{|\{x \in X \mid x_A = v\}|}{|X|}}_{\text{fraction of instances with value } v \text{ in attribute } A} \times \underbrace{H(\{x \in X \mid x_A = v\})}_{\text{entropy of those instances}}$$

Entropy:

$$H(X) = -\sum_{c \in \text{Classes}} \underbrace{\frac{|\{x \in X \mid \text{class}(x) = c\}|}{|X|} \log_2 \frac{|\{x \in X \mid \text{class}(x) = c\}|}{|X|}}_{\text{fraction of instances of class } C}$$

Attributes with Many Values

Use GainRatio( ) instead

$$\text{GainRatio}(X, A) = \frac{\text{InfoGain}(X, A)}{\text{SplitInformation}(X, A)}$$

$$\text{SplitInformation}(X, A) = -\sum_{v \in \text{values}(A)} \frac{|X_v|}{|X|} \log_2 \frac{|X_v|}{|X|}$$

* Where X_v is a subset of X for which A has value V.

Evaluation

1. Classification Matrics: $\text{accuracy} = \frac{\#\text{ correct predictions}}{\#\text{ test instances}}$

$$\text{error} = 1 - \text{accuracy} = \frac{\#\text{ incorrect predictions}}{\#\text{ test instances}}$$

2. Confusion Matrix

(Actual class) \ (Predicted class)	Yes	No
Yes	TP	FN
No	FP	TN

$$\text{accuracy} = \frac{TP + TN}{P + N}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

Probability that a randomly selected result is relevant

$$\text{recall} = \frac{TP}{TP + FN}$$

Probability that a randomly selected relevant doc is retrieved

Linear Regression

- Cost Function $J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2$

  Fit by solving $\min_\theta J(\theta)$

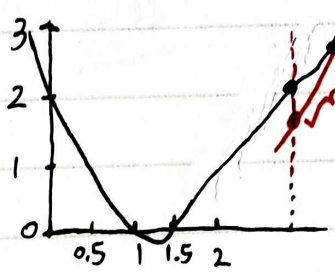
- Gradiend Descent :

  1). Initialize θ

  2). Repeat until convergence

  $\theta_j \leftarrow \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$

  Simultaneous update for $j = 0 \dots d$



α : learning rate (small e.g. $\alpha = .05$)

For linear regression : $\frac{\partial}{\partial\theta_j}J(\theta) = \frac{\partial}{\partial\theta_j}\frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2$

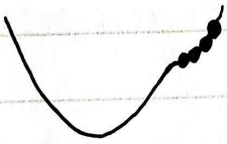
$= \dots\dots$

$= \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{k=0}^{d}\theta_k x_k^{(i)} - y^{(i)}\right)x_j^{(i)}$

Hence, repeat until convergence : $\theta_j \leftarrow \theta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$ simultaneous update for $j = 0 \dots d$

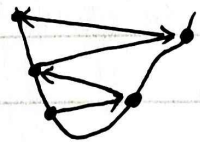
3). Choosing α

α too small



- Slow convergence

α too large



← Increasing value for $J(\theta)$

- May overshoot the minimum
- May fail to converge, even diverge

* To see if gradient descent is working, print out $J(\theta)$ each iteration

The value should decrease each iteration, if it doesn't, adjust α

- Basic Linear Model : $h_\theta(x) = \sum_{j=0}^{d}\theta_j x_j$

  Generalized Linear Model : $h_\theta(x) = \sum_{j=0}^{d}\theta_j \phi_j(x)$

  $\phi_j(x)$: basis function allows use of linear regression techniques to fit non-linear datasets

  ① Polynomial basis functions : $\phi_j(x) = x^j$

  ② Gaussian basis functions : $\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\}$

  ③ Sigmoidal : $\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right)$

  * $\sigma(\alpha) = \frac{1}{1+\exp(-\alpha)}$

Vectorization :

1. Benefits: ① More compact equations
② Faster Code (using optimized matrix libraries)

2. Consider our model for n instances: $h(x^{(i)}) = \sum_{j=0}^{d} \theta_j x_j^{(i)}$

Let: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$ $X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_d^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(i)} & \cdots & x_d^{(i)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & \cdots & x_d^{(n)} \end{bmatrix}$

dimensions: $\mathbb{R}^{(d+1)\times 1}$ $\mathbb{R}^{n\times(d+1)}$

3. For the linear regression cost function:

$$J(\theta) = \frac{1}{2n}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2n}\sum_{i=1}^{n}(\theta^T x^{(i)} - y^{(i)})^2 = \frac{1}{2n}(X\theta - y)^T(X\theta - y)$$

$\mathbb{R}^{n\times(d+1)}$ $\mathbb{R}^{(d+1)\times 1}$ → $\mathbb{R}^{1\times n}$ $\mathbb{R}^{n\times 1}$

4. Closed Form Solution

- Instead of using GD, solve for optimal θ analytically
- The solution is when $\frac{\partial}{\partial\theta}J(\theta) = 0$

* let $y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$

Derivation:

$$J(\theta) = \frac{1}{2n}(X\theta - y)^T(X\theta - y) \propto ((X\theta)^T - y^T)(X\theta - y)$$

(Constant can be deleted)

$$= (X\theta)^T(X\theta) - (X\theta)^T y - y^T(X\theta) + y^T y = \theta^T X^T X\theta - 2y^T X\theta + y^T y$$

($(X\theta)^T y = y^T(X\theta)$)

Take derivative and set equal to 0, then solve for θ

$$\frac{\partial}{\partial\theta}(\theta^T X^T X\theta - 2y^T X\theta + y^T y) = 0 \qquad * \frac{\partial(X^T X)}{\partial X} = 2X$$

$$X^T X\theta - X^T y = 0$$

$$\theta = (X^T X)^{-1} X^T y$$

Gradiend Descent	Closed Form Solution
• Requires multiple iterations	• Non-iterative
• Need to choose α	• No need for α
• Works well when n is large	• Slow if n is large
• Can support incremental learning	• 只适用于 linear-regression

Why we choose GD ?

① Can apply GD. to any model, can be done automatically by software packages (Theano, Tensor Flow)

② Much faster than solving the linear system

Improving Learning :

① Feature Scaling : Ensure that feature have similar scales ⇒ GD converges much faster

② Feature Standardization : Rescale features to have zero mean and unit variance

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

μ_j : the mean of feature j $\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)}$

s_j : the SD of feature j

③ Regularization : A method for automatically controlling the complexity of the learned hypothesis

Idea: Penalize for large values of θ_j

- Can incorporate into the cost function
- Works well when we have a lot of features, each that contributes a bit to predicting the label

Linear regression objective function

$$J(\theta) = \underbrace{\frac{1}{2n}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2}_{\text{model fit to data}} + \underbrace{\frac{\lambda}{2}\sum_{j=1}^{d}\theta_j^2}_{\text{regularization}}$$

regularization :
1). λ is the regularization parameter ($\lambda \geq 0$)
2). No regularization on θ_0

Understanding: $\sum_{j=1}^{d}\theta_j^2 = \|\theta_{1:d}\|_2^2$ ⇒ the magnitude of the feature coefficient vector

$\sum_{j=1}^{d}(\theta_j - 0)^2 = \|\theta_{1:d} - \vec{0}\|_2^2$ ⇒ L_2 regularization pulls coefficient toward 0

Gradient update : $\frac{\partial}{\partial\theta_j}J(\theta)$ $\theta_j \leftarrow \theta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \underbrace{- \alpha\lambda\theta_j}_{\text{regularization}}$

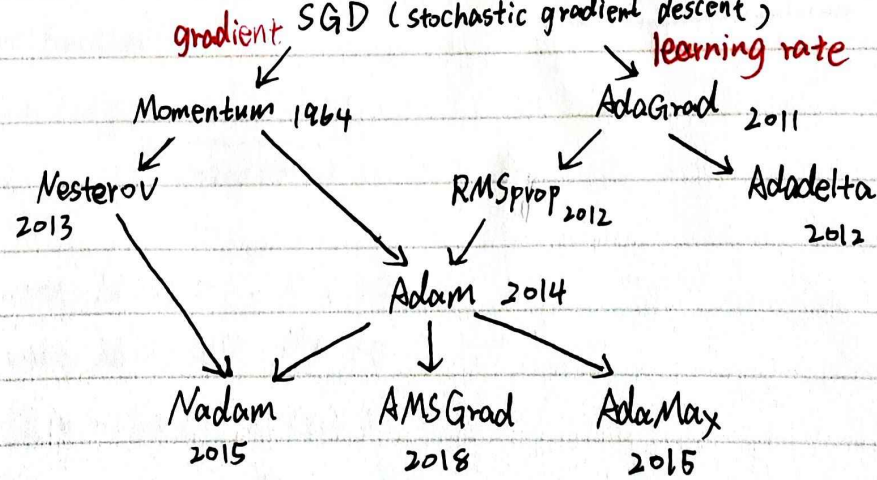
Gradient Descent Optimization

3 main ways : 1). modifying the learning rate α

2). modifying the gradient component $\frac{\partial L}{\partial w}$

3). both

Evolutionary Map:



Support Vector Machines & Kernels

Doing really well with linear decision surfaces

1. Why might predictions be wrong?
   i) True non-determinism
   ii) Partial observability
   iii) Noise in the observation x (Measurement error, instrument limitations)
   iv) Representational bias
   v) Algorithmic bias

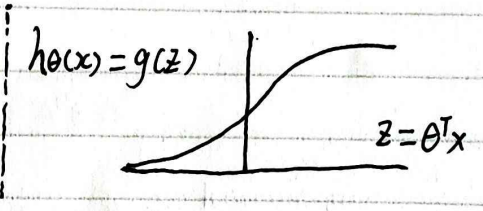
2. Strengths of SVMs:
   i) Good generalization
   ii) Works well with few training instances
   iii) Find globally best model
   iv) Efficient algorithms
   v) Amenable to the kernel trick

3. Why Maximize Margin?
   i). Increasing margin reduces capacity
   ii). If the following holds:
   - H is sufficiently constrained in size
   - and/or the size of the training data set n is large, then low training error is likely to be evidence of low generalization error

   then low training error is likely to be evidence of low generalization error

4. Alternative view of Logistic Regression

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$



The Logistic Regression Objective Function

$$J(\theta) = -\sum_{i=1}^{n}\left[\underbrace{y_i \log h_\theta(x_i)}_{\text{cost}_1(\theta^T x_i)} + \underbrace{(1-y_i)\log(1-h_\theta(x_i))}_{\text{cost}_0(\theta^T x_i)}\right]$$

Intuition: If $y=1$, we want $h_\theta(x) \approx 1$, $\theta^T x \gg 0$

If $y=0$, we want $h_\theta(x) \approx 0$, $\theta^T x \ll 0$

Cost: $-y_i \log h_\theta(x_i) - (1-y_i)\log(1-h_\theta(x_i))$

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}} \qquad z = \theta^T x$$

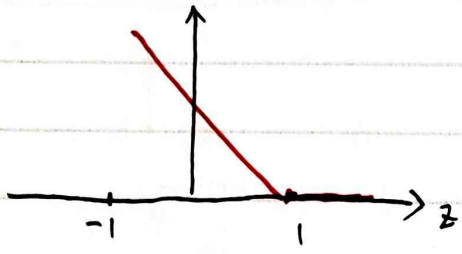
Logistic Regression to SVMs :

Logistic Regression : $\min_\theta -\sum_{i=1}^{n}[y_i \log h_\theta(x_i) + (1-y_i)\log(1-h_\theta(x_i))] + \frac{\lambda}{2}\sum_{j=1}^{d}\theta_j^2$

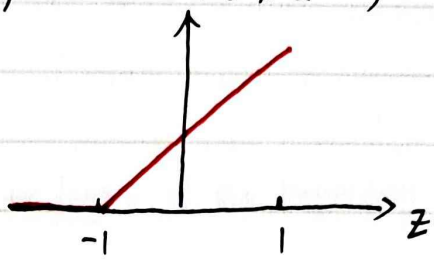
SVMs : $\min_\theta C\sum_{i=1}^{n}[y_i \text{cost}_1(\theta^T x_i) + (1-y_i)\text{cost}_0(\theta^T x_i)] + \frac{1}{2}\sum_{j=1}^{d}\theta_j^2$

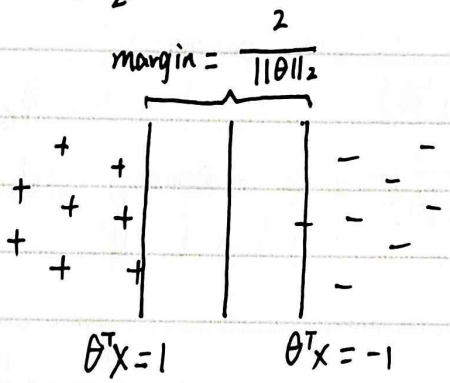
* you can think C as similar to $\frac{1}{\lambda}$

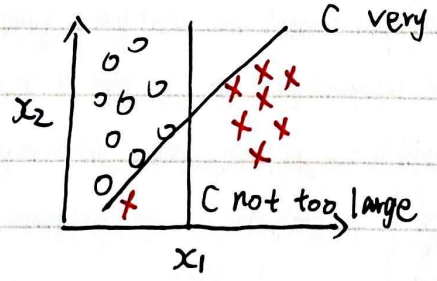
If $y=1$ (want $\theta^T x \geq 1$):



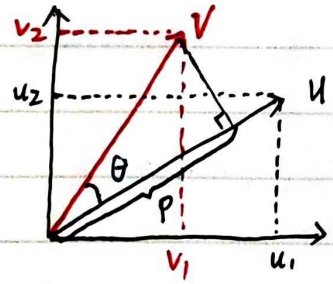
If $y=0$ (want $\theta^T x \leq -1$)







Vector Inner Product



$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$

$\|u\|_2 = \text{length}(u) \in \mathbb{R} = \sqrt{u_1^2 + u_2^2}$

$u^T v = v^T u = u_1 v_1 + u_2 v_2 = \|u\|_2 \|v\|_2 \cos\theta = p\|u\|_2$

Where $p = \|v\|_2 \cos\theta$

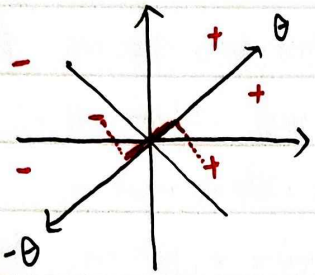
Understanding the Hyperplane

$\min_\theta \frac{1}{2}\sum_{j=1}^{d}\theta_j^2$ s.t. $\theta^T x_i \geq 1$ if $y_i = 1$

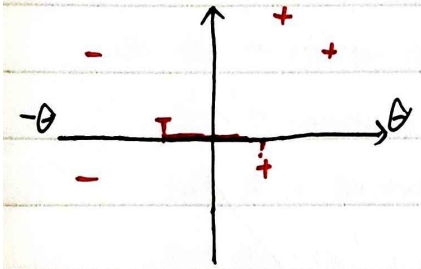
$\theta^T x_i \leq -1$ if $y_i = -1$

Assume $\theta_0 = 0$ so that the hyperplane is centered at the origin, and that $d=2$

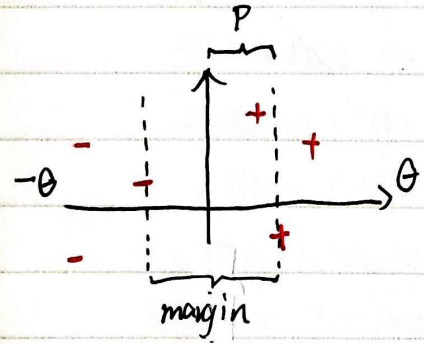
$\theta^T x = \|\theta\|_2 \underbrace{\|x\|_2 \cos\theta}_{p} = p\|\theta\|_2$



Since p is small, therefore $\|\theta\|_2$ must be large to have $p\|\theta\|_2 \geq 1$ (or ≤ -1)



Since p is larger, $\|\theta\|_2$ can be smaller in order to have $p\|\theta\|_2 \geq 1$ (or ≤ -1)



$p\|\theta\|_2 = \pm 1$ · p is the length of the projection of the SVs onto

Therefore, $p = \frac{1}{\|\theta\|_2}$

$\text{margin} = 2p = \frac{2}{\|\theta\|_2}$

Feature mappings : an easy way to use linear regression to learn nonlinear dependencies.

eg. $y = w_3 x^3 + w_2 x^2 + w_1 x + w_0$ ⇒ Polynomial regression

* Algorithmically, polynomial regression is no different from linear regression we define a feature mapping ϕ. $\phi(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$, compute the predictions as $y = w^T \phi(x)$ instead of $w^T x$

Limitations: 1). The features need to be known in advance

2). In high dimensions, the feature representations can get very large

Solution: Using Neural Networks to learn nonlinear predictors directly from the raw inputs.

Generalization error

* Hyperparameter : are values that we can't include in the training procedure itself, but which we need to set using other means. In practice, we tune hyperparameters by partioning the dataset into three different subsets:

1). training set. to train the model

2). validation set. to eliminate the generalization error of each hyperparameter setting

3). test set. - - - - - of the final model

PCA: principal component analysis

- Unsupervised learning algorithm
- A linear model, with a closed-form solution

Def: Choosing a subspace to maximize the projected variance, or minimize the reconstruction error

1). Learning a subspace

Q: How to choose a good subspace S?

Need to choose a vector μ and a $D \times K$ matrix U with orthonormal columns

Set μ to the mean of the data, $\mu = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}$

Two criteria:

① Minimize the reconstruction error $\quad \min \frac{1}{N}\sum_{i=1}^{N} \| x^{(i)} - \tilde{x}^{(i)} \|^2$

② Maximize the variance of the code vectors

$$\max \sum_j \mathrm{Var}(z_j) = \frac{1}{N}\sum_j \sum_i (z_j^{(i)} - \bar{z}_i)^2 = \frac{1}{N}\sum_i \| z^{(i)} - \bar{z} \|^2 = \frac{1}{N}\sum_i \| z^{(i)} \|^2$$

* $\bar{z}$ denotes the mean

These two criteria are equivalent

by unitarity: $\| \tilde{x}^{(i)} - \mu \| = \| U z^{(i)} \| = \| z^{(i)} \|$

by the Pythagorean Theorem: $\underbrace{\frac{1}{N}\sum_{i=1}^{N} \| \tilde{x}^{(i)} - \mu \|^2}_{\text{Projected variance}} + \underbrace{\frac{1}{N}\sum_{i=1}^{N} \| x^{(i)} - \tilde{x}^{(i)} \|^2}_{\text{reconstruction error}} = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \| x^{(i)} - \mu \|^2}_{\text{constant}}$

Stochastic Gradient Descent

mini-batch : size S : a hyperparameter that needs to be set

a reasonable value might be $S = 100$

In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

Typical Strategy: 1). Use a large learning rate early in training so you can get close to the optimum

2). Gradually decay the learning rate to reduce the fluctuations
逐渐地

* By reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly

Generalization

Decomposition of generalization error, or expected loss into bias, variance, and Bayes error.

$$\mathbb{E}[(y-t)^2] = \underbrace{(y^* - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\text{Var}(y)}_{\text{variance}} + \underbrace{\text{Var}(t)}_{\text{Bayes error}}$$

An overly simple model might have high bias and low variance

An overly complex model might have low bias and high variance

Q: How to achieve low bias and low variance?

Sol: ① Data augmentation

② Reduce the number of parameters { 1). Reduce the # of layers or the # of parameters per layer
2). Adding a linear bottleneck layer

③ Weight decay

④ Early Stopping

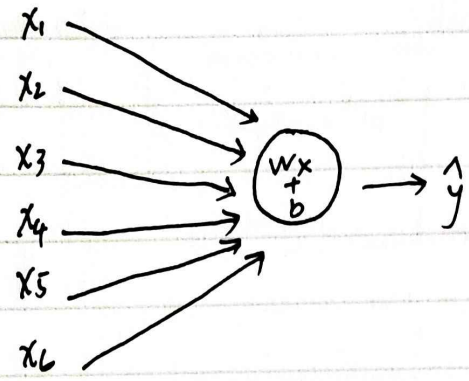
⑤ Ensembles

⑥ Stochastic regularization (e.g. dropout)

Q: Why are neurol networks vulnerable to adversarial examples?

Ex: Linear Regression:

The forward propagation: $\hat{y} = Wx + b$



Network has been trained and converged to:

$W = (1, 3, -1, 2, 2, 3)$, $b = 0$

* $x = \begin{bmatrix} \vdots \end{bmatrix}$ $w = [\cdots\cdots]$

i. $x = \begin{bmatrix} 1 \\ -1 \\ 2 \\ 0 \\ 3 \\ -2 \end{bmatrix}$ $\hat{y} = Wx + b = 1 - 3 - 2 + 0 + 6 - 6 = \boxed{-4}$

(Q): How to change $x \to x^*$ such that $\hat{y}$ changes radically but $x^* \approx x$? a prob of adversarial example

The intuition comes from the derivative

$\frac{\partial \hat{y}}{\partial x}$: the impact on $\hat{y}$ of small changes of x $\quad \frac{\partial \hat{y}}{\partial x} = W^T$

$x^* = x + \varepsilon w^T$ (小变动)

↳ value of perturbation

$\hat{y}^* = Wx^* + b \quad (=0) = Wx + \varepsilon w w^T$

$= Wx + \varepsilon w^2$

always > 0

Ex:

$x^* = x + \varepsilon w^T = \begin{bmatrix} 1 + 0.2 \\ -1 + 0.6 \\ 2 - 0.2 \\ 0 + 0.4 \\ 3 + 0.4 \\ -2 + 0.6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ -0.4 \\ 1.8 \\ 0.4 \\ 3.4 \\ -1.4 \end{bmatrix}$ ($\varepsilon = 0.2$)

$\hat{y}^* = Wx^* = 1.2 - 1.2 - 1.8 + 0.8 + 6.8 - 4.2 = \boxed{1.6}$

* A very slight change in x^* has pushed $\hat{y}$ from -4 to 1.6

Insights: ①: If w is large $\Rightarrow x^* \not\approx x$

↳ sign(w) instead of w^*

②: As x grows in dimensions, the impact of $+\varepsilon \text{sign}(w)$ on $\hat{y}$ increases

Images have high dimensions: $64 \times 64 \times 3$

(Conclusion: The fast way to generate adversarial example: (fast gradient sign method)

$x^* = x + \varepsilon \, \text{sign}(\nabla_x J(W, x, y))$

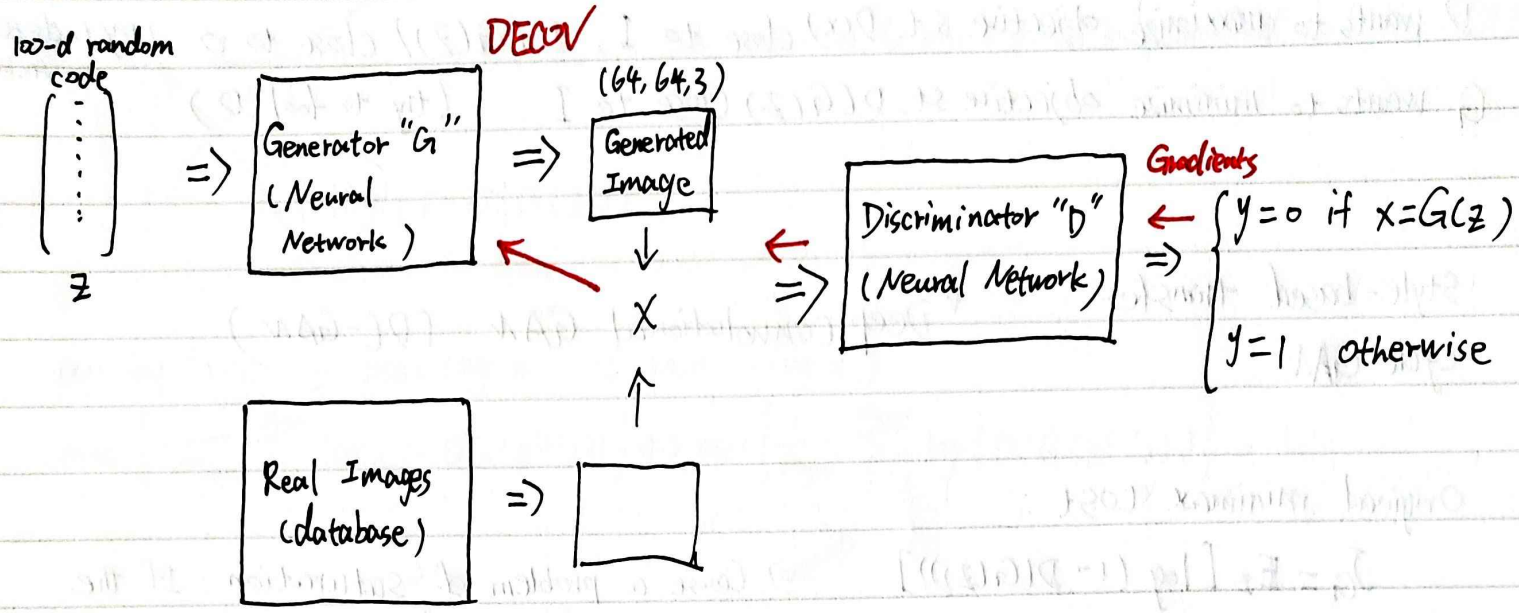
We are linearizing the cost function in the proximity of the parameters

We are pushing the pixel images in one direction that is going to impact highly the output

GANs

Goal: Collect a lot of data, use it to train a model to generate similar data from scratch

Intuition: # of parameters of the model << # of data



- The way we will train the "G" is that we will backpropagate the gradient to the "D" to train the "D", using a binary cross entropy.
  If the image is real, then the generator couldn't get the gradient, because x doesn't depend on z or on the features and parameters of the generator.

- Run Adam simultaneously on two minibatches (true data / generated data)

- Training procedure, we want to minimize: Labels: $\begin{cases} y_{real} \text{ is always } 1 \\ y_{gen} \text{ is always } 0 \end{cases}$

  i): The cost of the "D"

$$J^{(D)} = -\frac{1}{m_{real}} \sum_{i=1}^{m_{real}} y_{real}^{(i)} \cdot \log\left(D(x^{(i)})\right) - \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \left(1 - y_{gen}^{(i)}\right) \cdot \log\left(1 - D(G(z^{(i)}))\right)$$

  Cross-entropy 1: "D" should correctly label real data as 1

  Cross-entropy 2: --- generated data as 0

  We both want the D to correctly identify real data and also correctly identify fake data.

  ii): The cost of the "G"

$$J^{(G)} = -J^{(D)} = \frac{1}{m_{gen}} \sum_{i=1}^{m_{gen}} \log\left(1 - D(G(z^{(i)}))\right)$$

  G should try to fool D: by minimizing the opposite of what D is trying to minimize

We want D to go up to be very good and G to go up at the same time until the equilibrium: a certain point where D will always output one-half like, random probabilities.

Train GAN jointly via minimax game:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim P_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim P(z)} \log \left(1 - D_{\theta_d}(G_{\theta_d}(z))\right) \right]$$

D wants to maximize objective s.t. $D(x)$ close to 1, $D(G(z))$ close to 0 (try to detect fake)

G wants to minimize objective s.t. $D(G(z))$ close to 1 (try to fool D)

Style-based transfer

cycle GAN

* Deep convolutional GAN (DC-GAN)

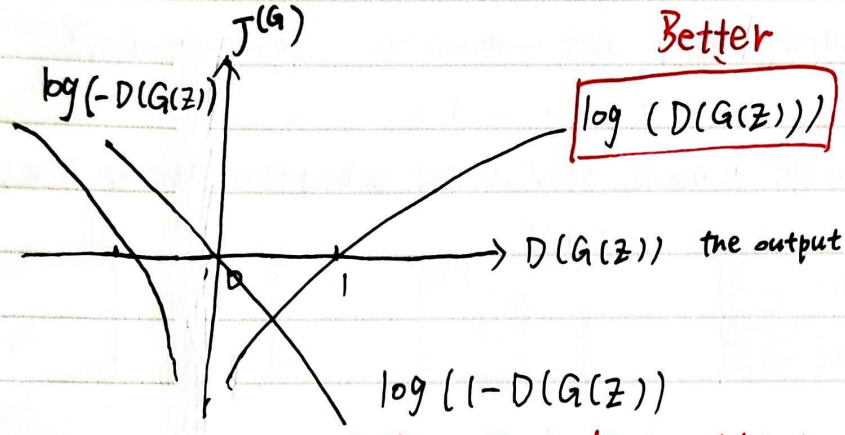
Original minimax cost:

$$J_G = \mathbb{E}_z \left[\log \left(1 - D(G(z))\right) \right]$$

=> Cause a problem of saturation: If the generated sample is really bad, the discriminator's prediction is close to 0, and the generator's cost is flat.

Modified generator cost:

$$J_G = \mathbb{E}_z \left[-\log D(G(z)) \right]$$



Problem: the gradient could be very large at "1"

$$\min \log(1-x) = \max \log x = \min(-\log x)$$

$$\min\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \log(1-D(G(z^{(i)})))\right] \Leftrightarrow \max\left[\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)})))\right]$$

$$\min\left[-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)})))\right]$$

New training procedure, we want to minimize:

$$J^{(D)} = \underbrace{-\frac{1}{m_{real}}\sum_{i=1}^{m_{real}} y^{(i)}_{real} \cdot \log(D(x^{(i)}))}_{\text{Cross-entropy 1}} \underbrace{-\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} (1-y^{(i)}_{gen}) \cdot \log(1-D(G(z^{(i)})))}_{\text{Cross-entropy 2}}$$

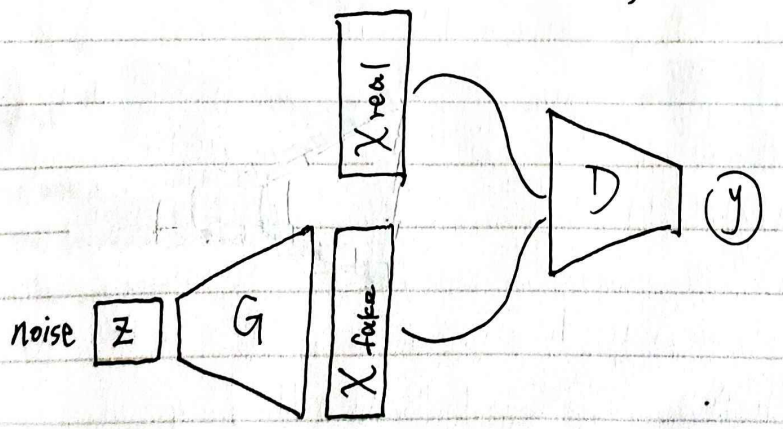
D should correctly label real data as 1 — D ... generated data as 0

$$J^{(G)} = -\frac{1}{m_{gen}}\sum_{i=1}^{m_{gen}} \log(D(G(z^{(i)}))) \Rightarrow$$ G should try to fool D by minimizing this

GANs' training tips:

1). Modification of the cost function
2). Keep D up-to-date with respect to G (k update for D / 1 update for G)
3). Use virtual Batchnorm
4). One-sided label smoothing

```
for num_iterations:
    for k iterations:
        update D
    update G
```

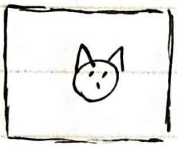
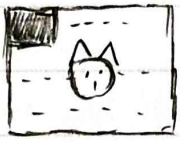


The generator turns noise into an imitation of the data to try to trick the discriminator

The D tries to identify real data from fakes created by the generator

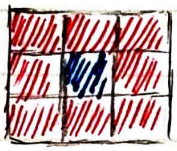
1. Saliency maps: a common technique to quickly visualize what the network is looking at.

2. A method more precise but a little slower: occlusion sensitivity

high confidence on the true class for the correspondiag position of the grey box

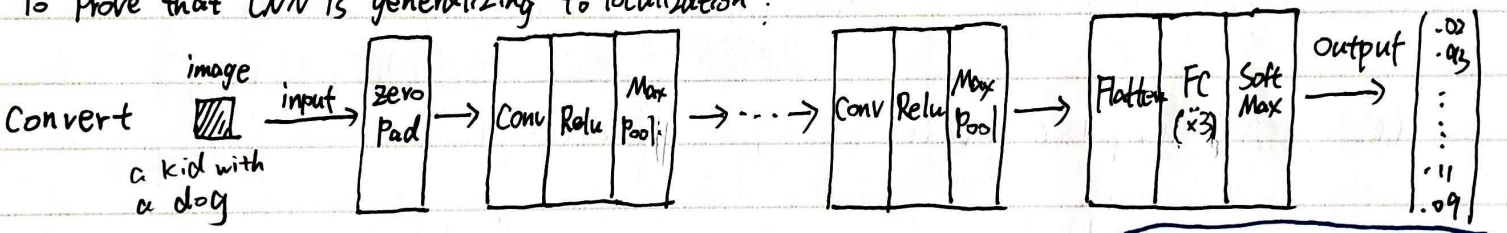
low confidence ...



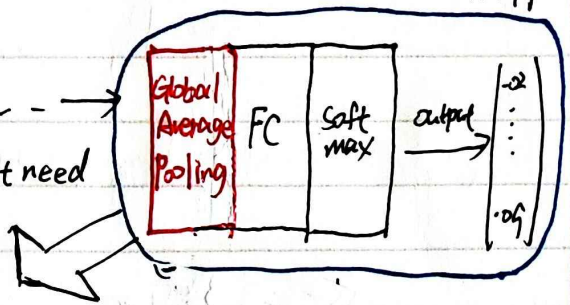
容易有问题，例：若人和狗相拥，难以检测出狗；车表面有图案或字，难以识别出车。

3. Class activation map shows that CNNs have a very good localization ability even if they were trained only on image level labels.

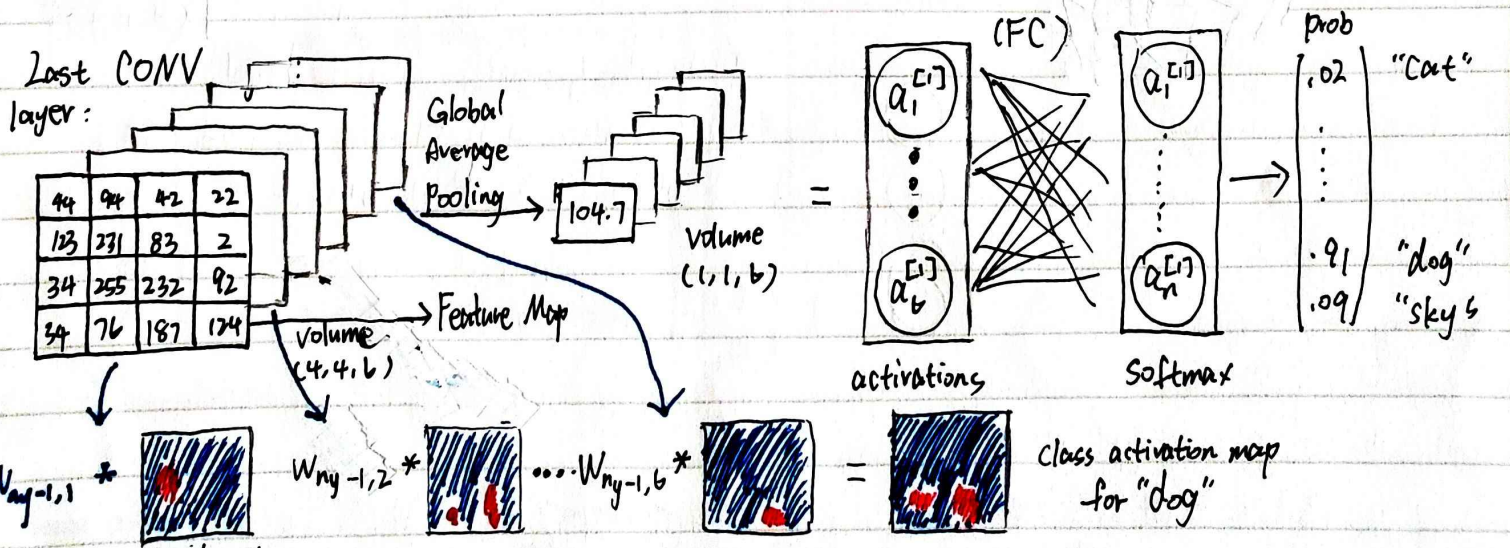
To prove that CNN is generalizing to localization:



to



* This network can be trained very quickly because we just need to train one layer



There was a visual pattern in the inputs that activated the feature map

Class activation map for "dog"

For non-image like speech detection, attention model.

Visualize NN_s from the inside:

A. With gradient ascent (class model visualization)

B. With dataset search

C. The deconvolution and its applications ✌

Deconvolution: will up-sample the height and width of an image (increase)

See Cov first:

(Goal:)

Frame the convolution as a simple matrix vector mathematical operation

Ⓐ 1D Conv:

$x = \begin{pmatrix} 0 \\ 0 \\ x_1 \\ x_2 \\ \vdots \\ x_8 \\ 0 \\ 0 \end{pmatrix}$ Input → 1 filter, whose filter size is 4, Stride = 2 (w_1, w_2, w_3, w_4) → $\begin{pmatrix} y_1 \\ \vdots \\ y_5 \end{pmatrix}$ output

The input is a one-dimensional vector which has two paddings on both sides.

$* n_y = \frac{n_x - f + 2p}{s} + 1$

$= \frac{12 - 8 + 4}{2} + 1$

$= 5$

$y_1 = w_1 0 + w_2 0 + w_3 x_1 + w_4 x_2$

$y_2 = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$

$\vdots$

$y_5 = w_1 x_7 + w_2 x_8 + w_3 0 + w_4 0$

Cov: $y = wx$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 & 0 & \cdots & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & w_4 & 0 \cdots 0 \\ & & & \ddots & & & \\ & & & & \ddots & & \\ 0 & \cdots & \cdots & 0 & w_1 & w_2 & w_3 w_4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ x_1 \\ x_2 \\ \vdots \\ x_8 \\ 0 \\ 0 \end{bmatrix}$$

For deconvolution: $x = \underbrace{w^{-1}}_{(12,5)} y$

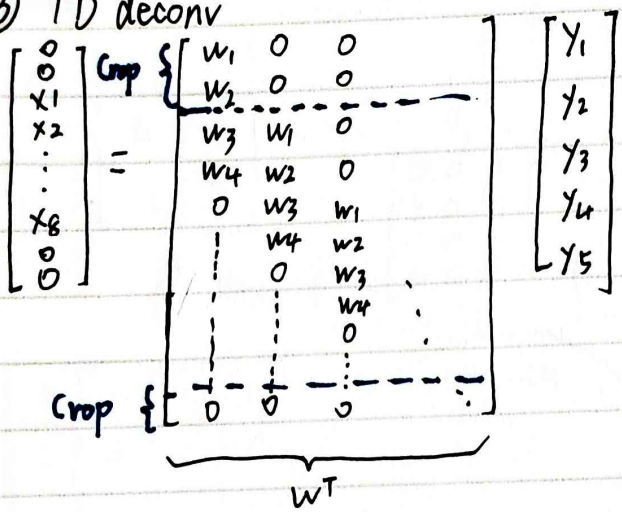
Some assumptions: ① w is an orthogonal matrix ex: filter = (1, 0, 0, -1)

$\therefore w^{-1} = w^T \quad w^T w = I \quad \Rightarrow \underline{x = w^T y}$

* often times, the assumption doesn't hold.

If you need the weights, you need the assumption, otherwise you don't need.

(B) 1D deconv

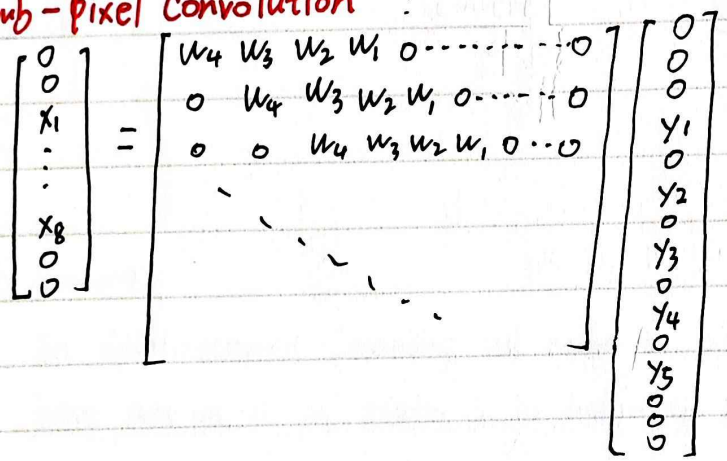


They are totally the same equations

Explain: ① In order to be able to efficiently compute the deconvolution the same way as we've learned to compute the convolution, we wanted to have the weights scattered from left to right with a stride moving from left to right. So we used a sub-pixel version of Y by inserting zeros in the middle, and we divided the stride by two. So, instead of having a stride of two as we had in our convolution, we have a stride of one in our deconvolution.

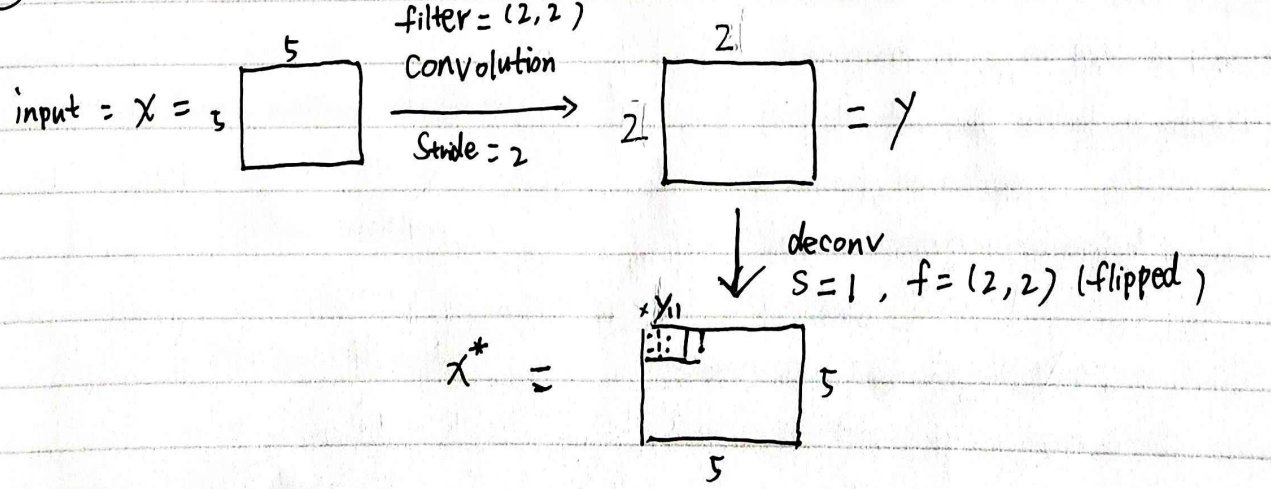
② I flipped my weights. Instead of w_1, w_2, w_3, w_4, now I have w_4, w_3, w_2, w_1

Sub-pixel convolution :

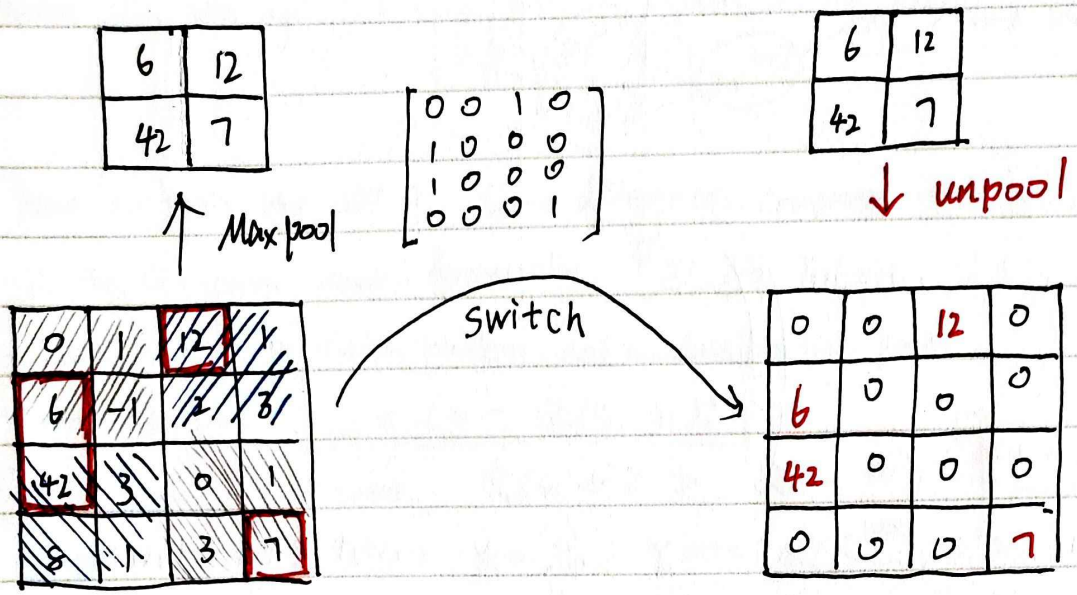


Summary:
To deconvolve, just flip all the weights, insert zeros, sub-pixel, and divide the stride

(C) 2D deconv



Unpool :



Q Learning

In Reinforcement Learning we want to obtain a function $Q(s,a)$ that predicts best action a in state s in order to maximize a cumulative reward.

This function can be estimated using Q-Learning, which iteratively updates $Q(s,a)$ using the **Bellman Equation**

$$Q(s,a) = \underbrace{r}_{\text{immediate reward}} + \underbrace{\gamma \max_{a'} Q(s',a')}_{\text{future reward}}$$

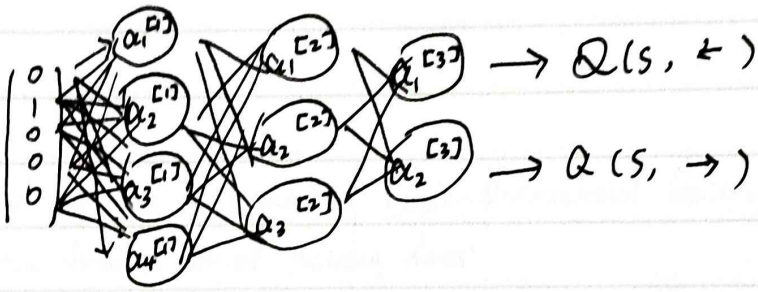
Initialize Q (table)
↓
→ Choose action from Q
↓
Perform action
↓
Measure Reward
↓
Update Q

- Q-table : matrix of entries representing "how good is it to take action a in state s"
- Policy : function telling us what's the best strategy to adopt
- Bellman equation satisfied by the optimal Q-table

Deep Q-Learning (DQN) :
Find a Q-function to replace the Q-table
↘ Neural Network

DQN

Replace Q-table by: $s = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$



How to train the NN? Some differences compared with the classic supervised learning:

① The Q-scores change dynamically ② No labels.

- The DQN is a regression problem, not a classification prob.

① **Loss function**: $L = (y - Q(s, \leftarrow))^2$

② **Target value.** Assume $Q(s, \leftarrow) > Q(s, \rightarrow)$ **Case 1**

Then define the target as $y = r_{\leftarrow} + \gamma \max(Q(s^{next}_{\leftarrow}, a'))$

↘ Immediate reward for taking action $\leftarrow$ in state s

↘ Discounted maximum future reward when you are in state $s^{next}_{\leftarrow}$

* Considering $Q(s^{next}_{\leftarrow}, a')$ fixed, until we get close to there, and our gradient is small, then we will update it and we'll fix it.

- We have two networks in parallel, one that is fixed and another is not.

Case 2

$Q(s, \leftarrow) < Q(s, \rightarrow)$

$y = r_{\rightarrow} + \gamma \max(Q(s^{next}_{\rightarrow}, a'))$

③ **Backpropagation**

Compute $\frac{\partial L}{\partial W}$ and update W using stochastic gradient descent

Summary of DQN implementation:

- Initialize your Q-network parameters
- Loop over episodes:
  - Start from initial state s.
  - Loop over time-steps:
    - Forward propagate s in the Q-network
    - Execute action a (that has the maximum $Q(s, a)$ output of Q-network)
    - Observe reward r and next state s' (immediate reward)
    - Compute targets y by forward propagating state s' in the Q-network, then compute loss.
    - Update parameters with gradient descent.

更详细的见视频 CS230 Lec 9

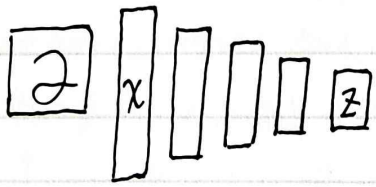
MIT 6.S191

Autoencoders:

1. Background:

① Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data.

② "Encoder" learns mapping from the data, x, to a low-dimensional latent space, z.

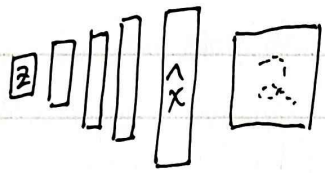


* Low-dimensional z allows us to find the most rich features in the image

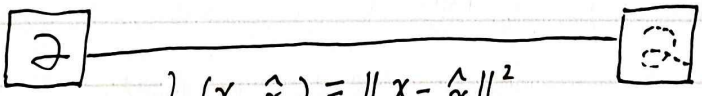
③ How can we learn this latent space?

Train the model to use these features to reconstruct the original data.

④ "Decoder" learns mapping back from latent z, to a reconstructed observation $\hat{x}$.



a blurry version of the raw image



$$L(x, \hat{x}) = \|x - \hat{x}\|^2$$

* Loss function doesn't use any labels

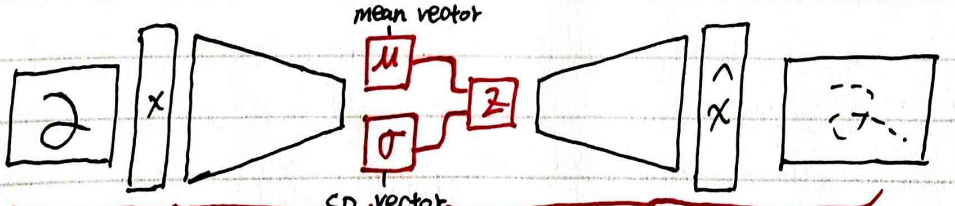
2. Summary:

1). Bottleneck hidden layer forces network to learn a compressed latent representation

2). Reconstruction loss forces the latent representation to capture (or encode) as much "information" about the data as possible

3). Autoencoding = Automatically encoding data

Variational Autoencoders (VAEs)



Encoder computes: $P_\phi(z|x)$ Decoder computes: $Q_\theta(x|z)$

VAEs are a probabilistic twist on autoencoders!

Sample from the mean and SD to compute latent sample

$$\|x - \hat{x}\|^2$$

$$\Uparrow$$

$$L(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

$$\Downarrow$$

$$D(P_\phi(z|x) \| P(z))$$

Inferred latent distribution ↳ Fixed prior on latent

$$D(P_\phi(z|x) \| P(z)) = -\frac{1}{2}\sum_{j=0}^{k-1}(\sigma_j + \mu_j^2 - 1 - \log\sigma_i)$$

Problem: We cannot backpropagate gradients through sampling layers!

Solu: Reparametrizing the sampling layer

Key idea: $z \sim N(\mu, \sigma^2)$

∵ We can't just simply sample from this distribution in a backward pass because we can't compute that backwards pass and we can't compute a channel through it. So Let's instead consider a different alternative:

Consider the sample latent vector as a sum of:

i). a fixed μ vector

ii). a fixed σ vector, scaled by random constants drawn from the prior distribution ⇒ $z = \mu + \sigma \odot \varepsilon$ * where $\varepsilon \sim N(0,1)$

LSTM : Long - Term Short Term Memory

- The idea is that a network's activations are its short-term memory and its weights are its long-term memory
- This is a kind of special RNN
- 有三种门来保持和控制细胞状态：

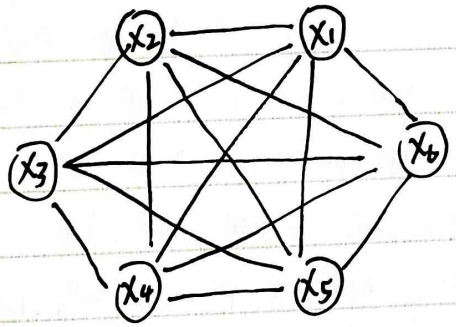
1). 决定从细胞状态中舍弃哪些信息：遗忘门 forget gate
2). 决定在细胞状态中保存哪些信息：输入门 input gate
3). 输出门 output gate

} sigmoid

DBN : Deep Belief Network

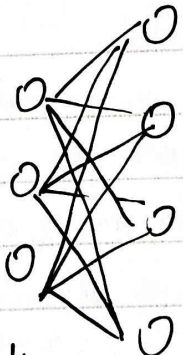
通过采用逐层训练的方式，为整个网络赋予了较好的初始权值，解决了深层次网络的优化问题。其中起重要作用的是 RBM (Restricted Boltzmann Machine).

BM:



x_1 x_2 x_3 : 可见层
x_4 x_5 x_6 : 隐层
全连接无向图
训练时间特别长

RBM:

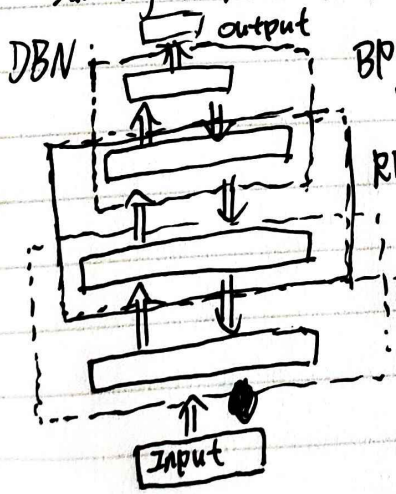


显元用于接受输入
隐元用于提取特征
RBM可以通过学习样数据表示成概率模型，还可以生成新数据

可见层 隐层
层与层全连接，层内无连接的二分图
⇒ 计算量大大减少

Summary : RBM 是非监督学习的利器，可以用于：降维，学习提取特征，自编码器，以及 DBN

DBN:



1). 最底部 RBM 以原始输入数据进行训练.
2). 将底部 RBM 抽取的特征作为顶部 RBM 的输入继续训练.
3). 重复这个过程训练尽可能多的 RBM 层

AlexNet 模型的特点：

1). 激活函数使用了 ReLU

2). 防止过拟合，使用了 Dropout 和 Data augmentation

3). 多 GPU 实现，LRN 归一化层的使用

其他：4). Overlapping Pooling 重叠池化

LRN：Local Response Normalization 局部归一化

Four modern approaches to generative modeling:

① GANs

② Reversible architectures

③ Autoregressive models (Neural language models, RNN language models)

④ Variational autoencoders

CNN vs. RNN

1). The RNN has a memory, so it can use information from all past time steps.
The CNN has a limited context. That is, the advantage of RNNs over CNNs is that their memory lets them learn arbitrary long-distance dependencies.

2). But training the RNN is very expensive since it requires a for loop over time steps.
The CNN only requires a series of convolutions

一些 ideas：

1). CNN 中，是 set stride 效果好还是 pooling 效果好？

1). Stride 要小于卷积核的 size，当 size of the kernel = 3，stride = 2 时，输出矩阵 size 降低为输入矩阵 size 的一半，可以利用这一特性代替池化层。
在 ResNet 中，每隔几层就 set the stride = 2 来压缩 size，降维的同时减小计算量

2). 对于卷积的思考：

i) Dilated convolution 空洞卷积，让卷积核看到更大的范围
Deformable convolution 可变形卷积核

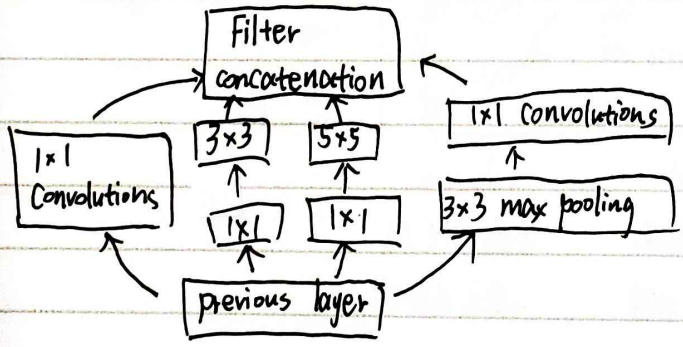
3). 网络的变迁

① AlexNet：创新点：① ReLu；② Dropout　　动态卷积过程？

② ZFNet：通过反卷积进行卷积网络可视化方法

③ GoogLeNet：利用 Inception 机制，在增大网络规模（深度+宽度）的同时防止过拟合及大量增加的计算量。

Inception 机制将多个不同尺度的卷积核，池化进行整合，形成一个 Inception 模块，对图像进行多尺度处理。



1×1 卷积，不改变图像的高度和宽度，只改变通道数
所以用它来对数据降维

为降低网络参数，用3个方法：
1). 去除最后的全连接层，用全局平均池化代替
2). 精心设计的 inception 模块

④ VGG　创新点：采用小尺寸的卷积核来模拟更大尺寸的卷积核
eg. 2层连续的 3×3 卷积层可以达到 1层 5×5 卷积层的感受野，但是所需参数量会更小。

⑤ ResNet：用跳层连接（shortcut Connections）拟合残差项（Residual Representations）
有争议 ？ 残差网络不是一个单一的超深网络，而是多个网络指数级的隐式集成，大多由一些浅层的网络组成，因此不能解决梯度消失问题

⇒ 进阶 ResNeXt

全连接层的作用：

Adding a fully-connected layer is (usually) a cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer.

⑫ DenseNet　一种具有密集连接的卷积神经网络.

优点：网络更窄，参数更少，减轻过拟合. 对小数据集合的学习有效果

一些优化算法：

1). Mini-batch Gradient Descent
2). AdaGrad (adaptive gradient) 自适应梯度
3). Adam (Adaptive moment estimation)
4). RMSProp
5). NAG

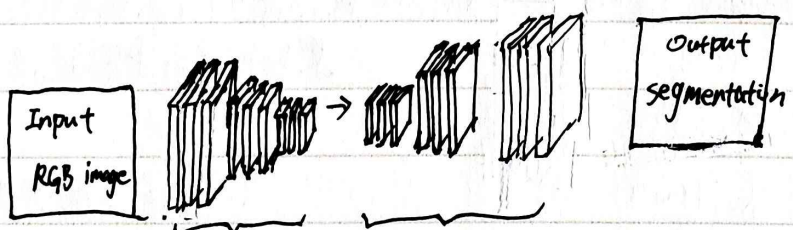
图像语义分割和图像识别：全卷积网络：能接受任意尺寸的输入图像，并产生相同尺寸的输出图像

输入图像和输出图像的像素一一对应，这种网络支持端到端、像素到像素的训练.

思路：在网络的最后，通过反卷积实现上采样

i) DeepLab 图像分割：
① 用上采样的滤波器进行卷积（atrous 卷积）以实现密集的、对像素级的预测.
② 用 atrous 空间金字塔下采样技术，以实现对物体的多尺度分割
③ 用概率图模型，实现更精准的目标边界定位.

ii) SegNet

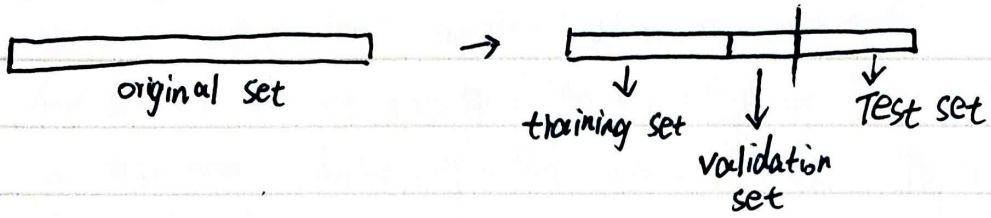


encoder：产生有语义信息的特征图像

decoder：将编码器网络输出的低分辨率特征图像映射回输入图像的尺寸

交叉验证法：用于评估模型的预测性能，在一定程度上减小过拟合

1. 留出法 holdout cross validation　　when dataset > 100,000



2. K折交叉验证 K-fold cross validation　　当数据集 < 100,000 observations

(K一般取10)

①将原始数据分为K份

②每一次选其中1份作为测试集，剩余K-1份作为训练集

③重复K次，这样每个子集都能有一次机会作为测试集，其余机会为训练集.

计算K组测试结果的平均值作为模型精度的估计

3. 留一法 Leave one out Cross Validation

把K换成M

4. Bootstrap

通过自取条样法，即在含有M个样本的数据集中进行m次有放回地随机抽样，组成新数据集作为训练集

超参数有哪些调优方法

1. 网格搜索
2. 随机搜索
3. 贝叶斯优化算法：

1. When using MSE, increasing capacity lowers bias but raises variance

2. Bayesian methods typically generalize much better when limited train data is available, but they typically suffer from high computational cost when the number of training examples is large

3. Pooling is essential for handling inputs of varying size.

4. Zero padding allows us to control the kernel width and output size independently to stop shrinkage

5. To evaluate convolutional architectures, use randomized weights and only train the last layer.

---

6. The most effective sequence models in practice are

gated RNNs { LSTMs : learn long-term dependencies more easily than simple RN
Gated recurrent units.

Adding a bias of 1 to the forget gate makes the LSTMs as strong as gated RNNs (GRN) variants

7. Good baseline optimization algorithms :
   ① SGD with momentum and a decaying learning rate
   ② ADAM

8. If there are < 10 million samples in the training set,
   ① almost always use early stopping
   ② Dropout is a good choice and batch normalization is a possible alternative

The LR is the most important hyperparameter because it controls the effective model capacity in a more complicated way than other hyperparameters

As long as your training error is low, you can always ↓ generalization error by collecting more training data.

Standardizing pixel ranges is the only strict preprocessing required for computer vision

Contrast normalization : a safe computer vision preprocessing step
- Global contrast normalization (GCN)

Weight decay = 1e-1 ⇒ only in NN

$L(x, w) = \text{mse}(M(x,w), y) + wd \cdot \Sigma w^2$

$\frac{d\, wd \cdot w^2}{w} = 2 \; wd \; w$

↑ (delete) ↓ 1e-5

L_2 regularization

~~Adam : RMSprop~~

Adam = Momentum + RMSprop

Random Forest

- A universal ML tech
- It can predict sth. that can be of any kind
  - Category ⇒ classification ⇒ classifier
  - Continuous variable ⇒ regression ⇒ regressor
- It can predict with columns of any kind (both structured and unstructured data)
- Does not generally overfit badly
- Do not need a separate validation set in general
- Has few statistical assumptions
- Requires very few pieces of feature engineering.

Deep Learning :

1. Infinitely flexible function (DL has multiple hidden layers)
2. All-purpose parameter fitting (Gradient Descent)
3. Fast and scalable (GPU)

paper.

Semantic Style Transfer and Turning Two-Bit Doodles into Fine Artworks by Alex J. Champandard

learn.fit (,)

↓ LR ↓ # of epochs

↑ # of epochs

↑ the loss of the training set

↑ the loss of the validation set

↘ accuracy

用 NN 对 cat data 和 cont data 建模：

对类别变量 (Categorical)： embeddings

对连续变量 (Continuous)： NN

MAE (L_1 损失) 对异常点有更好的鲁棒性。

MSE (L_2 损失)